# 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 mà 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 (rịch giơ đơ đy)
* Fragility (f phờ giơ lơ đỳ)
* Immobility (im mồ bớ lơ đỳ)
* Viscosity (vịt s co sơ đỳ)
* Needless complexity
* Needless repetition (rẹp pờ tí shần)
* Opacity (ố pa ci đy)

### 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 chỉ một thay đổi duy nhất mà 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 đang 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 vùng không có mối quan hệ với vùng 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,càng có khả năng thay đổi sẽ đưa ra các vấn đề bất ngờ. 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 trở nên 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 không ổn định 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

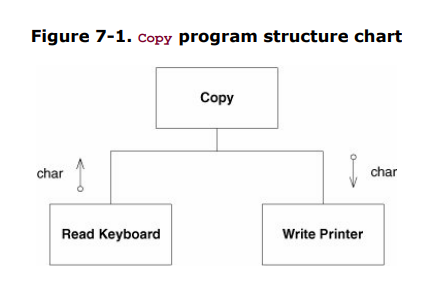
### A Familiar Scenario

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 họp 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 họp 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 đến lúc 3 giờ chiều, bạn quản lý 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. Nó biên dịch lần đầu tiên 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

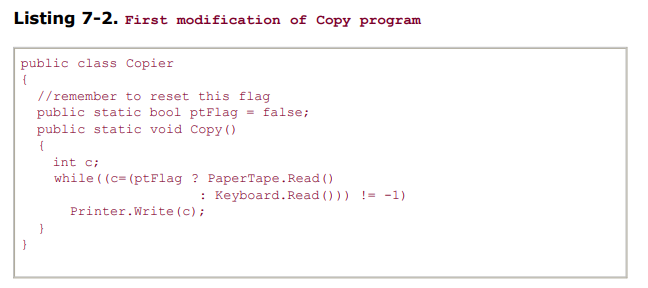
Vài tháng sau, sếp của bạn đến gặp bạn và nói rằng chương trình Copy cũng có thể đọc được từ paper tape reader. Bạn nghiến răng và đảo mắt. Bạn tự hỏi tại sao mọi người luôn thay đổi các yêu cầu. Chương trình của bạn không được thiết kế cho paper tape reader! Bạn cảnh báo sếp rằng những thay đổi như 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 kiểu Boolean vào chức năng Copy. Nếu True, nó sẽ đọc từ paper tape reader; nếu sai, nó 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 code 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.



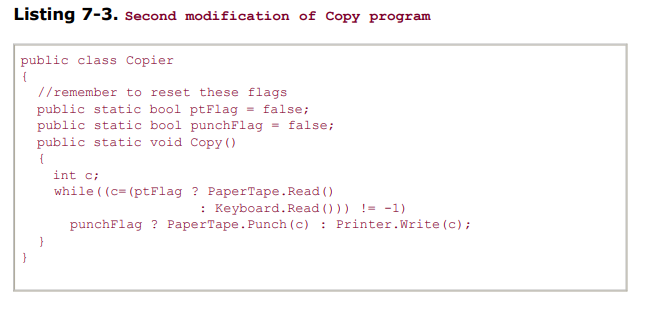
Copy đặt ptFlag thành true khi có người nhập bằng paper tape reader. Sau đó, họ có thể gọi Copy, và nó sẽ vui vẻ đọc từ paper tape reader. Khi Copy return giá trị, ptFlag được đặt lại; mặt khác, lần gọi tiếp theo có thể đọc nhầm từ đầu đọc băng giấy thay vì từ bàn phím. Để nhắc nhở các lập trình viên 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, và 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.

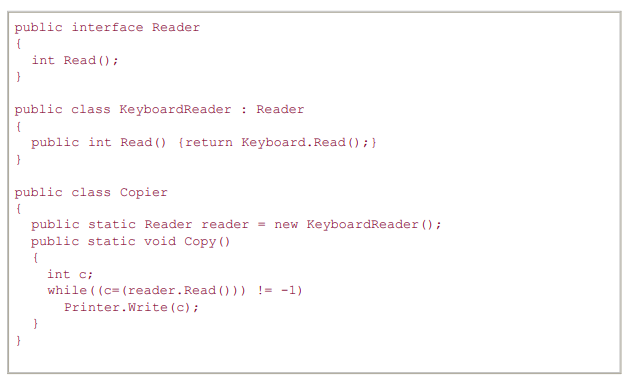
Chúng tôi có thể ngồi lại và đổ lỗi cho điều này về những thay đổi. Chúng tôi có thể phàn nàn rằng chương trình được thiết kế tốt cho thông số ban đầu và những thay đổi tiếp theo đối với thông số kỹ thuật khiến thiết kế xuống cấp. Tuy nhiên, điều này bỏ qua một trong những sự thật nổi bật nhất trong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

Hãy nhớ rằng, những thứ không ổn định 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 Copy 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 Copy 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

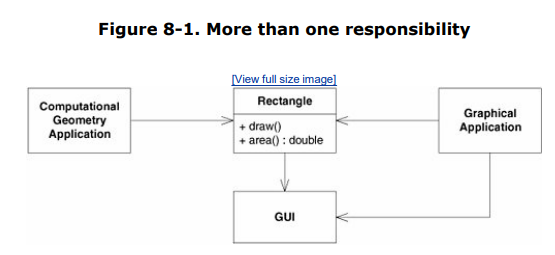
A class should have only one reason to change.

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.



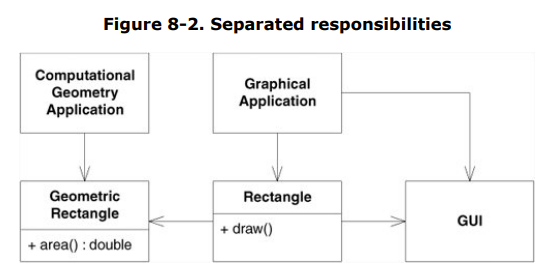
Hai ứng dụng khác nhau sử dụng lớp Rectangle. Một ứng dụng không tính toán hình học. Sử dụng Rectangle để giúp nó với các phép toán học của các hình dạng hình học nhưng không bao giờ vẽ hình chữ nhật trên màn hình. Ứng dụng khác có bản chất đồ họa và cũng có thể thực hiện một số hình học tính toán, nhưng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

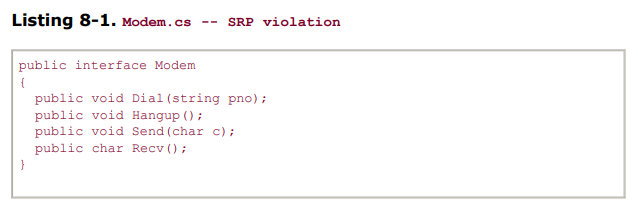
Thứ hai, nếu một sự thay đổi đối với GraphicalApplication khiến cho Rectange thay đổi vì một số lý do, sự thay đổi đó có thể buộc chúng ta phải xây dựng lại, kiểm tra lại và triển khai lại ComputationalGeometryApplication. Nếu chúng ta quên làm điều này, ứng dụng đó có thể bị hỏng theo những cách không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



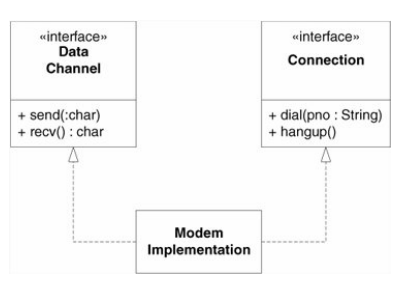
## Defining a Responsibility

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 và hợp lý. Bốn chức năng mà nó có 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 thì 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 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 lại một cách 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 lại với nhau.



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.

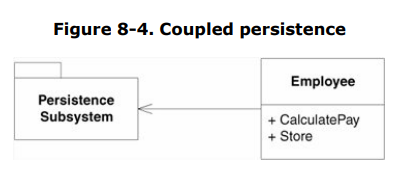
## Separating Coupled Responsibilities

Lưu ý rằng trong Hình 8-3, tôi giữ cả hai trách nhiệm được ghép nối trong ModemImplementation. 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 business rules 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, việc thực hành test-driven development 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 test 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

Single-Responsibility Principle 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 mộ t 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.

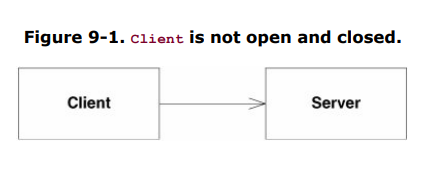
Có vẻ như hai thuộc tính này là bất hòa. Cách thông thường để mở rộng hành vi của một mô-đun là thay đổi mã nguồn của mô-đun đó. Một mô-đun không thể thay đổi thường được cho là có hành vi cố định.

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(object-oriented programming language) (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.

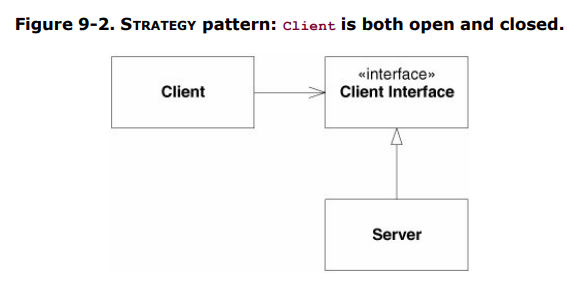
Có thể cho một mô-đun vận dụng một sự trừu tượng. Một mô-đun như vậy có thể được đóng lại để sửa đổi, vì nó phụ thuộc vào sự trừu tượng được cố định. Tuy nhiên, hành vi của mô-đun đó có thể được mở rộng bằng cách tạo ra các dẫn xuất mới của abstraction.

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). Trong trường hợp này, ClientInterface class là abstract với các abstract member functions.

The Client class sử dụng abstraction này. Tuy nhiên, đối tượng của lớp Client sẽ sử dụng đối tượng của lớp dẫn xuất Server. Nếu muốn các đối tượng Client sử dụng server class khác, một dẫn xuất mới của ClientInterface class sẽ được tạo. Client class có thể không thay đổi

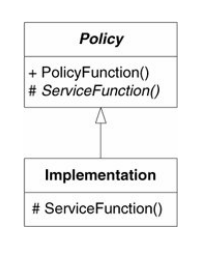


Client có một số công việc cần hoàn thành và có thể mô tả công việc đó theo abstract interface được trình bày bởi ClientInterface. Các kiểu con của Client interface có thể implement interface đó theo bất kỳ cách nào họ chọn. Do đó, hành vi được chỉ định trong Serve có thể được mở rộng và sửa đổi bằng 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 client của chúng hơn là các lớp thực hiện chúng.( abstract classes are more closely associated to their clients than to the classes that implement them.)

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 public functions 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

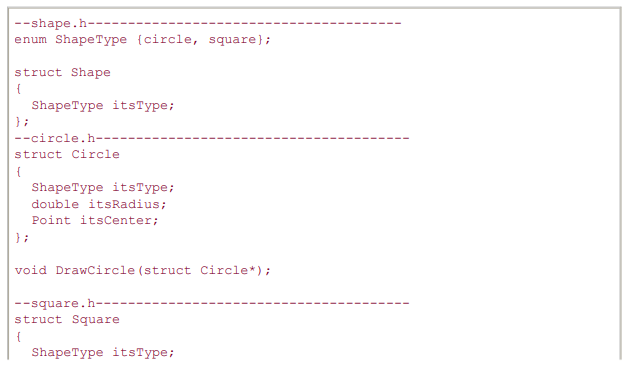
Ví dụ Shape đã được nói trong nhiều cuốn sách về thiết kế hướng đối tượng. Ví dụ nổi tiếng này thường được sử dụng để chỉ ra cách thức đa hình hoạt động. Tuy nhiên, lần này, chúng tôi sẽ sử dụng nó để làm sáng tỏ OCP.

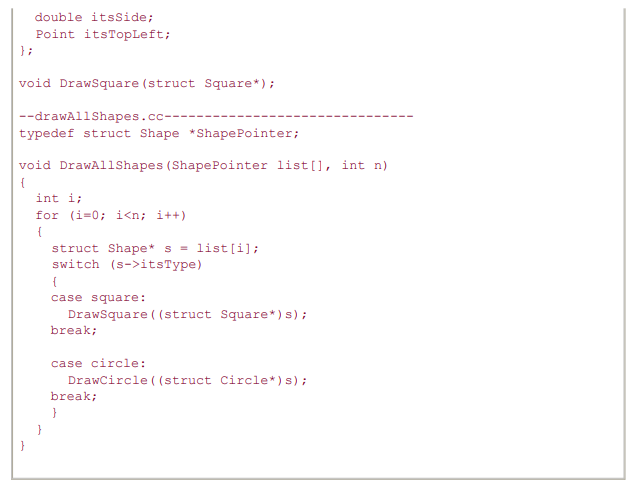
Chúng tôi có một ứng dụng phải có khả năng vẽ các vòng tròn và hình vuông trên GUI tiêu chuẩn. Các hình tròn và hình vuông phải được vẽ theo thứ tự cụ thể. Một danh sách các hình tròn và hình vuông sẽ được tạo theo thứ tự phù hợp và chương trình phải đi theo danh sách theo thứ tự đó và vẽ từng hình tròn hoặc hình vuông.

### Violating OCP

Trong C, sử dụng các kỹ thuật thủ tục (procedural ) không tuân thủ OCP, chúng tôi có thể giải quyết vấn đề này như trong Liệt kê 9-1. Ở đây, chúng ta thấy một tập hợp các cấu trúc dữ liệu có cùng phần tử đầu tiên nhưng khác nhau ngoài điều đó. Phần tử đầu tiên của mỗi phần là phần code loại xác định cấu trúc dữ liệu là Hình tròn hoặc Hình vuông. Hàm DrawAllShapes chuyển một mảng các con trỏ tới các cấu trúc dữ liệu này, kiểm tra mã loại và sau đó gọi hàm thích hợp, là DrawCircle hoặc DrawSapes.

Listing 9-1. Procedural solution to the Square/Circle problem





Vì không thể đóng đối với các hình dạng mới, nên hàm DrawAllShapes không tuân theo OCP. Nếu tôi muốn mở rộng chức năng này để có thể vẽ danh sách các hình bao gồm các hình tam giác, tôi sẽ phải sửa đổi chức năng. Thật vậy, tôi sẽ phải sửa đổi hàm cho bất kỳ hình học mới nào mà tôi muốn vẽ.

Tất nhiên, chương trình này chỉ là một ví dụ đơn giản. Trong cuộc sống thực, câu lệnh switch trong chức năng DrawAllShapes sẽ được lặp đi lặp lại nhiều lần trong các function khác nhau, với mỗi lệnh làm một điều gì đó khác nhau một chút. Có thể có một cái để kéo hình dạng, hình dạng kéo dài, hình dạng di chuyển, xóa hình dạng, v.v. Thêm một hình dạng mới vào một ứng dụng như vậy có nghĩa là săn tìm mọi nơi mà có switch hoặc chuỗi if / else tồn tại và thêm hình dạng mới cho mỗi vị trí.

Hơn nữa, rất khó có khả năng tất cả các câu lệnh switch và chuỗi if / else khác sẽ có cấu trúc dễ như chuỗi trong DrawAllShapes. Nhiều khả năng là các vị từ của câu lệnh if sẽ được kết hợp với toán tử logic hoặc các mệnh đề case của lệnh switch sẽ được kết hợp để "đơn giản hóa" việc ra quyết định cục bộ. Trong một số tình huống, các hàm có thể thực hiện chính xác những điều tương tự với Squares mà chúng thực hiện đối với Circles. Các hàm như vậy thậm chí sẽ không có các câu lệnh switch / case hoặc chuỗi if / else. Do đó, vấn đề tìm và hiểu tất cả những nơi mà hình dạng mới cần được thêm vào có thể không cần thiết.

