The fun and easy way™ to decode design patterns!

Design Patterns

FOR

DUMMIES®

Solve your programming problems with patterns

A Reference for the Rest of Us!®

FREE eTips at dummies.com®

Steve Holzner, PhD
Author of Ajax For Dummies

Design Patterns for Dummies – Giới thiệu

Xin chào các thành viên trong diễn đàn congdongcviet !

Mình yêu máy tính, và lập trình từ nhỏ. Thông thường khi "bí" một vấn đề nào, mình lên mạng tìm kiếm. Hầu hết là tìm ra được lời giải nhờ các bạn khắp nơi từng hỏi và từng trả lời. Thiết nghĩ mình cũng nên có một chút đóng góp gì đó ngược lại cho cộng đồng lập trình viên, dù ít dù nhiều. Qua quá trình tự học và tìm tòi, cứ khi nghe tới một ngôn ngữ mới nào là mình tìm sách đọc và thực hành, nay cũng đã trải qua một số ngôn ngữ và IDE sau: đầu tiên là lập trình file .bat(trong MS-Dos), rồi Foxpro, Pascal,C,C++,Visual Basic 3.0, Java, Visual Basic 6, Visual C++6.0 MFC, Visual J++ 6.0, Visual Basic.Net, HMTL,Asp, Javascript, PHP, và gần đây nhất là C# và ASP.net. Mình nhận thấy tất cả ngôn ngữ chỉ là công cụ, và mục đích của chúng là tạo ra sản phẩm. Công cụ nào cũng có điểm mạnh, điểm yếu, nhưng nếu bạn sử dụng không đúng cách thì không thể tạo ra một sản phẩm tốt.

Vậy làm thế nào để sử dụng công cụ một cách đúng cách. Đó chính là tư duy giải thuật lập trình. Tư duy về giải thuật là cách chúng ta trừu tượng bài toán thành từng bước nhỏ, từng bước, và lắp ráp chúng thành một sản phẩm đúng.

Ví dụ để xây dựng một căn nhà, các bạn cần chuẩn bị gạch, cát, xi măng, nước, gỗ, ngói lợp… Khi có đủ nguyên vật liệu, các bạn bắt tay thực hiện từng bước như xây dựng móng nhà, xây dựng các bức tường, lợp ngói…

Trước đây mọi người lập trình theo phương pháp thủ tục, chia một vấn đề lớn thành nhiều phần nhỏ khác nhau và xử lý từng phần một theo đúng trình tự. Đây là một phương pháp tốt và từng được áp dụng trong một thời gian dài. Tuy nhiên khi chương trình ngày một càng lớn, việc duy trì và phát triển hàng trăm ngàn thủ tục là một công việc khó khăn, tốn chi phí và dễ sai sót. Khi đó phương pháp lập trình hướng đối tượng ra đời. Phương pháp này cũng chia một bài toán lớn thành các phần nhỏ, nhưng các phần nhỏ này được đóng gói vào từng đối tượng, các đối tượng này gần gũi với thực tế hơn rất nhiều, nên việc phát triển phần mềm ngày càng dễ tiếp cận với mọi người.

Ví dụ để lắp ráp một cái ti vi, chúng ta có đối tượng màn hình, đối tượng mainboard, đối tượng loa, đối tượng remote… sau đó lắp lại với nhau. Yahooo! Thật dễ dàng đúng không các bạn.

Trong topic này mình không có ý định giới thiệu về lập trình hướng đối tượng, vì sách vở về chủ đề này có quá nhiều, từ tiếng anh qua tiếng việt, và ít nhiều các bạn vào diễn đàn này đều đã nắm qua.
Vậy chủ đề thật sự của topic này là gì?

Có ai trong các bạn từng đặt câu hỏi ”tôi có khả năng xây 1 căn nhà, căn nhà 1 tầng, 2 tầng thậm chí 3 tầng. Nhưng không biết xây căn nhà 100 tầng thì sao?” Đối với căn nhà 3 tầng, khi có sai sót, hay thay đổi ở tầng 2, bạn có thể sửa, hoặc thậm chí đập bỏ và làm lại, nhưng đối với căn nhà 100 tầng, bị sai sót ở tầng thứ 3, chẳng lẽ bạn đập bỏ cả 97 tầng còn lại?

Và đó là nguyên nhân ra đời của một thứ gọi là Design Patterns – Các Mẫu Thiết Kế. Các bạn chắc đã từng nghe ai đó nói tới mẫu thiết kế, đúng vậy, tôi nghĩ bạn đã nghe nói đến nó trong lĩnh vực xây dựng kiến trúc. Và giờ đây, chúng ta sẽ nói tới nó trong lĩnh vực xây dựng phần mềm. Vậy mẫu thiết kế là gì: Nói nôm na nó là những giải pháp để giải quyết những vấn đề thường gặp trong phát triển phần mềm theo hướng đối tượng. Nó là những bản mẫu, ta dựa vào đó xây dựng nên những đối tượng. Vậy lợi ích của nó thế nào? Đó là nó giúp ta có những đối tượng mềm dẻo, dễ thay đổi, dễ bảo trì. Và vì sao mẫu thiết kế lại dính dáng tới lập trình hướng đối tượng. Thật ra thì nó bổ sung cho lập trình hướng đối tượng, nó mở rộng khả năng to lớn cho lập trình hướng đối tượng.

Gần đây khi mình có dịp đọc tới Design Patterns, mình thật sự thấy nó cuốn hút, rất có ích cho dân lập trình. Tuy nhiên tài liệu về design patterns cũng không nhiều(mình biết khoảng hơn chục cuốn), ngôn ngữ chủ yếu là tiếng anh, tài liệu tiếng việt thì ít, chắc chỉ có 1 cuốn (do MKPUB phát hành, mình chưa được đọc, vì mình ở tỉnh, không ở Tp.HCM nên không mua được). Cách tiếp cận vấn đề của mỗi sách cũng rất khác nhau, có khi rất chuyên ngành, và trừu tượng, nhiều khi gây khó khăn cho cả người bản xứ chứ không dám nói tới người Việt mình.