Ngoài ra, hãy xem xét các loại thay đổi sẽ phải được thực hiện. Chúng ta phải thêm một thành viên mới vào enum ShapeType. Vì tất cả các hình dạng khác nhau phụ thuộc vào khai báo của enum này, chúng tôi sẽ phải biên dịch lại tất cả chúng. [3] Và chúng tôi cũng phải biên dịch lại tất cả các mô-đun phụ thuộc vào Shape

Vì vậy, chúng ta không chỉ phải thay đổi mã nguồn của tất cả các câu lệnh switch / case hoặc chuỗi if / else mà còn thay đổi các tệp nhị phân, thông qua việc biên dịch lại, của tất cả các mô-đun sử dụng bất kỳ cấu trúc dữ liệu Shape nào. Thay đổi các tệp nhị phân có nghĩa là bất kỳ tập hợp, DLL hoặc các loại thành phần nhị phân khác phải được triển khai lại. Hành động đơn giản của việc thêm một hình dạng mới vào ứng dụng gây ra một loạt các thay đổi tiếp theo cho nhiều mô-đun nguồn và thậm chí nhiều mô-đun nhị phân và các thành phần nhị phân hơn. Rõ ràng, tác động của việc thêm một hình dạng mới là rất lớn.

Chúng ta hãy xem lại điều này một lần nữa. Giải pháp trong Liệt kê 9-1 là cứng nhắc vì việc thêm hình tam giác làm cho Shape, Square, Circle và DrawAllShapes được biên dịch lại và triển khai lại. Giải pháp rất mong manh vì sẽ có nhiều câu lệnh switch / case khác hoặc if / else vừa khó tìm vừa khó giải mã. Giải pháp là bất động vì bất kỳ ai cố gắng sử dụng lại DrawAllShapes trong một chương trình khác đều được yêu cầu mang theo Square và Circle, ngay cả khi chương trình mới đó không cần đến chúng. Nói tóm lại, Liệt kê 9-1 thể hiện nhiều mùi của thiết kế xấu.

## Conforming to OCP

Hình 9-2 hiển thị code cho một giải pháp cho vấn đề hình vuông / hình tròn phù hợp với OCP. Trong trường hợp này, chúng tôi đã viết một lớp trừu tượng có tên Shape. Lớp trừu tượng này có một phương thức trừu tượng duy nhất có tên là Draw. Cả Circle và Square đều là dẫn xuất của lớp Shape.

Listing 9-2. OOD solution to Square/Circle problem



Lưu ý rằng nếu chúng ta muốn mở rộng hành vi của hàm DrawAllShapes trong Liệt kê 9-2 để vẽ một loại hình dạng mới, tất cả những gì chúng ta cần làm là thêm một derivative mới của lớp Shape. Hàm DrawAllShapes không cần thay đổi. Do đó, DrawAllShapes phù hợp với OCP. Hành vi của nó có thể được mở rộng mà không sửa đổi nó. Thật vậy, việc thêm một lớp tam giác hoàn toàn không có tác dụng đối với bất kỳ mô-đun nào được hiển thị ở đây. Rõ ràng, một số phần của hệ thống phải thay đổi để đối phó với lớp tam giác, nhưng tất cả các mã được hiển thị ở đây là miễn dịch với sự thay đổi.

Trong một ứng dụng thực tế, lớp Shape sẽ có nhiều phương thức hơn. Tuy nhiên, việc thêm một hình dạng mới vào ứng dụng vẫn còn khá đơn giản, vì tất cả những gì cần thiết là tạo ra derivative mới và thực hiện tất cả các chức năng của nó. Không cần phải tìm kiếm tất cả các ứng dụng, tìm kiếm những nơi yêu cầu thay đổi. Giải pháp này không dễ vỡ.

Cũng không phải là giải pháp cứng nhắc. Không có mô-đun nguồn hiện tại nào cần được sửa đổi và không có mô-đun nhị phân hiện tại nào cần được xây dựng lại với một ngoại lệ. Mô-đun tạo instances của derivative mới của Shape phải được sửa đổi. Thông thường, điều này được thực hiện bởi main, trong một số chức năng được gọi bởi main hoặc trong phương thức của một số đối tượng được tạo bởi main.

Cuối cùng, giải pháp không bất động. DrawAllShapes có thể được sử dụng lại bởi bất kỳ ứng dụng nào mà không cần phải mang Square hoặc Circle đi cùng. Do đó, giải pháp cho thấy không có thuộc tính nào của thiết kế xấu được đề cập.

Chương trình này phù hợp với OCP. Nó được thay đổi bằng cách thêm mã mới thay vì thay đổi mã hiện có. Do đó, chương trình không thực hiện hàng loạt thay đổi được thể hiện bởi các chương trình không phù hợp. Những thay đổi duy nhất cần có là bổ sung mô-đun mới và thay đổi chính liên quan cho phép các đối tượng mới được khởi tạo.

Nhưng hãy xem xét điều gì sẽ xảy ra với hàm DrawAllShapes từ Liệt kê 9-2 nếu chúng tôi quyết định rằng tất cả các Vòng tròn sẽ được vẽ trước bất kỳ Hình vuông nào. Hàm DrawAllShapes không được đóng đối với một thay đổi, như thế này. Để thực hiện thay đổi đó, chúng tôi sẽ phải truy cập vào DrawAllShapes và quét danh sách trước Circles và sau đó lại cho Hình vuông.

## Anticipation and "Natural" Structure

Nếu chúng ta dự đoán loại thay đổi này, chúng ta có thể đã tạo ra một sự trừu tượng bảo vệ chúng ta khỏi nó. Các abstractions mà chúng ta đã chọn trong Liệt kê 9-2 gây nhiều trở ngại cho loại thay đổi này hơn là một sự trợ giúp. Bạn có thể thấy điều này đáng ngạc nhiên; Rốt cuộc, điều gì có thể tự nhiên hơn một lớp cơ sở Shape với các dẫn xuất Square và Circle? Tại sao nó không phải là mô hình thực tế, tự nhiên tốt nhất để sử dụng? Rõ ràng, câu trả lời là mô hình đó không phải là tự nhiên trong một hệ thống trong đó thứ tự được kết hợp với loại hình.

Điều này dẫn chúng ta đến một kết luận đáng lo ngại. Nói chung, cho dù mô-đun "đóng" như thế nào, sẽ luôn có một số loại thay đổi mà nó không được đóng. Không có mô hình nào là tự nhiên cho tất cả bối cảnh!

Vì việc đóng cửa không thể hoàn thành, nó phải mang tính chiến lược. Nghĩa là, người thiết kế phải chọn các loại thay đổi để đóng thiết kế, phải đoán các loại thay đổi có khả năng nhất, sau đó xây dựng các abstractions để bảo vệ chống lại các thay đổi đó.

Điều này cần một lượng kinh nghiệm nhất định. Các nhà thiết kế có kinh nghiệm hy vọng rằng họ biết rõ người dùng và ngành công nghiệp đủ để đánh giá xác suất của các loại thay đổi khác nhau. Những nhà thiết kế này sau đó dùng OCP để chống lại những thay đổi có thể xảy ra nhất.

Điều này không dễ dàng. Nó giúp tạo ra những phỏng đoán có giáo dục về các loại thay đổi có khả năng mà ứng dụng sẽ phải chịu theo thời gian. Khi các nhà thiết kế đoán đúng, họ thắng. Khi họ đoán sai, họ thua. Và họ chắc chắn họ sẽ đoán sai một lần.

Ngoài ra, tuân thủ OCP là tốn kém. Phải mất thời gian phát triển và nỗ lực để tạo ra sự trừu tượng thích hợp. Những trừu tượng đó cũng làm tăng sự phức tạp của thiết kế phần mềm. Có một giới hạn về số lượng trừu tượng mà các nhà phát triển có thể chi trả. Rõ ràng, chúng tôi muốn giới hạn việc áp dụng OCP cho những thay đổi có khả năng.

Làm thế nào để chúng ta biết những thay đổi có khả năng? Chúng tôi thực hiện nghiên cứu thích hợp, chúng tôi đặt câu hỏi thích hợp và chúng tôi sử dụng kinh nghiệm và ý thức chung của chúng tôi. Và sau tất cả những điều đó, chúng tôi chờ đợi cho đến khi những thay đổi xảy ra!

## Putting the "Hooks" In

Làm thế nào để chúng ta bảo vệ bản thân khỏi những thay đổi? Trong thế kỷ trước, chúng tôi đã nói rằng chúng tôi sẽ "đưa các móc vào" cho những thay đổi mà chúng tôi nghĩ có thể xảy ra. Chúng tôi cảm thấy rằng điều này sẽ làm cho phần mềm của chúng tôi linh hoạt.

Tuy nhiên, các móc chúng tôi đưa vào thường không chính xác. Tồi tệ hơn, họ ngửi thấy sự phức tạp không cần thiết phải được hỗ trợ và duy trì, mặc dù họ không được sử dụng. Đây không phải là một điều tốt. Chúng tôi không muốn tải thiết kế với nhiều sự trừu tượng không cần thiết. Thay vào đó, chúng tôi muốn đợi cho đến khi chúng tôi cần sự trừu tượng và sau đó đặt chúng vào.

### Fool me once

"Đánh lừa tôi một lần, tôi xấu hổ với bạn. Đánh lừa tôi hai lần, tôi xấu hổ với tôi." Đây là một thái độ mạnh mẽ trong thiết kế phần mềm. Để tránh tải phần mềm của chúng tôi với sự phức tạp không cần thiết, chúng tôi có thể cho phép bản thân bị lừa một lần. Điều này có nghĩa là ban đầu chúng tôi viết mã của mình để mong nó không thay đổi. Khi một sự thay đổi xảy ra, chúng tôi implement các abstractions bảo vệ chúng tôi khỏi những thay đổi trong tương lai của loại đó. Nói tóm lại, chúng tôi lấy viên đạn đầu tiên và sau đó đảm bảo rằng chúng tôi được bảo vệ khỏi bất kỳ viên đạn nào đến từ khẩu súng cụ thể đó.

### Stimulating change

Nếu chúng tôi quyết định lấy viên đạn đầu tiên, đó là lợi thế của chúng tôi để có được những viên đạn bay sớm và thường xuyên. Chúng tôi muốn biết những loại thay đổi nào có khả năng trước khi chúng tôi ở rất xa con đường phát triển. Chúng ta càng chờ đợi lâu để tìm ra loại thay đổi nào có khả năng, thì việc tạo ra các abstractions phù hợp sẽ càng khó khăn hơn.

Do đó, chúng ta cần kích thích những thay đổi. Chúng tôi thực hiện điều này thông qua một số phương tiện được thảo luận trong Chương 2.

* Chúng tôi viết test đầu tiên. Testing là một loại sử dụng của hệ thống. Bằng cách viết các test trước, chúng tôi buộc hệ thống phải kiểm tra được. Do đó, những thay đổi trong khả năng kiểm tra sẽ không làm chúng ta ngạc nhiên sau này. Chúng tôi sẽ xây dựng các bản tóm tắt làm cho hệ thống có thể kiểm tra được. Chúng tôi có khả năng thấy rằng nhiều trong số những abstractions này sẽ bảo vệ chúng tôi khỏi các loại thay đổi khác sau này.
* Chúng tôi sử dụng chu kỳ phát triển rất ngắn: ngày thay vì tuần.
* Chúng tôi phát triển các tính năng trước cơ sở hạ tầng và thường xuyên hiển thị các tính năng đó cho các bên liên quan
* Chúng tôi phát triển các tính năng quan trọng nhất đầu tiên.
* Chúng tôi phát hành phần mềm sớm và thường xuyên. Chúng tôi đưa nó cho khách hàng và người dùng của chúng tôi nhanh nhất và thường xuyên nhất có thể

## Using Abstraction to Gain Explicit Closure

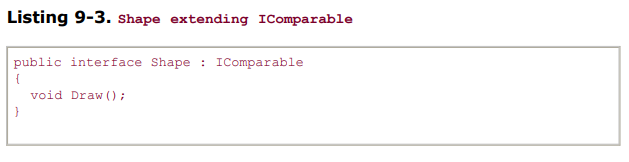
OK, chúng tôi đã lấy viên đạn đầu tiên. Người dùng muốn chúng tôi vẽ tất cả các Vòng tròn trước bất kỳ Hình vuông nào. Bây giờ chúng tôi muốn bảo vệ bản thân khỏi mọi thay đổi trong tương lai của loại đó.

Làm cách nào chúng ta có thể đóng chức năng DrawAllShapes trước những thay đổi trong thứ tự vẽ? Hãy nhớ rằng đóng dựa trên sự trừu tượng. Do đó, để đóng DrawAllShapes chống lại việc thay đổi, chúng tôi cần một số loại "ordering abstraction". Sự trừu tượng hóa này sẽ cung cấp một abstract interface mà qua đó bất kỳ chính sách sắp xếp như thế nào có thể được thể hiện.

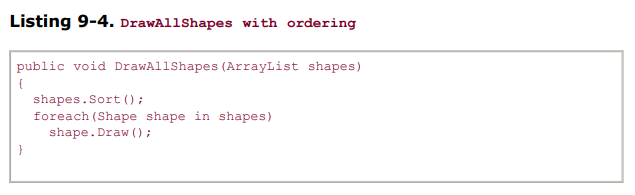
Một chính sách thứ tự ngụ ý rằng, với bất kỳ hai đối tượng nào, có thể khám phá ra cái nào nên được vẽ trước. C # cung cấp một sự trừu tượng như vậy. IComparable là một interface với một phương thức, CompareTo. Phương thức này lấy một đối tượng làm parameter và trả về -1 nếu đối tượng nhận nhỏ hơn parameter, trả về 0 nếu chúng bằng nhau và 1 nếu đối tượng nhận lớn hơn parameter

Hình 9-3 cho thấy lớp Shape có thể trông như thế nào khi nó extends the IComparable interface.

Listing 9-3. Shape extending Icomparable

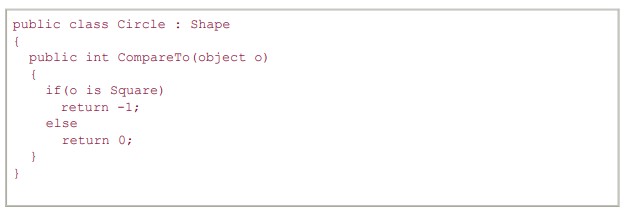


Bây giờ chúng ta có một cách để xác định thứ tự tương đối của hai đối tượng Shape, chúng ta có thể sắp xếp chúng và sau đó vẽ chúng theo thứ tự. Liệt kê 9-4 hiển thị mã C # thực hiện điều này



Điều này cho chúng ta một phương tiện để sắp xếp các đối tượng Shape và để vẽ chúng theo thứ tự thích hợp. Nhưng chúng ta vẫn chưa có một sự trừu tượng hóa trật tự. Khi nó đứng, các đối tượng Shape riêng lẻ sẽ phải ghi đè phương thức CompareTo để chỉ định thứ tự. Làm thế nào điều này sẽ làm việc? Loại mã nào chúng ta sẽ viết trong Circle.CompareTo để đảm bảo rằng các Circles được vẽ trước Squares?

Hãy xem xét Liệt kê 9-5.



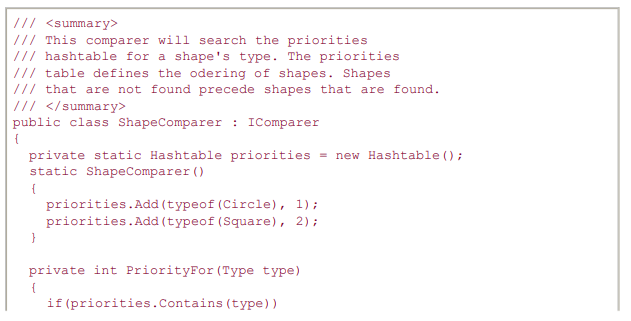
Cần phải rất rõ ràng rằng chức năng này và tất cả anh chị em của nó trong các dẫn xuất khác của Shape, không phù hợp với OCP. Không có cách nào để đóng chúng chống lại các dẫn xuất mới của Shape. Mỗi khi một dẫn xuất mới của Shape được tạo, tất cả các hàm CompareTo () sẽ cần phải được thay đổi.

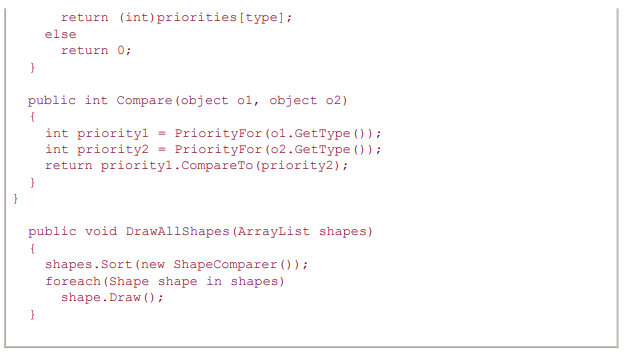
Tất nhiên, điều này không thành vấn đề nếu không có dẫn xuất mới nào của Shape được tạo ra. Mặt khác, nếu chúng được tạo ra 1 cách thường xuyên, thiết kế này sẽ gây ra một số lượng đáng kể. Một lần nữa, chúng tôi sẽ lấy viên đạn đầu tiên.

## Using a Data-Driven Approach to Achieve Closure

Nếu chúng ta phải đóng các dẫn xuất của Shape từ kiến thức của nhau, chúng ta có thể sử dụng cách tiếp cận dựa trên bảng. Liệt kê 9-6 cho thấy một khả năng

**Listing 9-6. Table driven type ordering mechanism**





Bằng cách sử dụng phương pháp này, chúng tôi đã đóng thành công chức năng DrawAllShapes chống lại các vấn đề phát sinh nói chung và từng công cụ dẫn xuất của lớp Shape chống lại việc tạo Shape derivative mới hoặc thay đổi chính sách sắp xếp lại các đối tượng Shape theo loại của chúng (ví dụ: thay đổi thứ tự mà Squares được vẽ trước).

Thứ duy nhất không được đóng theo thứ tự của các Shape khác nhau là chính bảng. Và bảng đó có thể được đặt trong mô-đun riêng của nó, tách biệt với tất cả các mô-đun khác, để thay đổi cho nó không ảnh hưởng đến bất kỳ mô-đun nào khác.

## Conclusion

Theo nhiều cách, Nguyên tắc Mở / Đóng là trung tâm của thiết kế hướng đối tượng. Sự phù hợp với nguyên tắc này là những gì mang lại lợi ích lớn nhất được yêu cầu cho công nghệ hướng đối tượng: tính linh hoạt, khả năng sử dụng lại và khả năng bảo trì. Tuy nhiên, việc tuân thủ nguyên tắc này không đạt được chỉ bằng cách sử dụng ngôn ngữ lập trình hướng đối tượng. Cũng không phải là một ý tưởng tốt để áp dụng trừu tượng tràn lan cho mọi phần của ứng dụng. Thay vào đó, nó đòi hỏi một sự cống hiến từ phía các nhà phát triển để chỉ áp dụng sự trừu tượng hóa cho những phần của chương trình thể hiện sự thay đổi thường xuyên. Chống lại sự trừu tượng sớm cũng quan trọng như chính sự trừu tượng hóa

# Chapter 10. The Liskov Substitution Principle (LSP)

Các cơ chế chính đằng sau Nguyên tắc Mở / Đóng là trừu tượng hóa và đa hình. Trong các ngôn ngữ được nhập tĩnh, chẳng hạn như C #, một trong những cơ chế chính hỗ trợ tính trừu tượng và đa hình là sự kế thừa. Đó là bằng cách sử dụng sự kế thừa mà chúng ta có thể tạo các lớp dẫn xuất implement abstract method trong các base class

Các quy tắc thiết kế chi phối việc sử dụng đặc biệt này của thừa kế là gì? Các đặc điểm của hệ thống phân cấp thừa kế tốt nhất là gì? Những cái bẫy sẽ khiến chúng ta tạo ra thứ bậc không phù hợp với OCP là gì? Đây là những câu hỏi được giải quyết bởi Liskov Substitution Principle (LSP).

The Liskov Substitution Principle

Subtypes must be substitutable for their base types.

Barbara Liskov đã viết nguyên tắc này vào năm 1988. Cô ấy nói:

Điều mong muốn ở đây là một cái gì đó giống như thuộc tính thay thế sau: Nếu với mỗi đối tượng o1 loại S có một đối tượng o2 loại T sao cho tất cả các chương trình P được xác định theo T, hành vi của P không thay đổi khi thay thế o1 với o2 thì S là một kiểu con của T.

Tầm quan trọng của nguyên tắc này trở nên rõ ràng khi bạn xem xét hậu quả của việc vi phạm nó. Giả sử rằng chúng ta có một hàm f lấy tham số của nó làm tham chiếu đến một số lớp cơ sở B.

Giả sử rằng khi được truyền cho f trong vỏ bọc của B, một số dẫn xuất D của B khiến f hoạt động sai.

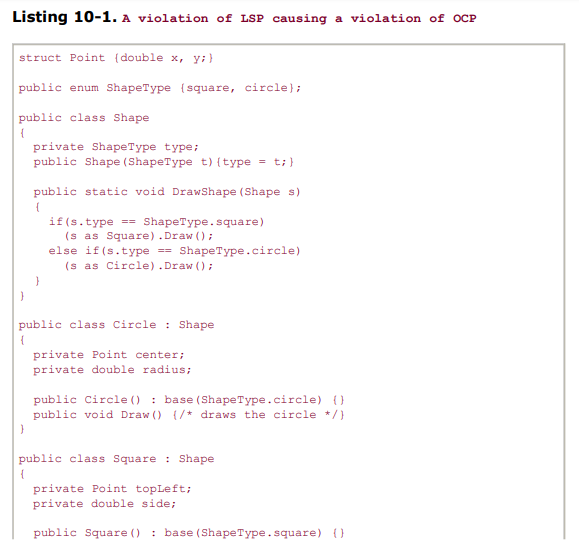
Sau đó D vi phạm LSP. Rõ ràng, D rất mong manh khi có mặt f.

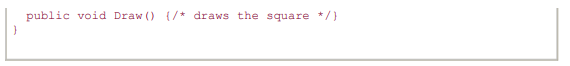
Các tác giả của f sẽ được khuyến khích đưa vào một loại thử nghiệm nào đó cho D để f có thể cư xử đúng đắn khi một D được truyền cho nó. Thử nghiệm này vi phạm OCP vì hiện tại, f không bị đóng đối với tất cả các dẫn xuất khác nhau của B. Các thử nghiệm này là mùi mã là kết quả của các nhà phát triển thiếu kinh nghiệm hoặc tệ hơn là các nhà phát triển vội vàng phản ứng với vi phạm LSP.

## Violations of LSP

### A Simple Example

Vi phạm LSP thường dẫn đến việc sử dụng runtime type checking theo cách vi phạm nghiêm trọng OCP. Thông thường, một câu lệnh if rõ ràng hoặc chuỗi if / else được sử dụng để xác định loại đối tượng để có thể chọn hành vi phù hợp với loại đó. Xem xét Liệt kê 10-1.





Rõ ràng, hàm DrawShape trong Liệt kê 10-1 vi phạm OCP. Nó phải biết về mọi dẫn xuất có thể có của lớp Shape và nó phải được thay đổi bất cứ khi nào các dẫn xuất mới của Shape được tạo.

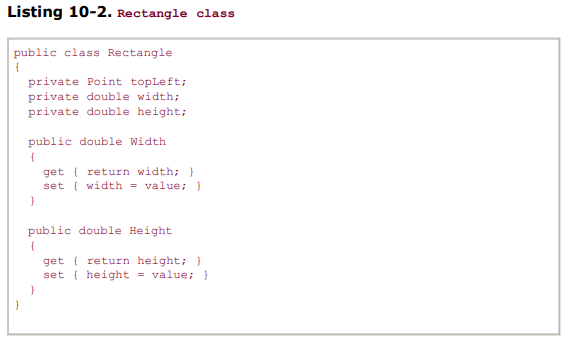
Thật vậy, nhiều người xem chính xác cấu trúc của chức năng này như là sự kết hợp với thiết kế tốt. Điều gì sẽ thúc đẩy một lập trình viên viết một chức năng như thế này?

Hãy xem xét Joe và các kỹ sư. Joe đã nghiên cứu công nghệ hướng đối tượng và đã kết luận rằng chi phí của đa hình là quá cao để trả. Do đó, ông đã định nghĩa lớp Shape mà không có bất kỳ hàm trừu tượng nào. Các lớp Square và Circle xuất phát từ Shape và có các hàm Draw (), nhưng chúng không ghi đè một hàm trong Shape. Vì Circle và Square không thể thay thế cho Shape, DrawShape phải kiểm tra Shape đến của nó, xác định loại của nó và sau đó gọi hàm Draw thích hợp

Việc Square và Circle không thể được thay thế cho Shape là vi phạm LSP. Vi phạm này đã buộc vi phạm OCP bởi DrawShape. Do đó, vi phạm LSP là vi phạm OCP tiềm ẩn.

### A More Subtle Violation

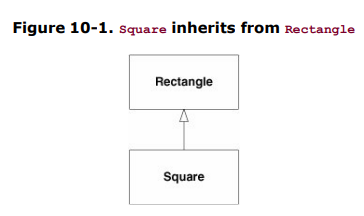
Tất nhiên, có những cách khác, vi phạm LSP tinh vi hơn nhiều. Hãy xem xét một ứng dụng sử dụng lớp Hình chữ nhật như được mô tả trong Liệt kê 10-2.



Hãy tưởng tượng rằng ứng dụng này hoạt động tốt và được cài đặt trong nhiều trang web. Như trường hợp của tất cả các phần mềm thành công, người dùng của nó yêu cầu thay đổi theo thời gian. Một ngày nọ, người dùng yêu cầu khả năng thao tác hình vuông ngoài hình chữ nhật.

Người ta thường nói rằng thừa kế là mối quan hệ IS-A. Nói cách khác, nếu một loại đối tượng mới có thể nói là hoàn thành mối quan hệ IS-A với một loại đối tượng cũ, thì lớp của đối tượng mới sẽ b xuất phát từ lớp của đối tượng cũ.

Đối với tất cả các mục đích và dự định bình thường, hình vuông là một hình chữ nhật. Vì vậy, thật hợp lý khi xem lớp Square là dẫn xuất từ lớp Rectangle. (Xem hình 10-1.)

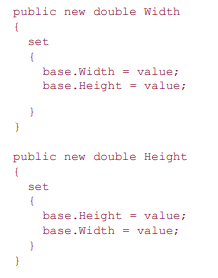


Việc sử dụng mối quan hệ IS-A này đôi khi được cho là một trong những kỹ thuật cơ bản của phân tích hướng đối tượng, một thuật ngữ thường được sử dụng nhưng hiếm khi được định nghĩa. Hình vuông là một hình chữ nhật, và do đó, lớp Square nên được bắt nguồn từ lớp Rectangle. Tuy nhiên, kiểu suy nghĩ này có thể dẫn đến một số vấn đề tinh tế nhưng quan trọng. Nói chung, những vấn đề này không lường trước được cho đến khi chúng ta thấy chúng trong code.

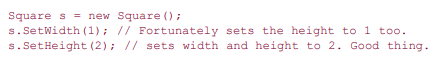
Manh mối đầu tiên của chúng tôi rằng có điều gì đó không ổn có thể là thực tế rằng Square không cần cả biến thành viên chiều cao và chiều rộng. Tuy nhiên, nó sẽ kế thừa chúng từ hình chữ nhật. Rõ ràng, điều này là lãng phí.

Trong nhiều trường hợp, sự lãng phí như vậy là không đáng kể. Nhưng nếu chúng ta phải tạo ra hàng trăm ngàn đối tượng Square như một chương trình CAD / CAE, trong đó mọi chân của mọi thành phần của một mạch phức tạp được vẽ như một hình vuông thì sự lãng phí này có thể là đáng kể.

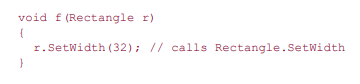
Chúng ta hãy giả sử, hiện tại, chúng ta không quan tâm lắm đến hiệu quả bộ nhớ. Các vấn đề khác xảy ra từ dẫn xuất Square đến Retangle. Square sẽ kế thừa Width and Height settter properties. Các thuộc tính này không phù hợp với Square, vì chiều rộng và chiều cao của hình vuông giống hệt nhau. Đây là một dấu hiệu mạnh mẽ rằng có một vấn đề. Tuy nhiên, có một cách để vượt qua vấn đề. Chúng ta có thể ghi đè Width và Height như sau:



Bây giờ, khi ai đó đặt chiều rộng của một đối tượng Square, chiều cao của nó sẽ thay đổi tương ứng. Và khi ai đó đặt chiều cao, chiều rộng của nó sẽ thay đổi theo. Do đó, các thuộc tính bất biến phải luôn luôn đúng bất kể trạng thái của Square vẫn còn nguyên. Đối tượng Square sẽ vẫn là một hình vuông đúng về mặt toán học:



Nhưng xem xét hàm sau



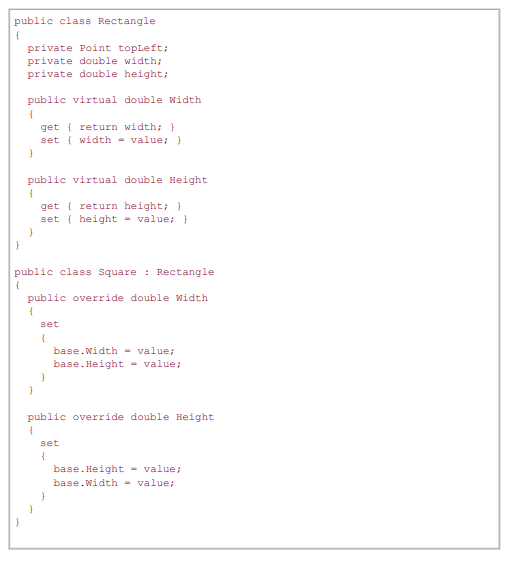
Nếu chúng ta chuyển tham chiếu đến một đối tượng Square vào chức năng này, đối tượng Square sẽ bị hỏng, vì chiều cao sẽ không bị thay đổi. Đây là một vi phạm rõ ràng về LSP. Hàm f không hoạt động cho các dẫn xuất của các đối số của nó. Lý do cho sự thất bại là Widht và Height không được khai báo virtual trong Rectangle và do đó không phải là đa hình.

Chúng ta có thể khắc phục điều này một cách dễ dàng bằng cách khai báo các thuộc tính setter là ảo. Tuy nhiên, khi việc tạo ra một lớp dẫn xuất khiến chúng ta thay đổi lớp cơ sở, nó thường ngụ ý rằng thiết kế bị lỗi.

Chắc chắn, nó vi phạm OCP. Chúng tôi có thể chống lại điều này bằng cách nói rằng việc quên làm cho chiều rộng và chiều cao ảo là lỗ hổng thiết kế thực sự và chúng tôi chỉ đơn giản là sửa nó ngay bây giờ. Tuy nhiên, điều này rất khó để biện minh, vì việc thiết lập chiều cao và chiều rộng của hình chữ nhật là các hoạt động cực kỳ nguyên thủy. Bằng lý do gì chúng ta sẽ biến chúng thành virtual nếu chúng ta không lường trước được sự tồn tại của Square?