Với tham vọng đóng góp cho cộng đồng, mình sẽ dịch thuật một cuốn sách mình cho là tương đối dễ tiếp cận.
Sau khi đọc qua một số cuốn sách như sau:

- Design Patterns: Elements of Reusable Object-Oriented Software (Do bộ tứ tác giả Erich Gamma, Richard Helm, Ralph Johnson, John M.Vlissides). Cuốn kinh điển và đầy đủ nhất
- C# 3.0 Design Patterns: ByJudith Bishop. (Nhà xuất bản Oreilly)

- Head First Design Patterns (Nhà xuất bản Oreilly)
- Design Patterns: by Christopher G. Lasater (Nhà xuất bản Wordware)
- C# Design Patterns: A Tutorial by James W.Cooper (Nhà xuất bản Addison-Wesley)
- Design Patterns for Dummies: by Steve Holzner,PhD. (Nhà xuất bản Wiley)

Mình quyết định chọn cuốn Design Patterns for Dummies, tác giả Steve Holzner,PhD để giới thiệu đến các bạn. Có các nguyên nhân sau:

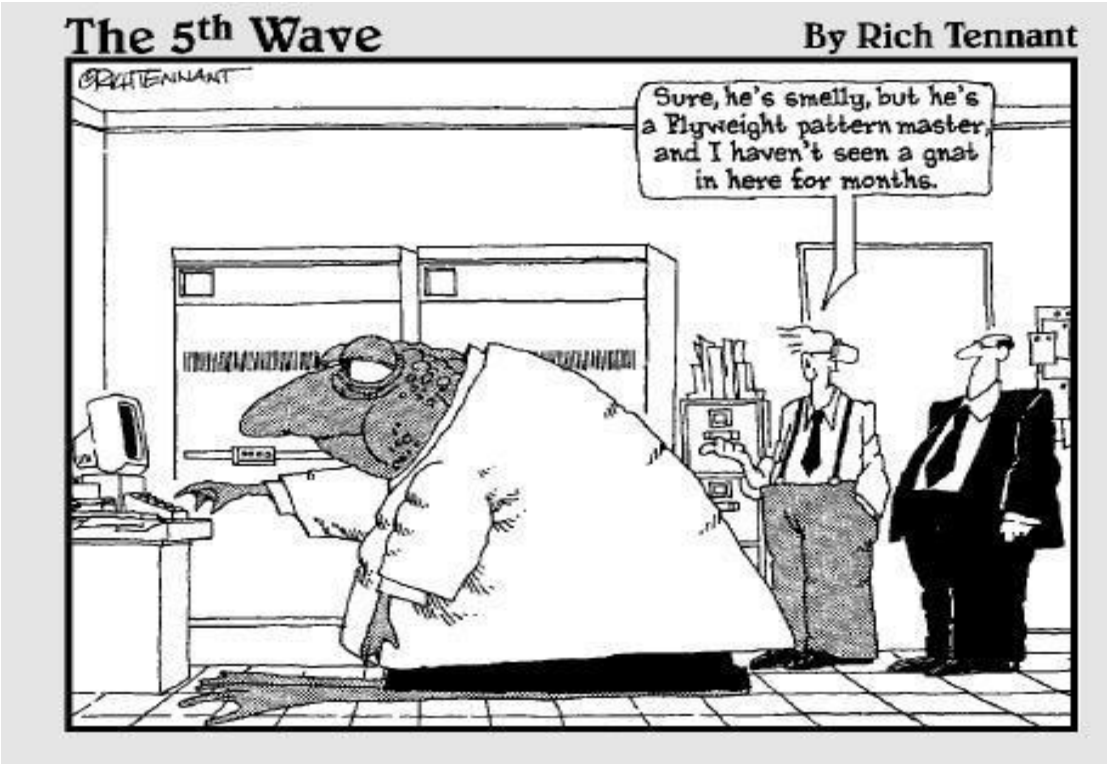
- Cách dẫn dắt dễ hiểu, ví dụ sinh động, ngôn ngữ thân thiện
- Không sử dụng UML để mô tả biểu đồ, (sẽ gây khó khăn cho các bạn chưa nắm vững UML)
- Số mẫu tương đối đầy đủ.
- Số trang sách không nhiều lắm. Chỉ khoảng 300trang.

Tuy nhiên các ví dụ trong sách là được viết từ Java. Có thể sẽ gây ra lúng túng một chút cho các bạn sử dụng C#. Nhưng các bạn cũng biết cha đẻ của C# chính là cha đẻ của Borland C, Borland C++, Delphi, Visual J++, nên các bạn sẽ thấy Java và C# không khác biệt lắm.

Đối với các bạn chưa thể tự mình chuyển đổi mã nguồn từ Java ->C# theo các ví dụ trong sách, mình cũng viết lại các ví dụ theo C#. Tuy không thể hiện 100% việc chuyển đổi, nhưng vẫn nổi bật được ý đồ của tác giả.

Vì trình độ anh ngữ của mình chưa tốt (chỉ tự học). Nên không đảm bảo dịch sát 100% ý đồ tác giả. Mặt khác việc dịch thuật, và chạy chương trình cũng tốn nhiều thời gian. Mỗi tuần mình sẽ đăng một chương của cuốn sách. Cuốn sách 12 chương. Tổng thời gian trong 3 tháng, các bạn sẽ được làm quen với hầu hết các mẫu thiết kế. Có thể thời gian trên so với mấy cuốn sách của SAM: “Học trong 24giờ”, sẽ làm các bạn thấy nản. Nhưng lập trình là một nghệ thuật, mà học cách để làm chủ nghệ thuật thì 3 tháng chắc chắn còn chưa đủ. Mong các bạn hãy kiên nhẫn.