Tuy nhiên, hãy giả sử rằng chúng tôi chấp nhận đối số và sửa các lớp. Chúng tôi kết thúc với mã trong Liệt kê 10-3.

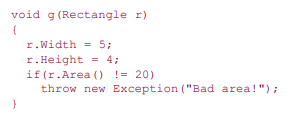
Listing 10-3. Rectangle and Square that are self consistent



#### The real problem

Square và Rectangle bây giờ xuất hiện để làm việc. Bất kể bạn làm gì với một đối tượng Square, nó sẽ vẫn phù hợp với một hình vuông toán học. Và bất kể bạn làm gì với một đối tượng Rectangle, nó sẽ vẫn là một hình chữ nhật toán học. Hơn nữa, bạn có thể chuyển Square vào một chức năng chấp nhận Rectangle và Square vẫn sẽ hoạt động như một hình vuông và sẽ vẫn nhất quán.

Vì vậy, chúng tôi có thể kết luận rằng thiết kế bây giờ là tự nhất quán và chính xác. Tuy nhiên, kết luận này sẽ không ổn. Một thiết kế tự đồng nhất không nhất thiết phải phù hợp với tất cả người dùng!



Hàm này gọi các thành viên Width và Height của cái mà nó tin là Rectangle. Hàm này chỉ hoạt động tốt đối với Rectangle nhưng ném Exception nếu là Square. Vì vậy, đây là vấn đề thực sự: Tác giả của hàm g giả định rằng việc thay đổi chiều rộng của Rectangle khiến chiều cao của nó không thay đổi.

Rõ ràng, thật hợp lý khi cho rằng việc thay đổi chiều rộng của hình chữ nhật không ảnh hưởng đến chiều cao của nó!

Tuy nhiên, không phải tất cả các đối tượng có thể được chuyển qua dưới dạng Rectangles đều thỏa mãn giả định đó. Nếu bạn chuyển một thể hiện của Square cho một hàm như g, tác giả đã đưa ra giả định đó, thì hàm đó sẽ gặp trục trặc. Hàm g rất mong manh đối với phân cấp Square/Rectangle

Hàm g cho thấy tồn tại các hàm lấy các đối tượng Rectangle nhưng không thể hoạt động đúng trên các đối tượng Rectangle. Vì, đối với các chức năng này, Square không thể thay thế cho Rectangle, mối quan hệ giữa Square và Rectangle vi phạm LSP.

Người ta có thể tranh luận rằng vấn đề nằm ở hàm g, rằng tác giả không có quyền đưa ra giả định rằng chiều rộng và chiều cao là độc lập. Tác giả của g sẽ không đồng ý. Hàm g lấy Rectagle làm đối số của nó. Có những bất biến, tuyên bố về sự thật, rõ ràng áp dụng cho một lớp có tên Rectangle, và một trong những bất biến đó là chiều cao và chiều rộng là độc lập. Tác giả của g có mọi quyền để khẳng định điều bất biến này.

Tác giả của Square đã vi phạm bất biến. Thật thú vị, tác giả của Square đã không vi phạm một bất biến của Square. Bằng cách lấy Square từ Rectangle, tác giả của Square đã vi phạm một bất biến Rectangle!

## Validity is not intrinsic

Nguyên tắc thay thế Laskov đưa chúng ta đến một kết luận rất quan trọng: Một mô hình, được xem trong sự cô lập, không thể được xác nhận một cách có ý nghĩa. (*A model, viewed in isolation, cannot be meaningfully validated* ). Tính hợp lệ của một mô hình chỉ có thể được thể hiện dưới dạng các client của nó. Ví dụ, khi chúng tôi kiểm tra phiên bản cuối cùng của các lớp Square và Rectangle trong sự cô lập, chúng tôi thấy rằng chúng tự đồng nhất và hợp lệ. Tuy nhiên, khi chúng tôi xem xét chúng từ quan điểm của một lập trình viên đã đưa ra các giả định hợp lý về lớp cơ sở, mô hình đã bị hỏng.

Khi xem xét liệu một thiết kế cụ thể có phù hợp hay không, người ta không thể chỉ xem giải pháp một cách đơn giản. Người ta phải xem nó theo các giả định hợp lý được thực hiện bởi những người sử dụng thiết kế đó.

Thông thường, bạn sẽ thấy rằng những giả định hợp lý đó được khẳng định trong các unit test được viết cho base class. Đây là một lý do tốt để thực hành test-driven development.

Ai biết được những giả định hợp lý mà người dùng thiết kế sẽ đưa ra? Hầu hết các giả định như vậy là không dễ dàng để dự đoán. Thật vậy, nếu chúng ta cố gắng lường trước tất cả, có khả năng chúng ta sẽ triển khai hệ thống của mình với mùi phức tạp không cần thiết. Do đó, như với tất cả các nguyên tắc khác, tốt nhất là trì hoãn tất cả các vi phạm LSP rõ ràng nhất cho đến khi sự mong manh liên quan đã được ngửi

## ISA is about behavior

Vậy chuyện gì đã xảy ra? Tại sao mô hình rõ ràng hợp lý của Square and Rectangle xấu đi?

Rốt cuộc, không phải là Square một Rectangle? Không phải mối quan hệ IS-A có giữ được không? Không xa như tác giả của g có liên quan! Hình vuông có thể là hình chữ nhật, nhưng theo quan điểm của g, đối tượng Square chắc chắn không phải là đối tượng Rectangle. Tại sao? Bởi vì hành vi của một đối tượng Square là không phù hợp với kỳ vọng của g về hành vi của một đối tượng Rectangle. Về mặt hành vi, Square không phải là Hình chữ nhật và đó là hành vi mà software thực sự là tất cả. LSP làm rõ rằng trong OOD, mối quan hệ IS-A liên quan đến hành vi có thể được giả định một cách hợp lý và khách hàng phụ thuộc vào.

## Design by contract

Nhiều nhà phát triển có thể cảm thấy không thoải mái với khái niệm hành vi được "giả định hợp lý".

Làm thế nào để bạn biết những gì khách hàng của bạn sẽ thực sự mong đợi? Có một kỹ thuật để làm cho những giả định hợp lý đó rõ ràng và do đó thực thi LSP. Kỹ thuật này được gọi là thiết kế theo hợp đồng (design by contract )(DBC) và được giải thích bởi Bertrand Meyer

Sử dụng DBC, tác giả của một lớp nêu rõ hợp đồng cho lớp đó. Hợp đồng thông báo cho tác giả về bất kỳ mã khách hàng nào về các hành vi có thể dựa vào. Hợp đồng được chỉ định bằng cách tuyên bố preconditions và postconditions cho mỗi phương pháp. Các preconditions phải đúng để phương thức thực thi. Khi hoàn thành, phương thức đảm bảo rằng điều kiện đúng.

Chúng ta có thể xem postcondition của Rectangle.Width setter như sau:



Trong đó old là giá trị của Rectangle trước khi gọi width. Bây giờ quy tắc cho các preconditions và postconditions của các dẫn xuất, như Meyer đã nêu, là: "Một khai báo thường quy [trong một dẫn xuất] chỉ có thể thay thế điều kiện tiên quyết ban đầu bằng một điều kiện bằng hoặc yếu hơn, và điều kiện hậu ban đầu bằng một điều kiện bằng hoặc mạnh hơn”.

Nói cách khác, khi sử dụng một đối tượng thông qua base class interface của nó, người dùng chỉ biết các preconditions và postconditions của lớp cơ sở. Vì vậy, các đối tượng dẫn xuất không được mong đợi những người dùng như vậy tuân theo các preconditions mạnh hơn các đối tượng được yêu cầu bởi lớp cơ sở. Đó là, người dùng phải chấp nhận bất cứ điều gì mà lớp cơ sở có thể chấp nhận. Ngoài ra, các lớp dẫn xuất phải phù hợp với tất cả các postconditions của cơ sở. Đó là, hành vi và đầu ra của họ không được vi phạm bất kỳ các ràng buộc được thiết lập cho lớp cơ sở. Người dùng của lớp cơ sở không được nhầm lẫn bởi đầu ra của lớp dẫn xuất

Rõ ràng, postcondition của Square.Width setter yếu hơn so với postcondition của Rectangle.Width setter, vì nó không thực thi ràng buộc (height == old.height). Do đó, thuộc tính Width của Square vi phạm hợp đồng của lớp cơ sở.

Một số ngôn ngữ, chẳng hạn như Eiffel, có hỗ trợ trực tiếp cho các preconditions và postconditions. Bạn có thể khai báo chúng và yêu cầu hệ thống thời gian chạy xác minh chúng cho bạn. C # không có tính năng như vậy. Trong C #, chúng ta phải xem xét thủ công các preconditions và postconditions của từng phương thức và đảm bảo rằng quy tắc của Meyer không bị vi phạm. Hơn nữa, nó có thể rất hữu ích để ghi lại các preconditions và postconditions trong các ý kiến cho mỗi phương pháp.

### Specifying contracts in unit tests

Hợp đồng cũng có thể được chỉ định bằng cách viết bài unit test. Bằng cách kiểm tra kỹ lưỡng hành vi của một lớp, các bài unit test làm cho hành vi của lớp rõ ràng. Tác giả của client code sẽ muốn xem xét các unit test để biết những gì cần giả định hợp lý về các lớp họ đang sử dụng.

## A Real-World Example

Quá đủ hình vuông và hình chữ nhật! LSP có liên quan đến phần mềm thực không? Hãy xem xét một trường hợp nghiên cứu xuất phát từ một dự án mà tôi đã làm việc vài năm trước.

### Motivation

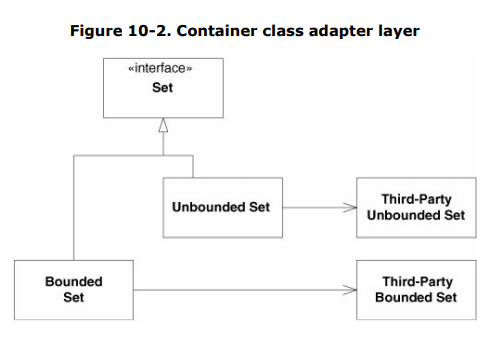
Đầu những năm 1990, tôi đã mua một thư viện của bên thứ ba có một số lớp container. Các thùng chứa gần như liên quan đến Bags và Sets của Smalltalk. Có hai loại Set và hai loại Bag tương tự nhau. Loại đầu tiên được gọi là bounded và dựa trên một mảng. Thứ hai được gọi là unbounded và được dựa trên một danh sách liên kết.

Hàm tạo cho BoundedSet đã chỉ định số lượng phần tử tối đa mà tập hợp có thể giữ. Không gian cho các phần tử này được phân chia thành một mảng trong Boundedset. Do đó, nếu việc tạo Boundedset thành công, chúng ta có thể chắc chắn rằng nó có đủ bộ nhớ. Vì nó dựa trên một mảng, nó rất nhanh. Không có phân bổ bộ nhớ được thực hiện trong quá trình hoạt động bình thường. Và vì bộ nhớ đã được phân bổ trước, chúng tôi có thể chắc chắn rằng việc vận hành Boundedset sẽ không làm cạn kiệt heap. Mặt khác, nó đã lãng phí bộ nhớ, vì nó sẽ hiếm khi sử dụng hoàn toàn tất cả không gian mà nó đã được phân bổ trước.

UnboundedSet , mặt khác, không có giới hạn khai báo về số lượng phần tử mà nó có thể giữ. Chừng nào bộ nhớ heap có sẵn, Unboundedset sẽ tiếp tục chấp nhận các phần tử. Do đó, nó rất linh hoạt. Nó cũng kinh tế ở chỗ nó chỉ sử dụng bộ nhớ cần thiết để chứa các phần tử mà nó hiện có. Nó cũng chậm, vì nó phải phân bổ và phân bổ bộ nhớ như một phần của hoạt động bình thường. Cuối cùng, một mối nguy hiểm là hoạt động bình thường của nó có thể làm cạn kiệt heap.

Tôi không hài lòng với interfaces của các lớp bên thứ ba này. Tôi không muốn code ứng dụng của mình phụ thuộc vào chúng, vì tôi cảm thấy rằng tôi muốn thay thế chúng bằng các lớp tốt hơn sau này.

Do đó, tôi đã bọc các thùng chứa của bên thứ ba trong giao diện trừu tượng của riêng mình, như trong Hình 10-2.



Tôi đã tạo một interface, đại diện là các abstract được gọi là Add, Delete, và IsMember và IsMember, như trong Liệt kê 10-4.

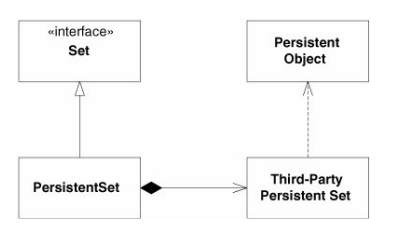
[8] Cấu trúc này thống nhất các loại unbounded và bounded của 2 bên thứ ba và cho phép chúng được truy cập thông qua một giao diện chung. Do đó, một số client có thể chấp nhận argument của loại Set và sẽ không quan tâm liệu Set thực tế mà nó hoạt động là thuộc loại bounded or unbounded . (Xem hàm Printset trong Liệt kê 10-5.)



Đó là một lợi thế lớn khi không phải biết hoặc quan tâm loại Set nào bạn đang sử dụng. Điều đó có nghĩa là lập trình viên có thể quyết định loại Set nào là cần thiết trong từng trường hợp cụ thể và không có client function nào sẽ bị ảnh hưởng bởi quyết định đó. Lập trình viên có thể chọn Unbounded khi bộ nhớ bị hạn chế và tốc độ không quan trọng hoặc có thể chọn Bounded set khi bộ nhớ dồi dào và tốc độ rất quan trọng. Các client function sẽ thao tác các đối tượng này thông qua interface của lớp cơ sở Set và do đó sẽ không biết hoặc không quan tâm đến loại Set nào chúng đang sử dụng

### Problem

Tôi muốn thêm một Persistentset vào hệ thống phân cấp này. Một Persistentset có thể được ghi ra thành một stream và sau đó đọc lại, có thể bằng một ứng dụng khác. Thật không may, container bên thứ ba duy nhất mà tôi có quyền truy cập cũng cung cấp persistence không được chấp nhận. Nó chấp nhận các đối tượng được bắt nguồn từ lớp cơ sở trừu tượng PersistentObject. Tôi đã tạo ra hệ thống phân cấp được hiển thị trong Hình 10-3.

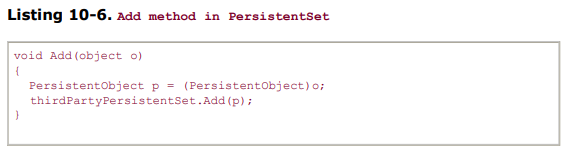


Lưu ý rằng PersistentSet chứa một thể hiện persistent set của bên thứ ba, mà nó ủy nhiệm tất cả các phương thức của nó. Do đó, nếu bạn gọi Add trên Persistentset, nó chỉ cần ủy quyền cho phương thức thích hợp của persistent set mà bên thứ ba có chứa

Nhìn bề ngoài, điều này có thể trông ổn. Tuy nhiên, có một hàm ý khá xấu xí. Các yếu tố được thêm vào persistent set của bên thứ ba phải được lấy từ PersistentObject. Do Persistentset chỉ đơn giản là ủy quyền cho persistent set của bên thứ ba, do đó, bất kỳ phần tử nào được thêm vào Persistentset đều phải xuất phát từ PersistentObject. Tuy nhiên, giao diện của Set không có ràng buộc như vậy.

Khi một máy khách đang thêm các thành viên vào base class Set, thì máy khách đó không thể chắc chắn liệu Set đó có thể là PersistentSet hay không. Do đó, khách hàng không có cách nào để biết liệu các yếu tố mà nó thêm có nên được bắt nguồn từ PersistentObject không

Hãy xem xét mã cho PersistentSet.Add () trong Liệt kê 10-6. Mã này làm rõ rằng nếu bất kỳ máy khách nào cố gắng thêm một đối tượng không xuất phát từ lớp PersistentObject vào Persistentset của tôi, một lỗi thời gian chạy sẽ xảy ra. Các cast sẽ ném một ngoại lệ. Không có ứng dụng khách hiện có nào của lớp abstract base class Set kỳ vọng các ngoại lệ sẽ được ném vào Add. Vì các hàm này sẽ bị nhầm lẫn bởi các lớp xuất của Set, nên sự thay đổi này đối với hệ thống phân cấp vi phạm LSP.



Đây co phải vân đê? Chắc chắn rồi. Các hàm chưa từng bị lỗi khi truyền dẫn xuất của Set bây giờ có thể gây ra lỗi thời gian chạy khi truyền Persistentset. Việc gỡ lỗi loại vấn đề này tương đối khó khăn, vì lỗi thời gian chạy xảy ra rất xa so với lỗ hổng logic. Lỗ hổng logic là quyết định chuyển một Persistentset vào một hàm hoặc thêm một đối tượng vào Persistentset không xuất phát từ PersistentObject. Trong cả hai trường hợp, quyết định có thể là hàng triệu hướng dẫn cách gọi phương thức Add. Tìm kiếm nó có thể là một thiệt hại. Sửa chữa nó có thể tồi tệ hơn.

### A solution that does not conform to the LSP

Làm thế nào để chúng ta giải quyết vấn đề này? Cách đây vài năm, tôi đã giải quyết nó theo quy ước, nghĩa là tôi đã không giải quyết nó trong mã nguồn. Thay vào đó, tôi đã kích hoạt một quy ước theo đó Persistentset và PersistentObject được giữ kín khỏi ứng dụng. Họ chỉ được biết đến một mô-đun cụ thể.

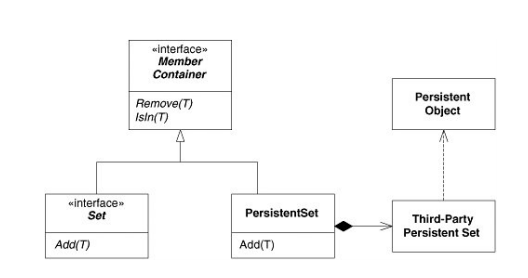
Mô-đun này chịu trách nhiệm đọc và ghi tất cả các container đến và đi từ persistent store. Khi một thùng chứa cần được viết, nội dung của nó đã được sao chép vào các dẫn xuất thích hợp của PersistentObject và sau đó được thêm vào PersistentSets, sau đó được lưu trên một luồng. Khi một container cần được đọc từ một luồng, quá trình đã được đảo ngược. Một Persistentset đã được đọc từ luồng và sau đó các PersistentObjects đã bị xóa khỏi Persistentset và được sao chép vào các đối tượng thông thường, không liên tục, sau đó được thêm vào một Set thông thường.

Giải pháp này có vẻ quá hạn chế, nhưng đó là cách duy nhất tôi có thể nghĩ đến để ngăn các đối tượng Persistentset xuất hiện tại interface của các chức năng muốn thêm các đối tượng nonpersistent vào chúng. Hơn nữa, nó đã phá vỡ sự phụ thuộc của phần còn lại của ứng dụng vào toàn bộ khái niệm về persistence

Giải pháp này có hiệu quả không? Không hẳn vậy. Qui định đã bị vi phạm trong một số phần của ứng dụng bởi các nhà phát triển không hiểu sự cần thiết cho nó. Đó là vấn đề với các quy ước: chúng phải được bán lại liên tục cho mỗi nhà phát triển. Nếu nhà phát triển chưa học được quy ước hoặc không đồng ý với nó, quy ước sẽ bị vi phạm. Và một vi phạm có thể làm nguy toàn bộ cấu trúc.

### An LSP-compliant solution

Làm thế nào tôi sẽ giải quyết điều này bây giờ? Tôi sẽ thừa nhận rằng một Persistentset không có mối quan hệ IS-A với Set, rằng nó không phải là một dẫn xuất thích hợp của Set. Vì vậy, tôi sẽ tách các thứ bậc nhưng không hoàn toàn. Set và PersistentSet có các tính năng chung. Trên thực tế, chỉ có phương pháp Add gây khó khăn với LSP. Do đó, tôi sẽ tạo một hệ thống phân cấp trong đó cả Set và Persistentset đều là anh em bên dưới một giao diện cho phép thử nghiệm thành viên, lặp lại, v.v. (xem Hình 10-4). Điều này sẽ cho phép các đối tượng Persistentset được lặp lại và được kiểm tra tư cách thành viên, v.v., nhưng sẽ không đủ khả năng để thêm các đối tượng không xuất phát từ PersistentObject vào Persistentset.



## Factoring Instead of Deriving

Một trường hợp thừa kế thú vị và khó hiểu khác là trường hợp của Line và LineSegment. Xem xét Danh sách 10-7 và 10-8. Lúc đầu, hai lớp này có vẻ là ứng cử viên tự nhiên cho sự kế thừa.

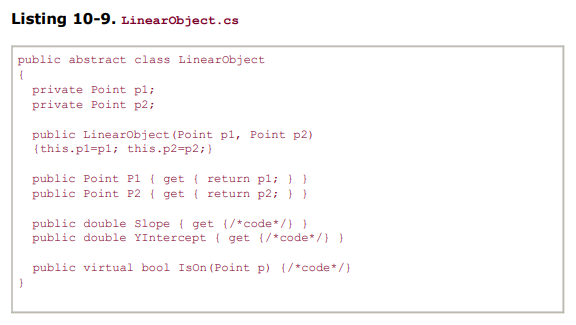
LineSegment cần mọi biến thành viên và mọi hàm thành viên được khai báo trong Line. Hơn nữa, LineSegment bổ sung một hàm thành viên mới của riêng nó, Độ dài và ghi đè lên ý nghĩa của hàm IsOn. Tuy nhiên, hai lớp này vi phạm LSP một cách tinh vi.

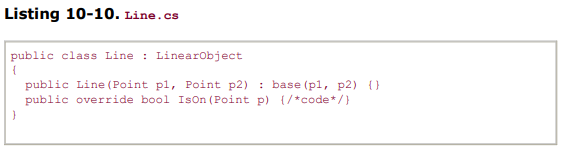


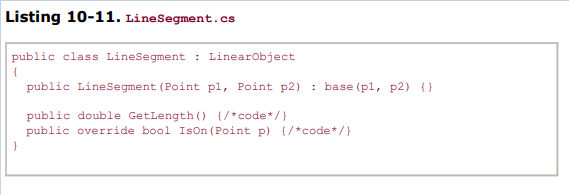
Một người dùng Line có quyền hy vọng rằng tất cả các điểm thuộc về nó đều nằm trên nó. Ví dụ: điểm được trả về bởi thuộc tính YIntercept là điểm tại đó đường thẳng cắt trục Y. Vì điểm này là colinear với dòng, người dùng Line có quyền mong đợi rằng IsOn (YIntercept) == true. Tuy nhiên, trong nhiều trường hợp của LineSegment, tuyên bố này sẽ thất bại.

Tại sao đây là một vấn đề quan trọng? Tại sao không chỉ đơn giản là lấy LineSegment từ Line và sống với những vấn đề tinh tế? Đây là một cuộc gọi phán xét. Có những trường hợp hiếm hoi khi chấp nhận một lỗ hổng tinh vi trong hành vi đa hình hơn là cố gắng điều khiển thiết kế thành tuân thủ LSP hoàn chỉnh. Chấp nhận thỏa hiệp thay vì theo đuổi sự hoàn hảo là một sự đánh đổi kỹ thuật. Một kỹ sư giỏi học được khi thỏa hiệp có lợi hơn là hoàn hảo. Tuy nhiên, không nên từ bỏ nhẹ nhàng việc tuân thủ LSP. Sự đảm bảo rằng một lớp con sẽ luôn hoạt động trong đó các lớp cơ sở của nó được sử dụng là một cách mạnh mẽ để quản lý sự phức tạp. Một khi nó bị từ bỏ, chúng ta phải xem xét từng lớp con riêng lẻ.

Trong trường hợp của Line và LineSegment, một giải pháp đơn giản minh họa một công cụ quan trọng của OOD. Nếu chúng ta có quyền truy cập vào cả hai lớp Line và LineSegment, chúng ta có thể đưa các yếu tố chung của cả hai vào một lớp cơ sở trừu tượng. Các liệt kê 10-9, 10-10 và 10-11 hiển thị factoring của Line và LineSegment vào base class LinearObject.





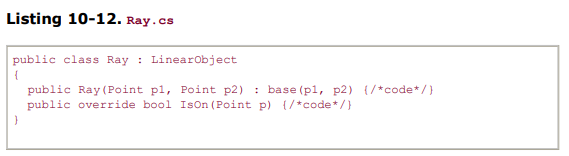


Đại diện cho cả Line và LineSegment, linearObject cung cấp hầu hết functionality và data members cho cả hai lớp con , ngoại trừ phương thức IsOn, là abstract. Người dùng của linearObject không được phép cho rằng họ hiểu phạm vi của đối tượng họ đang sử dụng. Do đó, họ có thể chấp nhận Line hoặc LineSegment mà không gặp vấn đề gì. Hơn nữa, người dùng Line sẽ không bao giờ phải đối phó với LineSegment.

Factoring là một công cụ mạnh mẽ. Nếu các phẩm chất có thể được bao gồm trong hai lớp con, thì có khả năng khác là các lớp khác sẽ xuất hiện sau đó cũng cần những phẩm chất đó. Về bao thanh toán, Rebecca Wirfs-Brock, Brian Wilkerson và Lauren Wiener nói:

* Chúng ta có thể nói rằng nếu một tập hợp các lớp đều hỗ trợ một trách nhiệm chung, thì chúng nên kế thừa trách nhiệm đó từ một superclass chung.
* Nếu một superclass phổ biến chưa tồn tại, hãy tạo một cái và chuyển các trách nhiệm chung sang nó. Rốt cuộc, một lớp như vậy là cực kỳ hữu dụng, chúng tôi đã chỉ ra rằng các trách nhiệm sẽ được kế thừa bởi một số lớp. Không thể hiểu được rằng một phần mở rộng sau này của hệ thống của bạn có thể thêm một lớp con mới sẽ hỗ trợ các trách nhiệm tương tự theo cách mới? Superclass mới này có thể sẽ là một lớp abstract

Liệt kê 10-12 cho thấy các thuộc tính của linearObject có thể được sử dụng bởi một lớp không dự đoán: Ray. Ray có thể thay thế cho một linearObject và không người dùng nào của linearObject sẽ gặp khó khăn khi xử lý nó.

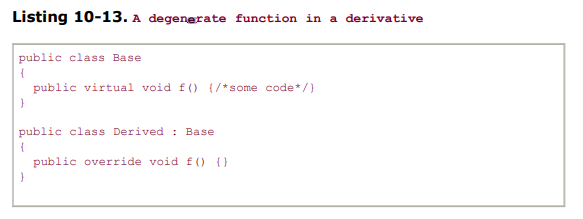


## Heuristics and Conventions

Một số heuristic đơn giản có thể cung cấp cho bạn một số manh mối về vi phạm LSP. Các heuristic này đều phải thực hiện với các derivative class bằng cách nào đó loại bỏ chức năng khỏi base class của chúng. Một derivative ít hơn base của nó thường không thể thay thế cho cơ sở đó và do đó vi phạm LSP.

Hãy xem xét hình 10-13. Hàm f trong Base được triển khai nhưng trong Derogen bị suy biến. Có lẽ, tác giả của Derive đã phát hiện ra rằng hàm f không có mục đích hữu ích trong Derogen.

Thật không may, người dùng Base không biết rằng họ không nên gọi f và do đó có vi phạm thay thế.



Sự hiện diện của các hàm suy biến trong các công cụ phái sinh không phải lúc nào cũng là dấu hiệu của vi phạm LSP, nhưng đáng để xem xét khi chúng xảy ra

## Conclusion

Open/Closed Principle là trung tâm của nhiều tuyên bố đưa ra cho thiết kế hướng đối tượng.

Khi nguyên tắc này có hiệu lực, các ứng dụng sẽ dễ bảo trì hơn, có thể tái sử dụng và mạnh mẽ hơn. Nguyên Liskov Substitution Principle là một trong những nguyên nhân chính của OCP. Khả năng thay thế của các kiểu con cho phép một mô-đun, được biểu thị dưới dạng loại cơ sở, có thể mở rộng mà không cần sửa đổi. Sự thay thế đó phải là thứ mà các nhà phát triển có thể phụ thuộc hoàn toàn. Do đó, hợp đồng của loại cơ sở phải được hiểu rõ và nổi bật, nếu không được thực thi rõ ràng, theo mã.

Thuật ngữ IS-A quá rộng để hoạt động như một định nghĩa của một kiểu con. Định nghĩa thực sự của một kiểu con là có thể thay thế, trong đó tính thay thế được xác định bởi một hợp đồng rõ ràng hoặc ẩn.

# Chapter 11. The Dependency-Inversion Principle (DIP)

The Dependency-Inversion Principle

A. Các mô-đun cấp cao không nên phụ thuộc vào các mô-đun cấp thấp. Cả hai nên phụ thuộc vào trừu tượng.

B. Trừu tượng không nên phụ thuộc vào chi tiết. Chi tiết nên phụ thuộc vào trừu tượng

Trong những năm qua, nhiều người đã đặt câu hỏi tại sao tôi sử dụng từ đảo ngược trong tên của nguyên tắc này.

Lý do là các nhiều phương pháp phát triển phần mềm truyền thống, như phân tích và thiết kế có cấu trúc, có xu hướng tạo ra các cấu trúc phần mềm trong đó các mô-đun cấp cao phụ thuộc vào các mô-đun cấp thấp và trong đó chính sách phụ thuộc vào chi tiết. Thật vậy, một trong những mục tiêu của các phương thức này là xác định hệ thống phân cấp chương trình con mô tả cách các mô-đun cấp cao thực hiện các cuộc gọi đến các mô-đun cấp thấp. Thiết kế ban đầu của chương trình Sao chép trong Hình 7-1 là một ví dụ điển hình về hệ thống phân cấp như vậy.

Cấu trúc phụ thuộc của chương trình hướng đối tượng được thiết kế tốt là "đảo ngược" đối với cấu trúc phụ thuộc thường xuất phát từ các procedural method truyền thống.

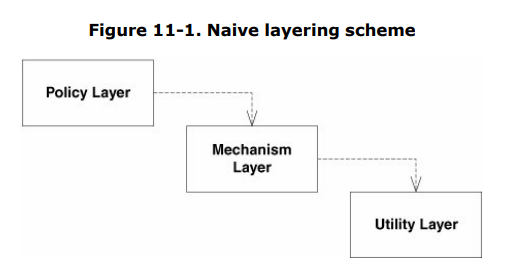
Hãy xem xét ý nghĩa của các mô-đun cấp cao phụ thuộc vào các mô-đun cấp thấp. Đây là các mô-đun cấp cao có chứa các quyết định chính sách và mô hình kinh doanh quan trọng của một ứng dụng. Các mô-đun này chứa danh tính của ứng dụng. Tuy nhiên, khi các mô-đun này phụ thuộc vào các mô-đun cấp thấp hơn, các thay đổi đối với các mô-đun cấp thấp hơn có thể có tác động trực tiếp đến các mô-đun cấp cao hơn và có thể buộc chúng thay đổi lần lượt.