DP4Dummies – Chương 1 – Tổng quan các mẫu DP



Chương 1: Xin chúc mừng, rắc rối của bạn đã được giải quyết triệt để.

Trong chương này, chúng ta sẽ nói đến:

- Giới thiệu về Mẫu thiết kế Design Patterns là gì?
- Hiểu biết về tác dụng của Design Patterns
- Mở rộng lập trình hướng đối tượng
- Điểm sơ qua một số mẫu Design Pattern

Là một lập trình viên, bạn biết rằng thật khó khăn để nhớ chi tiết những việc bạn đang thực hiện. Và khi bạn không nắm bắt được tổng quát công việc, bạn có thể dễ dàng bỏ lỡ những việc quan trọng. Khi đó, mã nguồn bạn đang viết có thể vẫn còn làm việc tốt đẹp, nhưng trừ khi bạn bao quát được bức tranh lớn hơn, lúc đó mã nguồn bạn viết mới thực sự hoàn hảo.

Những vấn đề nghiêm trọng thực sự thường xuất hiện sau khi bạn đã chỉnh sửa chúng ít nhất một lần. Những nhà phát triển thường tự mình xử lý bằng cách viết lại mã nguồn

và sửa các lỗi. Tuy nhiên trong môi trường công việc, những nhà lập trình thường bỏ phần lớn thời gian để bảo trì, chỉnh sửa những công việc cũ hơn là tập trung vào những sản phẩm mới.

Bạn thấy rằng thật vô lý khi cứ phải làm, rồi sửa, làm lại, sửa tiếp… Giải pháp hợp lý nhất là bạn đưa ra được một quy trình tổng quan cho việc thiết kế và bảo trì, có như vậy, bạn mới tránh được các rắc rối phát sinh khi môi trường ứng dụng thay đổi, hoặc ít nhất bạn cũng giúp cho việc bảo trì, chỉnh sữa dễ dàng hơn khi có phát sinh.

Ý tưởng đằng sau cuốn sách này là: Bạn sẽ sử dụng một tập hợp các mẫu thiết kế Design Patterns để làm đơn giản hóa quá trình trên. Kế hoạch này sẽ giúp bạn có một cái nhìn tổng quát. Một mẫu thiết kế Design Pattern là một giải pháp đã được kiểm nghiệm thành công khi đối diện một vấn đề lập trình phát sinh cụ thể. Khi bạn quen thuộc hết các mẫu thiết kế trong sách này, bạn nhìn vào một chương trình. Bam! Một giải pháp đúng đắn sẽ xuất hiện trong tâm trí bạn, thay vì bạn phải đập đầu vào tường trong vô vọng, giờ bạn có thể ung dung nói "Ở đây, tôi sẽ sử dụng mẫu Factory, mẫu Observer, hay mẫu Adapter…"

Đó là chưa nói, một số sách về thiết kế khuyên bạn nên dành phần lớn thời gian để phân tích và lên kế hoạch cho một đề án. Vẻ đẹp thật sự ở đây là một người nào đó đã đối mặt với vấn đề bạn đang gặp phải, họ đã có giải pháp đúng đắn cho nó. Và giờ khi bạn nhuần nhuyễn mẫu thiết kế, bạn có thể áp dụng các thiết kế đó một cách dễ dàng.

Làm sao để trở thành chuyên gia thiết kế trong lĩnh vực phần mềm, điều mà ai cũng thèm muốn? Thật dễ dàng, hãy đọc cuốn sách này, nghiền ngẫm những mẫu thiết kế mà tôi dành nhiều tâm huyết để viết. Bạn không cần phải nhớ mọi thứ, bạn chỉ cần biết là có những mẫu thiết kế đó. Và khi bạn đối diện với một vấn đề thực tế, sâu thẳm trong bạn tự nhiên thốt lên "À, có vẻ chỗ này có thể dùng mẫu Iterator…" Sau đó bạn chỉ cần tìm kiếm mẫu thiết kế đó trong cuốn sách này, duyệt qua các ví dụ để biết phải làm gì. Và vì vậy, chương này sẽ là một tour du lịch nho nhỏ, giúp bạn đi qua một số mẫu thiết kế tiện dụng và hữu ích.

Chỉ cần tìm ra mẫu thiết kế thích hợp

Điểm kỳ diệu của Design Patterns là nó giúp cho công việc của bạn dễ dàng tái sử dụng, dễ mở rộng và bảo trì. Khi bạn thiết kế không tốt, phần mềm của bạn không có khả

năng tái sử dụng và bảo trì, khi gặp vấn đề phát sinh, bạn sẽ phải dành nhiều thời gian, có khi là nhiều hơn cả lúc bạn viết ban đầu, chỉ là để sửa chữa chúng.

Ví dụ: Bạn đang muốn tạo một đối tượng Java, nhiệm vụ là đọc và phân tích một tài liệu XML. Bạn cần phải tạo một lớp Parser (chuyên dùng để phân tích XML) sau đó bạn tạo một đối tượng của lớp này. Bạn thầm nghĩ "Tới giờ mọi việc vẫn ổn". Nhưng thực tế thì đã có hàng tá lớp Parser do người khác viết, và họ luôn muốn sử dụng lại những tính năng đặc biệt trong lớp của họ. Nếu bạn có thể sử dụng mẫu thiết kế Nhà máy Factory, giờ đây bạn có thể sử dụng bất cứ lớp Parser nào, kể cả của những người khác, thay vì cứ khư khư xài lớp Parser do chính bạn viết ra. Và vì vậy, chương trình của bạn đã trở nên dễ mở rộng, tái sử dụng được và bảo trì dễ dàng.