Tình trạng khó khăn này là vô lý! Đó là các mô-đun thiết lập chính sách cấp cao, phải ảnh hưởng đến các mô-đun chi tiết cấp thấp. Các mô-đun có chứa các quy tắc kinh doanh cấp cao sẽ được ưu tiên hơn và độc lập với các mô-đun có chứa các chi tiết triển khai. Các mô-đun Highlevel đơn giản không nên phụ thuộc vào các mô-đun cấp thấp theo bất kỳ cách nào.

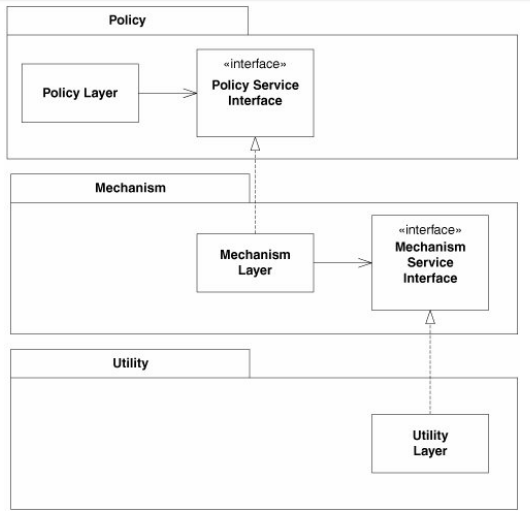
Hơn nữa, đây là mô-đun thiết lập chính sách cấp cao mà chúng tôi muốn có thể sử dụng lại. Chúng tôi đã khá giỏi trong việc sử dụng lại các mô-đun cấp thấp dưới dạng thư viện chương trình con. Khi các mô-đun cấp cao phụ thuộc vào các mô-đun cấp thấp, việc sử dụng lại các mô-đun cấp cao đó trong các bối cảnh khác nhau trở nên rất khó khăn. Tuy nhiên, khi các mô-đun cấp cao độc lập với các mô-đun cấp thấp, các mô-đun highlevel có thể được sử dụng lại khá đơn giản. Nguyên tắc này là cốt lõi của thiết kế framework

## Layering

Theo Booch, "tất cả các kiến trúc hướng đối tượng có cấu trúc tốt đều có các lớp được xác định rõ ràng, với mỗi lớp cung cấp một số dịch vụ mạch lạc thông qua interface được xác định rõ và được kiểm soát." Một cách giải thích ngây thơ về tuyên bố này có thể khiến một nhà thiết kế tạo ra một cấu trúc tương tự như Hình 11-1. Trong sơ đồ này, lớp Policy cấp cao sử dụng lớp Mechanism cấp thấp hơn, lần lượt sử dụng lớp Utility cấp chi tiết. Mặc dù điều này có vẻ phù hợp, nhưng có một đặc điểm xảo quyệt là lớp Policy rất nhạy cảm với những thay đổi hoàn toàn trong lớp Utility. Sự phụ thuộc mang tính bắc cầu. Lớp Policy phụ thuộc vào thứ gì đó phụ thuộc vào lớp Utility; do đó, lớp Policy liên tục phụ thuộc vào lớp Utility. Điều này rất đáng tiếc.



Hình 11-2 cho thấy một mô hình phù hợp hơn. Mỗi lớp cấp trên khai báo một abstract interface cho các dịch vụ mà nó cần. Các lớp cấp thấp hơn sau đó được nhận ra từ các abstract interface này. Mỗi lớp cấp cao hơn sử dụng lớp thấp nhất tiếp theo thông qua abstract interface. Vì vậy, các lớp trên không phụ thuộc vào các lớp dưới. Thay vào đó, các lớp thấp hơn phụ thuộc vào các abstract service interface được khai báo ở các lớp trên. Không chỉ sự phụ thuộc quá độ của Policy đối với UtilityLayer bị phá vỡ; mà còn là sự phụ thuộc trực tiếp PolicyLayer vào MechanismLayer.



## Ownership Inversion

Lưu ý rằng đảo ngược ở đây là một trong những phụ thuộc không chỉ mà còn là quyền sở hữu interface. Chúng ta thường nghĩ về các thư viện tiện ích như sở hữu interface của riêng họ. Nhưng khi DIP được áp dụng, chúng tôi thấy rằng các máy khách có xu hướng sở hữu các abstract interface và máy chủ của chúng bắt nguồn từ chúng.

Điều này đôi khi được gọi là nguyên tắc Hollywood: "Đừng gọi cho chúng tôi; chúng tôi sẽ gọi cho bạn." [2] Các mô-đun cấp thấp hơn cung cấp việc triển khai cho các giao diện được khai báo bên trong và được gọi bởi các mô-đun cấp trên. Sử dụng sự đảo ngược quyền sở hữu này, PolicyLayer không bị ảnh hưởng bởi bất kỳ thay đổi nào đối với MechanismLayer hoặc UtilityLayer. Hơn nữa, PolicyLayer có thể được sử dụng lại trong bất kỳ ngữ cảnh nào xác định các mô-đun cấp thấp hơn phù hợp với PolicyService-Interface. Do đó, bằng cách đảo ngược các phụ thuộc, chúng tôi đã tạo ra một cấu trúc đồng thời linh hoạt hơn, bền hơn và di động.

Trong bối cảnh này, quyền sở hữu đơn giản có nghĩa là các interfaces sở hữu được phân phối với các máy khách sở hữu chứ không phải với các máy chủ thực hiện chúng. Interface nằm trong cùng package hoặ library với máy khách. Điều này buộc server library hoặc package phụ thuộc vào client library hoặc package

Tất nhiên, có những lúc chúng tôi không muốn máy chủ phụ thuộc vào máy khách. Điều này đặc biệt đúng khi có nhiều máy khách nhưng chỉ có một máy chủ. Trong trường hợp đó, khách hàng phải đồng ý về service interface và xuất bản nó trong một gói riêng.

## Dependence on Abstractions

Một cách giải thích có phần ngây thơ hơn, nhưng vẫn rất mạnh mẽ của DIP là cách nói đơn giản: "Phụ thuộc vào sự trừu tượng". Nói một cách đơn giản, heuristic này khuyên rằng bạn không nên phụ thuộc vào một lớp cụ thể và thay vào đó, tất cả các mối quan hệ trong một chương trình nên chấm dứt trên một abstract class hoặc một interface.

* Không có biến nên giữ một tham chiếu đến một lớp cụ thể.
* Không có lớp nên xuất phát từ một lớp cụ thể.
* Không có phương thức nào nên ghi đè một phương thức đã thực hiện của bất kỳ lớp cơ sở nào của nó

Chắc chắn, heuristic này thường bị vi phạm ít nhất một lần trong mỗi chương trình. Ai đó phải tạo ra các thể hiện của các lớp cụ thể và bất kỳ mô-đun nào làm điều đó sẽ phụ thuộc vào chúng.

Hơn nữa, dường như không có lý do để theo dõi heuristic này cho các lớp cụ thể nhưng không ổn định. Nếu một lớp cụ thể sẽ không thay đổi nhiều và sẽ không có dẫn xuất tương tự nào khác được tạo ra, thì việc phụ thuộc vào nó rất ít gây hại.

Ví dụ, trong hầu hết các hệ thống, lớp mô tả một chuỗi là cụ thể. Trong C #, ví dụ, nó là chuỗi lớp cụ thể. Lớp học này không dễ bay hơi. Đó là, nó không thay đổi rất thường xuyên. Do đó, không có hại khi phụ thuộc trực tiếp vào nó.

Tuy nhiên, hầu hết các lớp cụ thể mà chúng tôi viết như một phần của chương trình ứng dụng đều không ổn định. Đó là những lớp cụ thể mà chúng tôi không muốn phụ thuộc trực tiếp vào. Sự biến động của chúng có thể được phân lập bằng cách giữ chúng phía sau một abstract interface.

Đây không phải là một giải pháp hoàn chỉnh. Đôi khi interface của một lớp dễ bay hơi phải thay đổi và thay đổi này phải được truyền đến giao diện trừu tượng đại diện cho lớp. Những thay đổi như vậy phá vỡ sự cô lập của abstract interface.

Đây là lý do mà heuristic là một chút ngây thơ. Mặt khác, nếu chúng ta xem rằng các client modules hoặc layers khai báo các service interface mà chúng cần, interface sẽ chỉ thay đổi khi máy khách cần thay đổi. Thay đổi đối với các lớp thực abstract interface sẽ không ảnh hưởng đến client.

Ví dụ, trong hầu hết các hệ thống, lớp mô tả một chuỗi là cụ thể. Trong C #, ví dụ, nó là chuỗi lớp cụ thể. Lớp học này không ổn định. Đó là, nó không thay đổi rất thường xuyên. Do đó, không có hại khi phụ thuộc trực tiếp vào nó.

Tuy nhiên, hầu hết các lớp cụ thể mà chúng tôi viết như một phần của chương trình ứng dụng đều không ổn định. Đó là những lớp cụ thể mà chúng tôi không muốn phụ thuộc trực tiếp vào. Sự biến động của chúng có thể được phân lập bằng cách giữ chúng phía sau một abstract interface.

Đây không phải là một giải pháp hoàn chỉnh. Đôi khi giao diện của một lớp không ổn định phải thay đổi và thay đổi này phải được truyền đến abstract interface đại diện cho lớp. Những thay đổi như vậy phá vỡ sự cô lập của abstract interface.

Đây là lý do mà heuristic là một chút ngây thơ. Mặt khác, nếu chúng ta rằng các mô đun hoặc lớp máy khách khai báo các service interface mà chúng cần, interface sẽ chỉ thay đổi khi client cần thay đổi. Thay đổi đối với các lớp thực hiện abstract interface sẽ không ảnh hưởng đến client.

## A Simple DIP Example

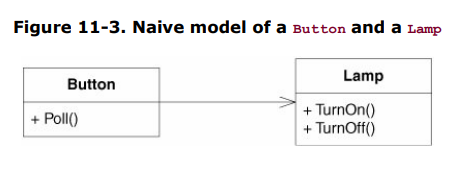
Dependency inversion có thể được áp dụng bất cứ nơi nào một lớp gửi tin nhắn cho người khác. Ví dụ, hãy xem xét trường hợp của đối tượng Button và đối tượng Lamp..

Đối tượng Button cảm nhận môi trường bên ngoài. Khi nhận được thông báo Poll, đối tượng Button xác định xem người dùng có "nhấn" nó hay không. Nó không quan trọng cơ chế cảm biến là gì.

Nó có thể là biểu tượng nút trên GUI, nút vật lý được ấn bằng ngón tay người hoặc thậm chí là máy dò chuyển động trong hệ thống an ninh gia đình. Đối tượng Button phát hiện người dùng đã kích hoạt hoặc hủy kích hoạt nó.

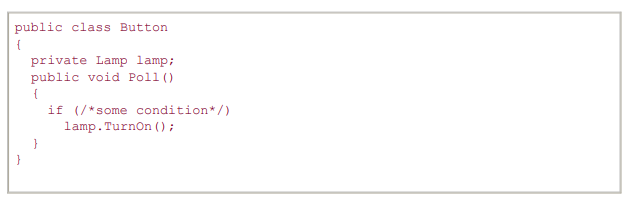
Đối tượng Lamp ảnh hưởng đến môi trường bên ngoài. Khi nhận được thông báo TurnOn, đối tượng Lamp sẽ chiếu sáng một loại ánh sáng nào đó. Khi nhận được tin nhắn TurnPack, nó sẽ tắt ánh sáng đó. Cơ chế vật lý là không quan trọng. Nó có thể là một đèn LED trên bảng điều khiển máy tính, đèn hơi thủy ngân trong bãi đậu xe hoặc thậm chí là laser trong máy in laser.

Làm thế nào chúng ta có thể thiết kế một hệ thống sao cho đối tượng Button điều khiển đối tượng Lamp? Hình 11-3 cho thấy một mô hình ngây thơ. Đối tượng Button nhận thông báo Poll, xác định xem nút đã được nhấn chưa, và sau đó chỉ cần gửi tin nhắn TurnOn hoặc TurnPack đến Lamp



Tại sao điều này là ngây thơ? Hãy xem xét mã C # được ngụ ý bởi mô hình này (Liệt kê 11-1). Lưu ý rằng lớp Button phụ thuộc trực tiếp vào lớp Lamp. Sự phụ thuộc này ngụ ý rằng Button sẽ bị ảnh hưởng bởi những thay đổi đối với Lamp. Ngoài ra, sẽ không thể sử dụng lại Button để điều khiển đối tượng Motor. Trong mô hình này, các đối tượng Button điều khiển các đối tượng Lamp và chỉ các đối tượng Lamp.

Listing 11-1. Button.cs



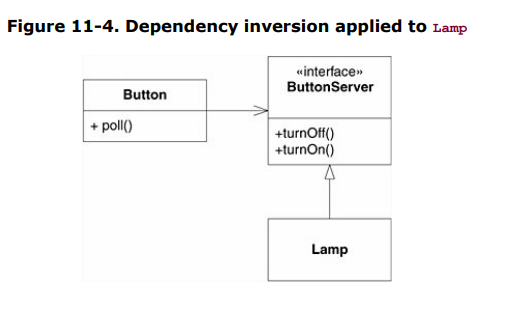
Giải pháp này vi phạm DIP. Policy ở cấp cao của ứng dụng chưa được tách biệt với việc thực hiện cấp thấp. Các abstractions chưa được tách ra khỏi các chi tiết. Không có sự phân chia như vậy, chính sách cấp cao sẽ tự động phụ thuộc vào các mô-đun cấp thấp và các abstract tự động phụ thuộc vào các chi tiết

## Finding the Underlying Abstraction

High-level policy là gì? Đó là sự trừu tượng làm nền tảng cho ứng dụng, những sự thật không thay đổi khi các chi tiết được thay đổi. Đó là hệ thống bên trong hệ thống là phép ẩn dụ. Trong ví dụ về Button/Lamp, sự trừu tượng cơ bản là phát hiện cử chỉ on/off từ người dùng và chuyển tiếp cử chỉ đó đến một đối tượng đích. Cơ chế nào được sử dụng để phát hiện cử chỉ của người dùng? Không liên quan Đối tượng mục tiêu là gì? Không liên quan! Đây là những chi tiết không ảnh hưởng đến sự trừu tượng.

Mô hình trong Hình 11-3 có thể được cải thiện bằng cách đảo ngược sự phụ thuộc vào đối tượng Lamp. Trong Hình 11-4, chúng ta thấy rằng Button hiện giữ một liên kết đến một thứ gọi là ButtonServer,

cung cấp các interface mà Button có thể sử dụng để bật hoặc tắt thứ gì đó. Lamp implements ButtonServer interface. Vì vậy, đèn hiện đang làm phụ thuộc thay vì phụ thuộc vào.( doing the depending rather than being depended on)



Thiết kế trong Hình 11-4 cho phép Button điều khiển bất kỳ thiết bị nào sẵn implement ButtonServer interface. Điều này cho chúng ta rất nhiều sự linh hoạt. Điều đó cũng có nghĩa là các đối tượng Button sẽ có thể điều khiển các đối tượng chưa được phát minh.

Tuy nhiên, giải pháp này cũng đặt ra một ràng buộc đối với bất kỳ đối tượng nào cần được điều khiển bằng Button. Một đối tượng như vậy phải implement ButtonServer interface. Điều này thật đáng tiếc, bởi vì những đối tượng này cũng có thể muốn được điều khiển bởi một đối tượng Switch hoặc một loại đối tượng nào đó không phải là Button.

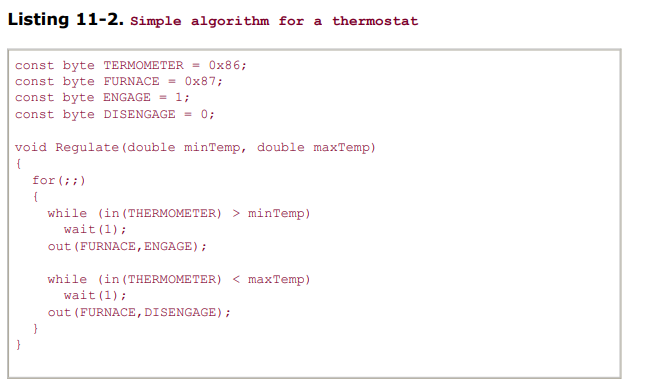
Bằng cách đảo ngược hướng của sự phụ thuộc và làm cho Lamp thực hiện tùy thuộc thay vì phụ thuộc vào, chúng tôi đã làm cho Lamp phụ thuộc vào một chi tiết khác: Button. Hay là chúng ta?

Đèn chắc chắn phụ thuộc vào ButtonServer, nhưng ButtonServer không phụ thuộc vào Button. Bất kỳ loại đối tượng nào biết cách thao tác với ButtonServer interface đều có thể điều khiển Lamp. Vì vậy, sự phụ thuộc chỉ trong tên. Và chúng ta có thể khắc phục điều đó bằng cách thay đổi tên của ButtonServer thành một cái gì đó chung chung hơn một chút, chẳng hạn như SwitchableDevice. Chúng tôi cũng có thể đảm bảo rằng Button và SwitchableDevice được giữ trong các thư viện riêng biệt, do đó việc sử dụng SwitchableDevice không bao hàm việc sử dụng Button.

Trong trường hợp này, không ai sở hữu interface. Chúng tôi có một tình huống thú vị, theo đó interface có thể được sử dụng bởi nhiều client khác nhau và được thực hiện bởi nhiều máy chủ khác nhau. Do đó, interface cần phải đứng một mình mà không thuộc về một trong hai nhóm. Trong C #, chúng tôi sẽ đặt nó trong một không gian tên và thư viện riêng

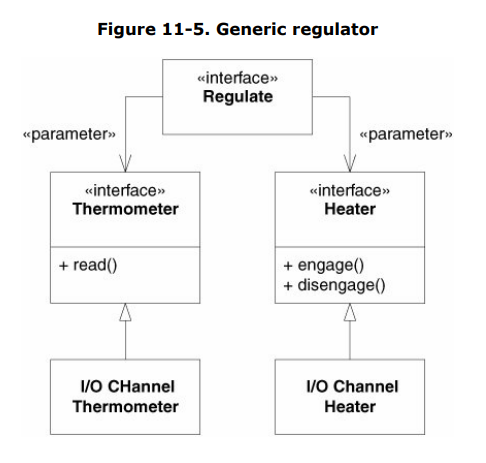
## The Furnace Example

Hãy xem xét một ví dụ thú vị hơn. Hãy xem xét phần mềm có thể điều khiển bộ điều chỉnh của lò. Phần mềm có thể đọc nhiệt độ hiện tại từ kênh I / O và hướng dẫn lò bật hoặc tắt bằng cách gửi lệnh đến kênh I / O khác. Cấu trúc của thuật toán có thể trông giống như Liệt kê 11-2.



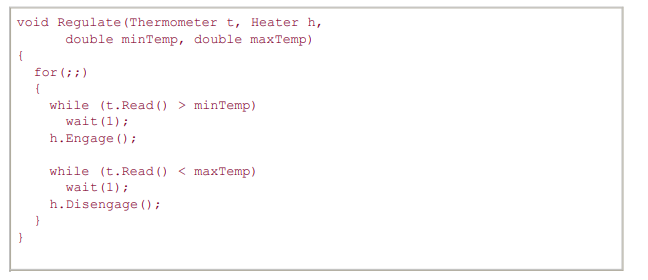
Mục đích cấp cao của thuật toán là rõ ràng, nhưng mã bị lộn xộn với nhiều chi tiết cấp thấp. Mã này không bao giờ có thể được sử dụng lại với phần cứng điều khiển khác nhau.

Điều này có thể không mất mát nhiều, vì mã rất nhỏ. Nhưng ngay cả như vậy, thật đáng tiếc khi thuật toán bị mất để sử dụng lại. Chúng tôi muốn đảo ngược các phụ thuộc và xem một cái gì đó như Hình 11-5.



Điều này cho thấy hàm Regulate có hai đối số là cả hai interface. Hermometer interface có thể được đọc và Heater interface có thể được tham gia và giải phóng. Đây là tất cả các thuật toán điều chỉnh cần. Bây giờ nó có thể được viết như trong Liệt kê 11-3 Điều này đã đảo ngược các phụ thuộc sao cho chính sách điều chỉnh mức cao không phụ thuộc vào bất kỳ chi tiết cụ thể nào của nhiệt kế hoặc lò. Thuật toán có thể tái sử dụng độc đáo.

Listing 11-3. Generic regulator



## Conclusion

Lập trình thủ tục truyền thống tạo ra một cấu trúc phụ thuộc trong đó chính sách phụ thuộc vào chi tiết. Điều này thật đáng tiếc, vì các chính sách sau đó dễ bị tổn thương trước những thay đổi trong chi tiết. Lập trình bị cản trở đảo ngược cấu trúc phụ thuộc sao cho cả chi tiết và chính sách phụ thuộc vào sự abstraction và service interface thường được sở hữu bởi khách hàng của họ.

Thật vậy, sự đảo ngược của các phụ thuộc này là đặc trưng của thiết kế hướng đối tượng tốt. Không quan trọng chương trình được viết bằng ngôn ngữ nào. Nếu phần phụ thuộc của nó bị đảo ngược, nó có thiết kế OO.

Nếu các phụ thuộc của nó không được đảo ngược, nó có một thiết kế thủ tục.

Nguyên tắc đảo ngược phụ thuộc là cơ chế cấp thấp cơ bản đằng sau nhiều lợi ích được yêu cầu cho công nghệ hướng đối tượng. Ứng dụng thích hợp của nó là cần thiết cho việc tạo ra

khung tái sử dụng. Nó cũng cực kỳ quan trọng đối với việc xây dựng mã có khả năng phục hồi để thay đổi. Do các tóm tắt và chi tiết được cách ly với nhau, nên mã dễ bảo trì hơn nhiều.

# Chapter 12. The Interface Segregation Principle (ISP)

Nguyên tắc này liên quan đến những nhược điểm của fat" interface . Các lớp mà giao diện không gắn kết thì có "fat" interface. Nói cách khác, các interface của lớp có thể được chia thành các nhóm phương thức. Mỗi nhóm phục vụ một nhóm client khác nhau. Do đó, một số client sử dụng một nhóm phương thức và các client khác sử dụng các nhóm khác.

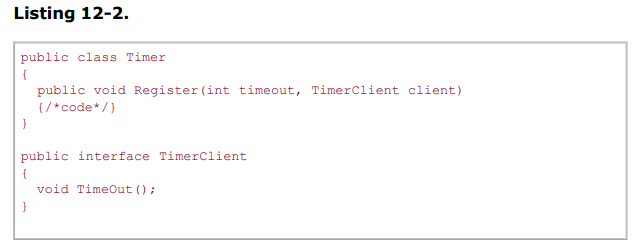
ISP thừa nhận rằng có những đối tượng yêu cầu interface không phổ biến; tuy nhiên, nó gợi ý rằng client không nên biết về họ như một lớp duy nhất. Thay vào đó, khách hàng nên biết về các abstract base classe có interface gắn kết.

## Interface Pollution

Hãy xem xét một hệ thống bảo mật trong đó các đối tượng Door có thể bị khóa và mở khóa và biết chúng đang mở hay đóng. (Xem Liệt kê 12-1.) Cửa này được mã hóa dưới dạng interface để khách hàng có thể sử dụng các đối tượng phù hợp với giao diện Door mà không phải phụ thuộc vào việc triển khai cụ thể của Door.



Bây giờ hãy xem xét rằng một triển khai như vậy, TimedDoor, cần phát ra âm thanh báo động khi cánh cửa đã bị mở quá lâu. Để thực hiện việc này, đối tượng TimedDoor liên lạc với đối tượng khác được gọi là Timer. (Xem Liệt kê 12-2.)

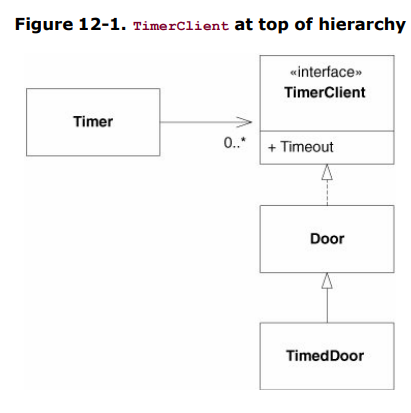


Khi một đối tượng muốn được thông báo về thời gian chờ, nó sẽ gọi chức năng Register của Timer.

Các đối số của hàm này là thời gian chờ và tham chiếu đến đối tượng TimerClient có chức năng TimeOut sẽ được gọi khi hết thời gian chờ.

Làm thế nào chúng ta có thể khiến lớp TimerClient liên lạc với lớp TimedDoor để mã trong TimedDoor có thể được thông báo về thời gian chờ? Có một số lựa chọn thay thế. Hình 12-1 cho thấy một

giải pháp chung. Chúng tôi buộc Door, và do đó TimedDoor, kế thừa từ TimerClient. Điều này đảm bảo rằng TimerClient có thể tự đăng ký với Timer và nhận thông báo TimeOut.



Vấn đề với giải pháp này là lớp Door hiện phụ thuộc vào TimerClient. Không phải tất cả các loại cửa cần thời gian. Thật vậy, Door abstraction ban đầu không liên quan gì đến thời gian. Nếu các dẫn xuất không có thời gian của Door được tạo ra, chúng sẽ phải cung cấp các triển khai suy biến cho phương thức TimeOut, một vi phạm tiềm năng của LSP. Hơn nữa, các ứng dụng sử dụng các dẫn xuất đó sẽ phải nhập định nghĩa của lớp TimerClient, mặc dù nó không được sử dụng. Điều đó có mùi của sự phức tạp không cần thiết và sự dư thừa không cần thiết.

Đây là một ví dụ về ô nhiễm interface, một hội chứng phổ biến trong các ngôn ngữ được nhập tĩnh, chẳng hạn như C #, C ++ và Java. Interface của Door đã bị ô nhiễm với một phương thức mà nó không

yêu cầu. Nó đã bị buộc phải kết hợp phương pháp này chỉ vì lợi ích của một trong các lớp con của nó. Nếu làm vậy, mỗi khi một công cụ phái sinh cần một phương thức mới, phương thức đó sẽ được thêm vào lớp cơ sở. Điều này sẽ làm ô nhiễm thêm interface của lớp cơ sở, làm cho nó "béo".

Hơn nữa, mỗi khi một phương thức mới được thêm vào lớp cơ sở, phương thức đó phải được thực hiện hoặc được phép mặc định trong các lớp dẫn xuất. Thật vậy, một thực tiễn liên quan là thêm các phương thức này vào lớp cơ sở, cho chúng thực hiện suy biến hoặc mặc định, cụ thể để các lớp dẫn xuất không bị gánh nặng với nhu cầu thực hiện chúng. Như chúng ta đã học trước đây, một thực tế như vậy có thể vi phạm LSP, dẫn đến các vấn đề về bảo trì và tái sử dụng.

## Separate Clients Mean Separate Interfaces

Door và TimerClient đại diện cho các interface được sử dụng bởi các client khác nhau. Timer sử dụng TimerClient và các lớp thao tác với cửa sử dụng Door. Vì các máy khách là riêng biệt, các giao diện cũng nên tách biệt. Tại sao? Bởi vì khách hàng tác động lực trên server interface của họ.

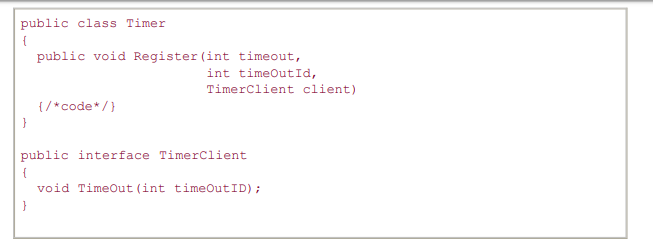
Khi chúng ta nghĩ về các lực gây ra thay đổi trong phần mềm, chúng ta thường nghĩ về việc thay đổi interface sẽ ảnh hưởng đến người dùng của họ như thế nào. Ví dụ: chúng tôi sẽ lo ngại về các thay đổi đối với tất cả người dùng TimerClient nếu interface của nó thay đổi. Tuy nhiên, có một lực lượng hoạt động theo hướng khác. Đôi khi, người dùng buộc phải thay đổi interface.

Ví dụ: một số người dùng Timer sẽ đăng ký nhiều hơn một yêu cầu hết thời gian chờ. Hãy xem xét TimedDoor. Khi phát hiện Cửa đã được mở, nó sẽ gửi thông báo Register đến Timer, yêu cầu thời gian chờ. Tuy nhiên, trước khi hết thời gian, cửa đóng lại, vẫn đóng trong một thời gian và sau đó mở lại. Điều này khiến chúng tôi phải đăng ký một yêu cầu hết thời gian mới trước khi yêu cầu cũ hết hạn. Cuối cùng, yêu cầu hết thời gian đầu tiên hết hạn và chức năng TimeOut của TimedDoor được gọi. Cửa báo động giả.

Chúng ta có thể sửa tình huống này bằng cách sử dụng quy ước được hiển thị trong Liệt kê 12-3. Chúng tôi bao gồm một mã timeOutId duy nhất trong mỗi lần đăng ký hết thời gian và lặp lại mã đó trong lệnh gọi TimeOut tới TimerClient. Điều này cho phép mỗi dẫn xuất của TimerClient biết yêu cầu hết thời gian chờ nào đang được đáp ứng.

Rõ ràng, thay đổi này sẽ ảnh hưởng đến tất cả người dùng TimerClient. Chúng tôi chấp nhận điều này, vì việc thiếu thời gianOutId là một sự giám sát cần điều chỉnh. Tuy nhiên, thiết kế trong Hình 12-1 cũng sẽ khiến Door và tất cả các khách hàng của Door bị ảnh hưởng bởi bản sửa lỗi này! Điều này có mùi của độ cứng và độ nhớt. Tại sao một lỗi trong TimerClient có bất kỳ ảnh hưởng nào đến clients của Door derivatives không yêu cầu thời gian? Loại phụ thuộc lẫn nhau kỳ lạ này làm lạnh khách hàng và người quản lý đến tận xương. Khi một sự thay đổi trong chương trình của chương trình ảnh hưởng đến các phần khác, hoàn toàn không liên quan đến chương trình, chi phí và hậu quả của những thay đổi trở nên khó lường và nguy cơ thất bại từ thay đổi sẽ tăng lên theo chiều hướng.

Listing 12-3. Timer with ID



The Interface Segregation Principle

Clients should not be forced to depend on methods they do not use.

Khi khách hàng bị buộc phải phụ thuộc vào các phương thức họ không sử dụng, những khách hàng đó có thể thay đổi các phương thức đó. Điều này dẫn đến một khớp nối vô tình giữa tất cả các khách hàng. Nói một cách khác, khi một khách hàng phụ thuộc vào một lớp có chứa các phương thức mà khách hàng không sử dụng nhưng các khách hàng khác sử dụng, thì khách hàng đó sẽ bị ảnh hưởng bởi những thay đổi mà các khách hàng khác buộc trên lớp.