Nói cách khác, Mẫu thiết kế Design Patterns là những giải pháp giúp cho ta có một thiết kế tốt khi đối diện những vấn đề phát sinh trong việc lập trình. Nhiều người đã gặp vấn đề này, và đã giải quyết tốt, việc của bạn là áp dụng chúng. Bạn không cần phải ghi nhớ mọi thứ, chỉ cần nhìn ra đâu là mẫu thiết kế phù hợp và đặt nó vào đúng chỗ.

Thật tuyệt đúng không các bạn.

Đôi nét về cuốn sách tên "Gang of Four" Bộ tứ tác giả.

Quyển sách là tập hợp 23 mẫu thiết kế được phát hành bởi Erich Gamma, Richard Helm, Ralph Johnson và John Vlissides, trong một nghiên cứu của họ năm 1995. Với tựa đề gốc là "Design Patterns: Elements of Reusable Object-Oriented Software". Tạm dịch "Mẫu thiết kế: những thành phần tái sử dụng trong lập trình hướng đối tượng". Họ được giới lập trình gọi là Bộ tứ Gang of Four, hay GoF. (ND: Bộ tứ ở đây là ẩn dụ với Tên nhóm nhạc nổi tiếng Gang Of Four của Anh hay Bộ tứ quyền lực Mafia trong tác phẩm Bố già hay là bộ tứ quyền lực chính trị của Trung quốc, ở VN cũng có bộ tứ quyền lực của Vietnam Next Top Model…).

Đã có rất nhiều sự thay đổi kể từ khi xuất hiện, một số trong 23 mẫu được sử dụng nhiều, số khác ít khi được sử dụng. Tôi sẽ nói đến đầy đủ 23 mẫu trong cuốn sách này, nhưng tôi sẽ nhấn mạnh ở những mẫu thường được sử dụng hơn. Và kể cả mẫu mới không có trong sách của GoF, trong chương 11.

Có một sự thật là, bạn không chỉ phải nhớ kỹ từng mẫu thiết kế, bạn phải hiểu sâu sắc về nó, để có thể áp dụng đúng đắn trong thực tiễn. Tôi cũng sẽ lưu ý nhiều về lập trình hướng đối tượng xuyên suốt quyển sách này. Lập trình hướng đối tượng OOP là một

bước tiến tuyệt vời trong lĩnh vực lập trình. Nhưng có quá nhiều lập trình viên sử dụng chúng một cách tùy tiện, thiếu chiều sâu, và điều đó gây ra nhiều rắc rối tiềm ẩn.

Phần lớn việc tìm hiểu những mẫu thiết kế chính là việc mở rộng khái niệm lập trình hướng đối tượng. Ví dụ: đóng gói những gì thay đổi nhiều nhất (encapsulating what changes most), cách chuyển đổi một quan hệ kế thừa is-a sang quan hệ kết hợp has-a (xem chương 2 để biết chi tiết) – và tôi sẽ nói chi tiết về chúng.

Hãy bắt đầu bằng mẫu Mediator Pattern (Người trung gian)

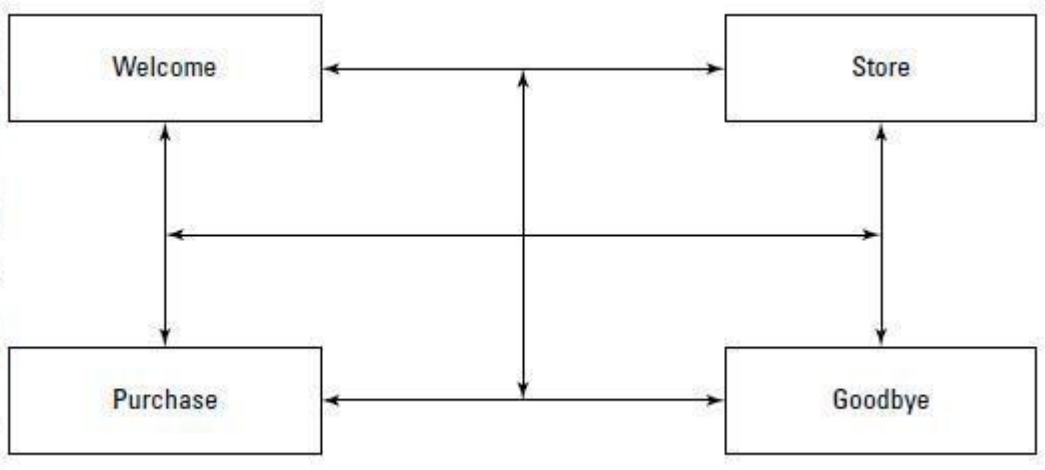


Figure 1-1: In this illustration of the basic navigation of a typical online store, the Mediator pattern mediates between objects.

Hình bên trên là một ví dụ về mẫu thiết kế, mẫu Mediator. Hình cho chúng ta thấy chức năng của một mẫu Mediator. Theo hình ta đang có một website với 4 trang. Website cho phép khách hàng duyệt qua kho hàng và đặt mua. Khách hàng có thể đi từ trang này qua trang khác theo đường vẽ trên hình. Ở đây có một vấn đề phát sinh. Tại từng trang, bạn phải viết mã để nhận biết khi nào khách hàng muốn nhảy qua trang khác và kích hoạt trang đó. Tại một trang bạn có quá nhiều đường để đi tới trang khác, và vì vậy sẽ phát sinh nhiều đoạn code trùng lặp trên nhiều trang khác nhau.

Bạn có thể sử dụng mẫu Mediator để đóng gói tất cả các đường dẫn tới trang vào một module duy nhất, và đặt nó vào trong một đối tượng Mediator. Từ bây giờ, từng trang chỉ cần phải thông báo bất cứ sự thay đổi nào cho Mediator, và Mediator tự biết dẫn trang cần thiết cho khách hàng, như trong hình bên dưới.