Chúng tôi muốn tránh các khớp nối như vậy nếu có thể, và vì vậy chúng tôi muốn tách các giao diện.

## Class Interfaces versus Object Interfaces

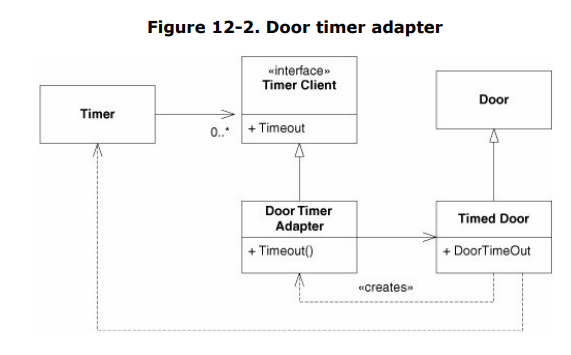
Hãy xem xét TimedDoor một lần nữa. Đây là một đối tượng có hai interface riêng được sử dụng bởi hai client riêng biệt: Timer và người dùng của Door. Hai interface này phải được thực hiện trong cùng một đối tượng, vì việc thực hiện cả hai interface thao tác cùng một dữ liệu. Làm thế nào chúng ta có thể phù hợp với ISP? Làm thế nào chúng ta có thể tách các interface khi chúng phải ở cùng nhau?

Câu trả lời nằm ở chỗ các client của một đối tượng không cần truy cập nó thông qua interface của đối tượng. Thay vào đó, họ có thể truy cập nó thông qua ủy quyền hoặc thông qua một lớp cơ sở của đối tượng.

### Separation Through Delegation

Một giải pháp là tạo một đối tượng derives từ TimerClient và delegates cho TimedDoor.

Hình 12-2 cho thấy giải pháp này. Khi nó muốn đăng ký một yêu cầu hết thời gian với Timer, TimedDoor tạo ra DoorTimerAdapter và đăng ký nó với Timer. Khi Timer gửi tin nhắn TimeOut đến DoorTimerAdapter, DoorTimerAdapter sẽ ủy quyền tin nhắn trở lại TimedDoor.



Giải pháp này phù hợp với ISP và ngăn việc ghép các Door client với Timer. Ngay cả khi thay đổi thành Timer hiển thị trong Liệt kê 12-3 được thực hiện, không ai trong số những người dùng của Door sẽ bị ảnh hưởng.

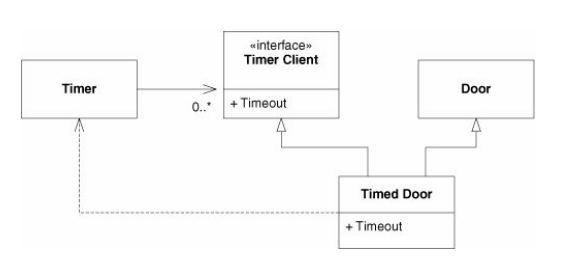
Hơn nữa, TimedDoor không phải có interface chính xác như TimerClient DoorTimerAdapter có thể dịch interface TimerClient sang interface TimedDoor. Vì vậy, đây là một giải pháp rất chung chung. (Xem Liệt kê 12-4.)



Tuy nhiên, giải pháp này cũng có phần không phù hợp. Nó liên quan đến việc tạo ra một đối tượng mới mỗi khi chúng ta muốn đăng ký thời gian chờ. Hơn nữa, delegation yêu cầu rất nhỏ, nhưng vẫn không khác, số lượng thời gian chạy và bộ nhớ. Trong một số miền ứng dụng, chẳng hạn như các hệ thống điều khiển thời gian thực được nhúng, thời gian chạy và bộ nhớ đủ khan hiếm để khiến điều này trở thành mối quan tâm.

### Separation Through Multiple Inheritance

Hình 12-3 và Liệt kê 12-5 cho thấy cách sử dụng nhiều kế thừa để đạt được ISP. Trong mô hình này, TimedDoor kế thừa từ cả Door và TimerClient. Mặc dù các máy khách của cả hai lớp cơ sở đều có thể sử dụng TimedDoor, nhưng không phụ thuộc vào lớp TimedDoor. Vì vậy, họ sử dụng cùng một đối tượng thông qua các giao diện riêng biệt.

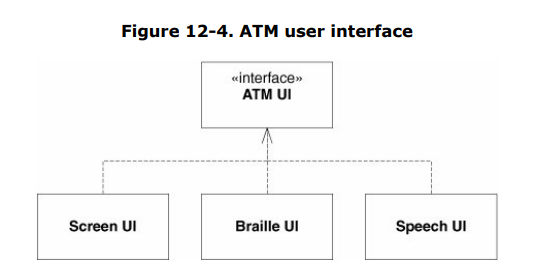




Giải pháp này là sở thích bình thường của tôi. Lần duy nhất tôi sẽ chọn giải pháp trong Hình 12-2 so với Hình 12-3 là nếu bản dịch được thực hiện bởi đối tượng DoorTimerAdapter là cần thiết hoặc nếu cần các bản dịch khác nhau vào các thời điểm khác nhau.

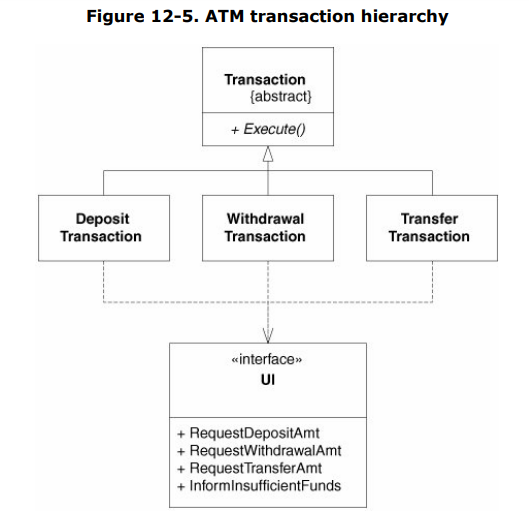
## The ATM User Interface Example

Bây giờ hãy xem xét một ví dụ quan trọng hơn một chút: sự cố máy rút tiền tự động truyền thống (ATM). Giao diện người dùng của ATM cần phải rất linh hoạt. Đầu ra có thể cần được dịch sang nhiều ngôn ngữ khác nhau và nó có thể cần được trình bày trên màn hình, trên máy tính bảng chữ nổi hoặc nói ra một bộ tổng hợp giọng nói (Hình 12-4). Rõ ràng, tính linh hoạt này có thể đạt được bằng cách tạo abstract base class có các phương thức trừu tượng cho tất cả các thông điệp khác nhau cần được trình bày bởi interface



Cũng xem xét rằng mỗi giao dịch mà ATM có thể thực hiện được gói gọn như derivative của class transaction. Vì vậy, chúng tôi có thể có các lớp như DepositTransaction, WithdrawalTransaction,

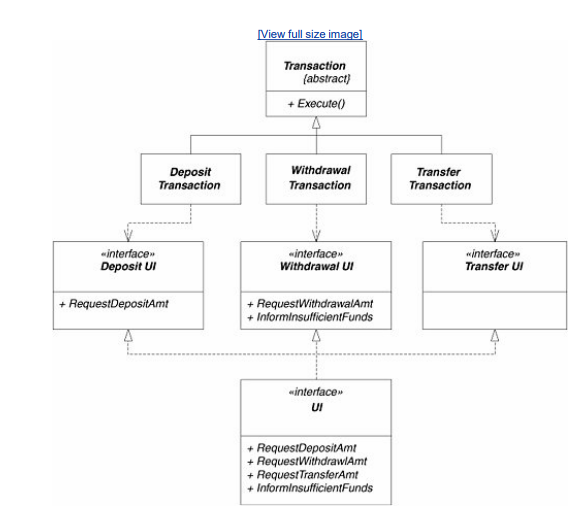
transferTransaction, v.v. Mỗi lớp gọi các phương thức UI. Ví dụ, để yêu cầu người dùng nhập số tiền cần gửi, đối tượng DepositTransaction gọi phương thức RequestDepositAmount của lớp UI. Tương tự, để hỏi người dùng cần chuyển bao nhiêu tiền giữa các tài khoản, đối tượng transferTransaction gọi phương thức UITransferAmount của UI. Điều này tương ứng với sơ đồ trong Hình 12-5..



Lưu ý rằng đây chính xác là tình huống mà ISP bảo chúng tôi tránh. Mỗi giao dịch đang sử dụng các phương thức UI mà không có lớp nào khác sử dụng. Điều này tạo ra khả năng thay đổi một trong các derivatives của TRansaction sẽ buộc thay đổi tương ứng với UI, do đó ảnh hưởng đến tất cả các derivatives khác của giao dịch và mọi lớp khác phụ thuộc vào giao diện UI. Một cái gì đó có mùi như cứng nhắc và mong manh xung quanh đây.

Ví dụ: nếu chúng tôi thêm PayGasBillTransaction, chúng tôi sẽ phải thêm các phương thức mới vào UI để xử lý các thông báo duy nhất mà giao dịch này muốn hiển thị. Thật không may, vì DepositTransaction, WithdrawalTransaction và transferTransaction đều phụ thuộc vào giao diện UI, tất cả chúng đều có khả năng được xây dựng lại. Tồi tệ hơn, nếu tất cả các giao dịch được triển khai như các thành phần trong các hội đồng riêng biệt, các hội đồng đó rất có thể phải được triển khai lại, mặc dù không có logic nào của chúng bị thay đổi. Bạn có thể ngửi thấy độ nhớt?

Sự ghép đôi đáng tiếc này có thể tránh được bằng cách tách giao diện UI thành các giao diện riêng lẻ, chẳng hạn như DepositUI, WithdrawUI và TRansferUI. Các giao diện riêng biệt này sau đó có thể được nhân rộng vào giao diện UI cuối cùng. Hình 12-6 và Liệt kê 12-6 hiển thị mô hình này



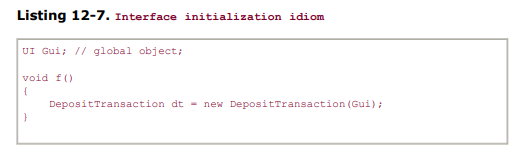
Bất cứ khi nào một derivative mới của transaction class được tạo, một base class tương ứng cho abstract UI interface sẽ là cần thiết, và do đó interface UI và tất cả các dẫn xuất của nó phải thay đổi.

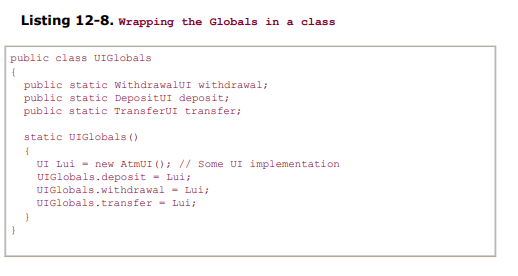
Tuy nhiên, các lớp này không được sử dụng rộng rãi. Thật vậy, chúng có lẽ chỉ được sử dụng bởi chính hoặc bất kỳ quy trình nào khởi động hệ thống và tạo ra cá thể UI cụ thể. Vì vậy, tác động của việc thêm các lớp cơ sở UI mới được giảm thiểu.

Một kiểm tra cẩn thận của Hình 12-6 cho thấy một trong những vấn đề với sự phù hợp của ISP không rõ ràng từ ví dụ TimedDoor. Lưu ý rằng mỗi giao dịch phải bằng cách nào đó biết về phiên bản UI cụ thể của nó. DepositTransaction phải biết về DepositUI, WithdrawTransaction phải biết về WithdrawalUI, v.v. Trong Liệt kê 12-6, tôi đã giải quyết vấn đề này bằng cách buộc mỗi giao dịch được xây dựng với tham chiếu đến UI cụ thể của nó. Lưu ý rằng điều này cho phép tôi sử dụng thành ngữ trong Liệt kê 12-7.

Điều này rất tiện lợi nhưng cũng buộc mỗi giao dịch phải chứa một thành viên tham chiếu đến giao diện người dùng của nó. Trong C #, người ta có thể muốn đưa tất cả các thành phần UI vào một lớp duy nhất. Liệt kê 12-8 cho thấy một cách tiếp cận như vậy. Điều này, tuy nhiên, có một tác dụng đáng tiếc. Lớp UIGlobals phụ thuộc vào DepositUI, WithdrawalUI và TRansferUI. Điều này có nghĩa là một mô-đun muốn sử dụng bất kỳ giao diện UI nào liên tục phụ thuộc vào tất cả chúng, chính xác là tình huống mà ISP cảnh báo chúng ta nên tránh. Nếu một thay đổi được thực hiện đối với bất kỳ giao diện UI nào, tất cả các mô-đun sử dụng UIGlobals có thể bị buộc phải biên dịch lại. Lớp UIGlobals đã kết hợp lại các giao diện mà chúng tôi đã làm việc rất chăm chỉ để phân tách!







Bây giờ hãy xem xét một hàm g cần truy cập vào cả DepositUI và transferUI. Cũng xem xét rằng chúng tôi muốn chuyển giao diện người dùng vào chức năng này. Chúng ta có nên viết khai báo hàm như thế này không:



Hay



Sự cám dỗ để viết các hình thức sau (đơn nguyên) là mạnh mẽ. Rốt cuộc, chúng ta biết rằng ở dạng pastb (polyadic), cả hai đối số sẽ đề cập đến cùng một đối tượng. Hơn nữa, nếu chúng ta sử dụng hình thức đa âm, cách gọi của nó có thể giống như sau:



Bằng cách nào đó điều này có vẻ đồi trụy. Nghịch ngợm hay không, hình thức đa thê thường thích hợp hơn hình thức đơn nguyên. Dạng đơn âm buộc g phải phụ thuộc vào mọi giao diện có trong UI. Do đó, khi WithdrawalUI thay đổi, g và tất cả khách hàng của g có thể bị ảnh hưởng. Điều này là sai lầm hơn g (ui, ui)! Hơn nữa, chúng ta không thể chắc chắn rằng cả hai đối số của g sẽ luôn luôn đề cập đến cùng một đối tượng! Trong tương lai, có thể các đối tượng giao diện được tách ra vì một số lý do. Thực tế là tất cả các giao diện được kết hợp thành một đối tượng là thông tin mà g không cần biết. Vì vậy, tôi thích hình thức đa âm cho các chức năng như vậy.

Khách hàng thường có thể được nhóm lại với nhau bằng các phương thức dịch vụ mà họ gọi. Các nhóm như vậy cho phép các giao diện tách biệt được tạo cho mỗi nhóm thay vì cho từng khách hàng. Điều này làm giảm đáng kể số lượng giao diện mà dịch vụ phải nhận ra và ngăn dịch vụ phụ thuộc vào từng loại máy khách.

Đôi khi, các phương thức được gọi bởi các nhóm khách hàng khác nhau sẽ chồng chéo lên nhau. Nếu sự trùng lặp là nhỏ, các giao diện cho các nhóm sẽ được tách biệt. Các hàm chung phải được khai báo

các giao diện chồng chéo. Lớp máy chủ sẽ kế thừa các chức năng chung từ mỗi giao diện đó nhưng sẽ chỉ thực hiện chúng một lần.

Khi các ứng dụng hướng đối tượng được duy trì, các giao diện cho các lớp và các thành phần hiện có thường thay đổi. Đôi khi, những thay đổi này có tác động rất lớn và buộc phải biên dịch lại và triển khai lại một phần rất lớn của hệ thống. Tác động này có thể được giảm thiểu bằng cách thêm giao diện mới vào các đối tượng hiện có thay vì thay đổi giao diện hiện có. Nếu khách hàng của giao diện cũ muốn truy cập m