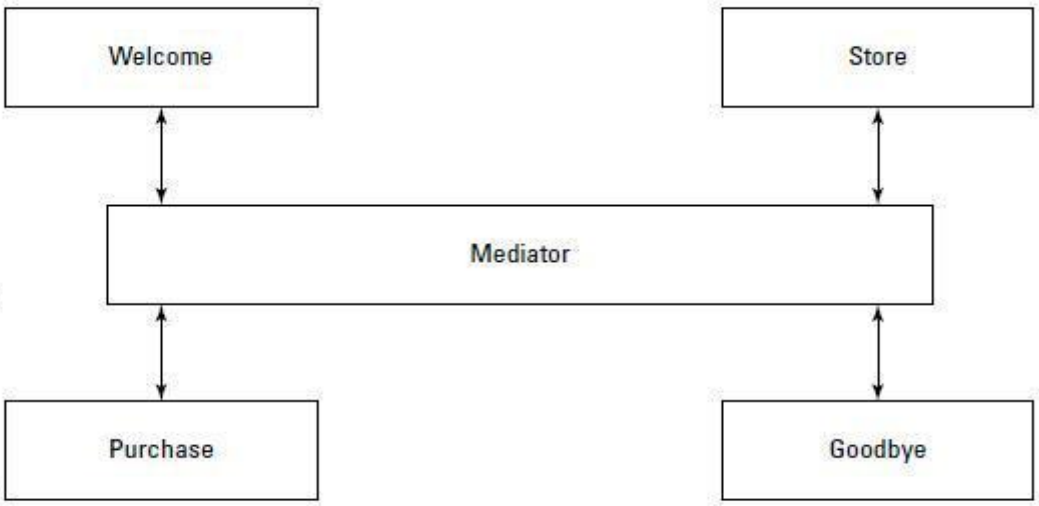


Figure 1-2: The mediator directs Web site traffic.

Bạn có thể tạo ra một Mediator với chức năng dẫn trang. Tại đây bạn có thể chỉnh sửa và thay đổi dễ dàng. Đó chính là chức năng của Mediator (Người trung gian)

Chuyển đổi với mẫu thiết kế Adaptor (Người chuyển đổi)

Đây là một mẫu khác, mẫu chuyển đổi Adaptor.

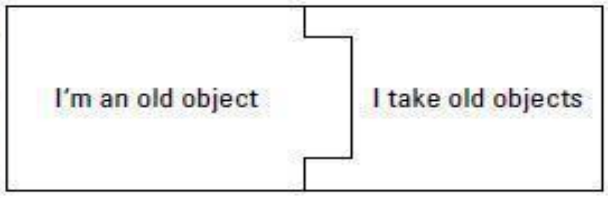


Figure 1-3: Everything seems to be working here.

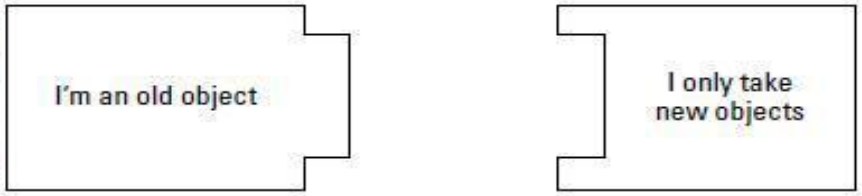


Figure 1-4: This isn't going to work.

Figure 1-5: Old objects work with new objects via an adapter.

Hãy nhìn vào hình 1-3. Đầu vào là một đối tượng cũ. Hệ thống tiếp nhận đối tượng cũ. Với hình 1-4. Khi hệ thống thay đổi, hệ thống không tiếp nhận đối tượng cũ nữa, chỉ tiếp nhận đối tượng mới.

(ND: thực ra hình ảnh 1-4 có chút chưa chính xác, phần "I only take new objects, nên vẽ nhỏ lại)
Hình 1- 5. Đây là nơi xuất hiện mẫu Adaptor (Người chuyển đổi). Mục đích là chuyển đổi đối tượng cũ, thành đối tượng mới, khi đó hệ thống sẽ sẵn sàng tiếp nhận đối tượng này.

"ND: Các bạn hãy tưởng tượng. Các bạn có một chiếc tivi với đầu cắm điện 3 chân. Ổ cắm điện ở nhà bạn là loại ổ 2 chân. Bạn ra ngoài cửa tiệm, mua 1 cục chuyển đổi, từ 3 chân ra 2 chân. Lúc đó bạn có thể sử dụng được ổ điện 2 chân rồi. Cục chuyển đổi từ 3 chân ra 2 chân, chính là Adaptor"

Vấn đề đã được giải quyết. Ai nói học các mẫu thiết kế là khó khăn nhỉ.

Đứng trong một đối tượng. Mẫu Proxy. (Người đại diện)

Đây là một mẫu khác. Mẫu Proxy. Mẫu này nói rằng, mã bạn viết chỉ tương tác với đối tượng cục bộ như hình dưới

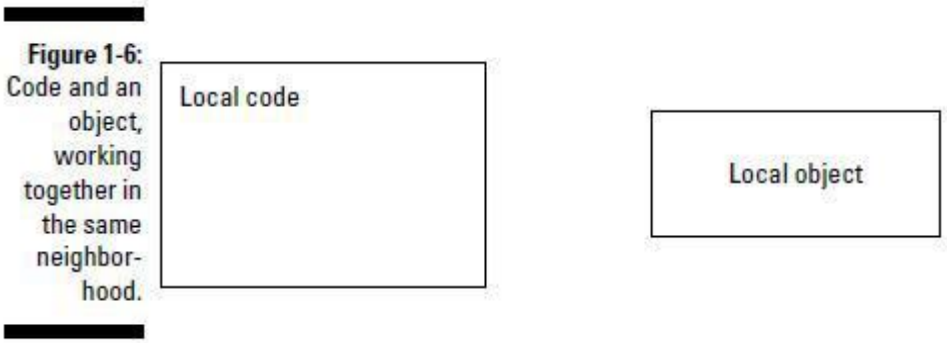


Figure 1-6: Code and an object, working together in the same neighbor-hood.

Thực tế phát sinh, bạn buộc phải tương tác với một đối tượng ở xa, đâu đó trên thế giới. Làm sao bạn có thể làm cho chương trình của bạn tương tác với một đối tượng cục bộ trong khi thực tế là nó đang làm việc với một đối tượng ở xa.

Ở đây mẫu Proxy (Người đại diện) xuất hiện. Nó là một đối tượng nằm bên trong chương trình, làm trách nhiệm tương tác với chương trình, giúp cho chương trình tưởng

rằng nó đang tương tác cục bộ, thay vì tương tác với một đối tượng từ xa. Bên trong, Proxy chịu trách nhiệm kết nối với đối tượng từ xa. Như hình dưới

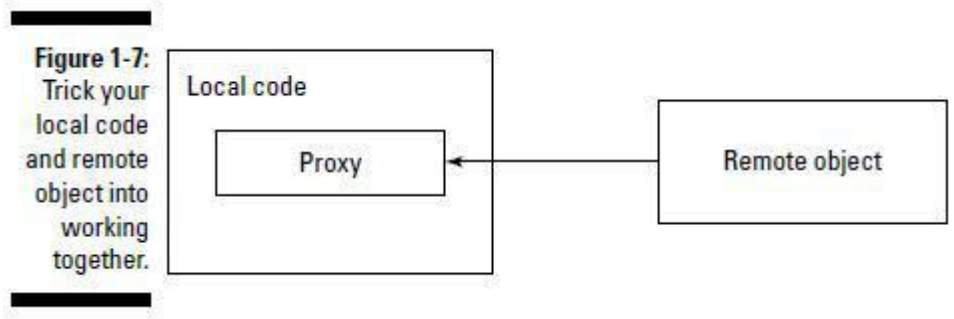


Figure 1-7: Trick your local code and remote object into working together.

Bạn sẽ biết cách mẫu Proxy hoạt động trong chương 9.

Đôi nét về mẫu Observer (Người quan sát)

Bạn có thể quen thuộc với một vài mẫu trong sách này, ví dụ như mẫu Observer này chẳng hạn.

Mẫu Observer có thể đăng ký với hệ thống. Khi hệ thống có sự thay đổi, hệ thống sẽ thông báo cho Observer biết. Khi không nữa cần, mẫu Observer sẽ được gỡ khỏi hệ thống.

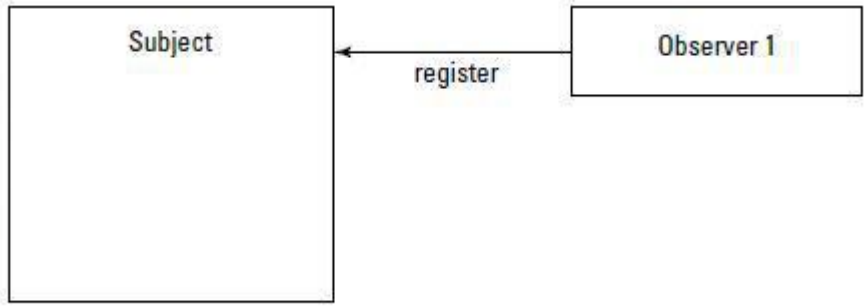


Figure 1-8: The Observer pattern lets observers register with subjects.

And another observer, Observer 2, can register itself as well, as shown in Figure 1-9.

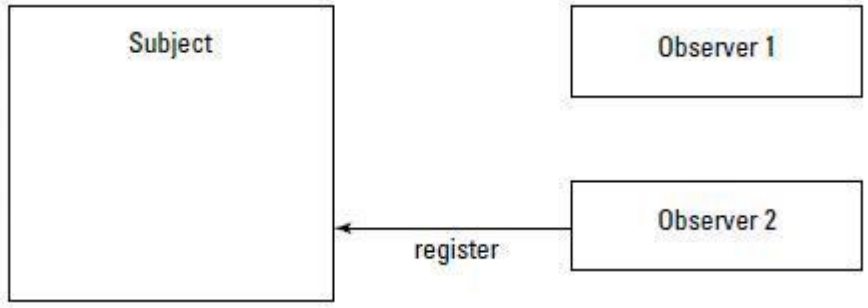


Figure 1-9: More than one observer can register with a subject.

Now the subject is keeping track of two observers. When an event occurs, the subject notifies both observers. (See Figure 1-10.)

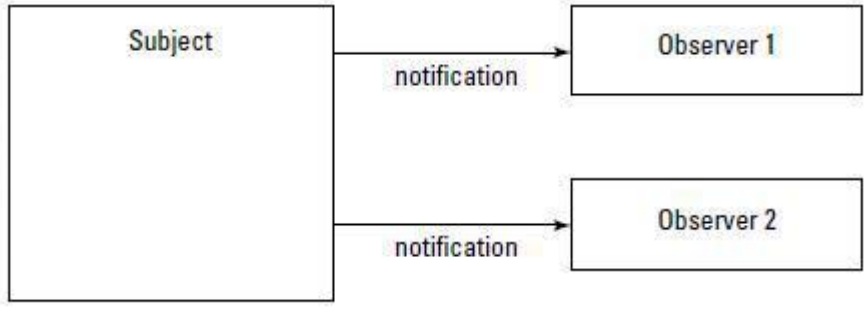


Figure 1-10: When events occur in the subject, registered observers are notified.

Hình 8 cho thấy mẫu Observer cho phép 1 observer đăng ký với hệ thống. Hình 9, cho phép observer thứ 2 đăng ký với hệ thống. Hiện tại hệ thống đang liên lạc với 2 observer. Khi hệ thống phát sinh một sự kiện cụ thể nào đó, nó sẽ thông báo với cả 2 observer như hình số 10.

Tôi cố gắng trình bày tất cả các Mẫu thiết kế theo cách dễ hiểu, dễ tiếp cận nhất. Bạn sẽ không phải nhìn vào đống biểu đồ, cùng với các lớp trừu tượng đầy phức tạp nữa. Các chương trong sách nhắm vào các độc giả là lập trình viên, rất hữu ích cho các bạn, cho dù các bạn không đọc hết tất cả các mẫu. Các mẫu thiết kế trong sách này đã trở thành các tiêu chuẩn về lập trình trên thế giới, và chúng hữu dụng cho các bạn, dù bạn đang ở trình độ nào. Hy vọng rằng, trong tương lai, khi bạn đối diện với chương trình của mình, bạn đột nhiên nhận ra: Aha, đây chính là mẫu Façade.

DP4Dummies – Chương 2: Strategy

Chương 2: Lên kế hoạch hành động với Mẫu Strategy (Mẫu chiến lược)

Trong chương này, chúng ta sẽ đi qua các nội dung sau:

- Mở rộng việc lập trình hướng đối tượng
- Làm quen với các khái niệm trừu tượng, đóng gói, đa hình và kế thừa
- Chuyển đổi qua lại giữa 2 khái niệm "is-a" và "has-a"
- Xử lý công việc bằng các thuận toán
- Áp dụng mẫu Strategy vào thực tế

Là một chuyên gia thiết kế mẫu, bạn đi vào phòng họp của công ty MegaGigaCo, giám đốc điều hành và các thành viên ban quản trị đang ăn mừng một hợp đồng mới về thiết kế xe hơi, mọi người vỗ tay và hò reo ăn mừng quanh phòng.

"Hợp đồng này sẽ đem đến doanh số lớn cho chúng ta", giám điều hành nói, cùng với tiếng vang bốp bốp của rượu champagne và sự phấn khích của giám đốc. "Việc của chúng ta là phải chắc chắn có được một quy trình thiết kế đúng". Ông nhấn nút lên chiếc đèn chiếu và hình ảnh các biểu đồ hiện lên tường. Ông nói tiếp: "Đây là ý kiến của tôi…"

"Sai", bạn nói

Giám đốc thoáng một chút giật mình và nói tiếp, "Nhưng nếu chúng ta…"

"Không," bạn lắc đầu nói.

“Xin thứ lỗi”, bạn nói với Giám đốc và ban điều hành, “Rõ ràng là chúng ta đang mạo hiểm với hợp đồng này vì đã đi sai hướng. Tôi có thể thấy cả tá vấn đề khi nhìn vào các biểu đồ này”

Ban giám đốc thì thầm với vẻ tập trung và Giám đốc hỏi. “Theo ý kiến anh thì sao…”

“Tôi là chuyên gia thiết kế mẫu, người sẽ giải quyết các vấn đề về liên quan về thiết kế,” Bạn nói. “Dĩ nhiên là cho những hợp đồng lớn”

Giám đốc viết ra một con số dự đoán cho chi phí, một con số khá lớn, tuy nhiên hình như vẫn chưa đủ lớn đối với bạn.

“Lại sai”, bạn nói

Vị giám đốc nhìn bạn nhíu mày.

“Mẫu thiết kế”, bạn giải thích. “Các giải pháp chung để giải quyết cho các vấn đề lập trình thường gặp. Không chỉ vậy, nó còn giúp việc lập trình tốt hơn, bảo dưỡng, và công việc nâng cấp dễ dàng hơn . Ông thấy đó, việc thuê một chuyên gia như tôi có nhiều ý nghĩa, khi tôi thấy một vấn đề trong việc lập trình mà có thể giải quyết theo một mẫu thiết kế nào đó, tôi có thể nói chi tiết về nó cho ông biết.

“Tốt”, các lập trình viên trong công ty nói một cách miễn cưỡng, “ý kiến của anh về mẫu thiết kế nghe cũng hay đấy. Nhưng chúng ta đã sử dụng phương pháp lập trình hướng đối tượng, điều đó chưa giải quyết được vấn đề à?”

“Không” bạn nói. Thực tế thì nội dung chính của mẫu thiết kế là chúng mở rộng khái niệm lập trình hướng đối tượng.

MỞ RỘNG KHÁI NIỆM LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG?

Chúng ta nhắc lại cuốn sách của Gang Of Four (GOF: Bộ tứ tác giả), tựa đề “Mẫu thiết kế: Sử dụng lại các thành phần trong việc lập trình hướng đối tượng” do nhà xuất bản Addison Wesley, 1995 xuất bản. “Sử dụng lại” là một khía cạnh quan trọng khi làm việc với các mẫu thiết kế, và vì thế nó cũng giải quyết được các vấn đề của lập trình hướng đối tượng(OOP). Tôi sẽ thảo luận trước về OOP trong chương này, và sau đó sẽ cho bạn thấy mối tương quan giữa OOP và mẫu “Strategy” hay còn gọi là mẫu chiến lược

OOP ban đầu được phát triển như là một phương pháp lập trình cho các vấn đề lớn và phức tạp. Ý tưởng chính là đóng gói tất cả chức năng vào trong đối tượng. Nói cách khác, đây là phương pháp chia để trị. Trước khi OOP ra đời, bạn lập trình theo phương pháp thủ tục, bạn có thể chia các tính năng của chương trình thành các thủ tục khác nhau, nhưng điều đó càng ngày càng phức tạp khi kích cỡ chương trình lớn thêm. Khi đó chúng ta cần phải có một phương pháp mới để quản lý các thủ tục một cách dễ dàng, và đó là nguyên nhân ra đời của một phương pháp mới, phương pháp quản lý trên đối tượng.

Ví dụ, bạn hãy nhìn vào nhà bếp và cách hoạt động của nó, thật là cơ man các sự phức tạp. Tủ lạnh phải có các bơm làm mát, bộ phận cảm biến nhiệt, quạt và vân vân. Bếp lò có thể bao gồm nhiều thiết bị như bộ phận nhiệt độ, bộ định thời gian, đèn..Theo cách nhìn nhận này, khi ta xem xét nhà bếp với mọi bộ phận cùng một lúc, ta thấy nhà bếp quá phức tạp.

Nhưng nếu bạn bao bọc từng thành phần, thành các đối tượng, tình hình đã có thể dễ dàng xử lý hơn rất nhiều. Đây là cái tủ lạnh. Đây là cái bếp lò. Đó là cái máy rửa chén và vân vân. Không vấn đề gì lớn cả. Các chi tiết nhỏ làm việc cùng nhau được ta đóng gói thành một đối tượng.

Đó cũng chính là các đối tượng trong lập trình hướng đối tượng. Bạn gộp các chức năng vào trong một đối tượng và chúng dễ dàng được nhận biết, nào là cái tủ lạnh, bếp lò hay máy rửa chén…Và phương pháp lập trình dựa trên các đối tượng được gọi là lập trình hướng đối tượng (Tất nhiên bạn chẳng nghe ai nói tới lập trình hướng tủ lạnh, hay lập trình hướng bếp lò…)

Ví dụ, trong chương trình của bạn, bạn có một đối tượng tên là “Màn hình”, nó bao gồm các chức năng để hiển thị dữ liệu mà bạn mong muốn. Một đối tượng khác tên “Cơ sở dữ liệu” sẽ làm nhiệm vụ giao tiếp với máy chủ cơ sở dữ liệu và vân vân…Có thể có nhiều phức tạp bên trong từng đối tượng, nhưng khi bạn đóng gói mọi thứ vào đối tượng, mọi thứ đã trở nên dễ dàng hơn rất nhiều. Bạn có thể làm việc với khái niệm đối tượng “Màn hình” và một số chức năng đơn giản của nó như Thiết Lập Hệ Số Quét , Canh Chỉnh Màn Hình , Thiết Lập Bộ Đệm Video… và hàng tá chức năng khác. Nó làm cho lập trình trở nên dễ dàng hơn, và đó là lý do tại sao lập trình hướng đối tượng đã trở thành phương pháp mạnh mẽ và phổ biến hơn bao giờ hết.

BỐN KHÁI NIỆM CHÍNH CỦA OOP

OOP bao gồm bốn khái niệm chính là trừu tượng, đóng gói, đa hình và kế thừa. Tôi sẽ thảo luận chúng trong phần sau đây:

Trừu tượng là gì?

Khi bạn làm việc với mẫu thiết kế, bạn sẽ thấy phần lớn đều liên quan đến khái niệm trừu tượng. Trừu tượng là cách thức bạn nghĩ ra để xem xét việc giải quyết một vấn đề nào đó. Trừu tượng không phải là một kỹ thuật lập trình. Thực chất, nó chỉ có nghĩa là bạn phải nhận thức được vấn đề trước khi áp dụng kỹ thuật hướng đối tượng.

Trừu tượng là cách bạn phân chia, cách giải quyết vấn đề thành những phân đoạn nhỏ hơn. Đây là cách thức bạn giải quyết vấn đề bằng cách chia chúng ra thành từng phần nhỏ có thể quản lý được. Nói cách khác, trừu tượng hóa một bài toán, đơn giản chỉ là cách giải quyết bài toán theo kiểu hướng đối tượng. Các dữ liệu cần thiết cho từng đối tượng sẽ trở thành thuộc tính của đối tượng đó, thuộc tính đó có thể là riêng tư cho đối tượng hoặc công cộng cho đối tượng khác sử dụng. Các hành vi mà đối tượng thể hiện trong thế giới thực cũng trở thành một hành động của chúng và được viết thành mã của chương trình.

Khi bạn chắc chắn đã tìm ra hướng giải quyết một bài toán đúng đắn, bạn mới có thể áp dụng các mẫu thiết kế. Thông thường, khi làm việc với mẫu thiết kế bạn sẽ tốn nhiều thời gian hơn cho việc trừu tượng hóa các khái niệm, hơn là làm việc với từng đối tượng cụ thể.

Đóng gói là gì?

Khi bạn đưa tất cả chức năng và dữ liệu vào trong một đối tượng, bạn đã "đóng gói" chúng. Đây là sức mạnh thực sự của việc lập trình hướng đối tượng. Bạn đã gỡ bỏ sự phức tạp của đối tượng khi đóng gói tất cả dữ liệu vào trong đối tượng đó. "Đóng gói" là cách bạn đã đưa hàng tá đường dây điện, ống, cảm biến nhiệt, đèn… vào trong một đối tượng là tủ lạnh.

Khi bạn đóng gói chức năng vào trong một đối tượng, bạn quyết định cách thức mà đối tượng giao tiếp với thế giới bên ngoài. Một cái tủ lạnh có thể một quản lý hàng tá thứ phức tạp bên trong, tuy nhiên người sử dụng chỉ quan tâm là nó có thể làm lạnh thức ăn

hay không? Cùng cách thức đó, bạn cũng chỉ định đâu là chức năng, thuộc tính bên trong của tủ lạnh, đâu là chức năng thuộc tính nó giao tiếp với thế giới bên ngoài.

Có một ý tưởng đằng sau việc đóng gói – Bạn che giấu những thứ phức tạp bên trong đối tượng và tạo ra một giao diện đơn giản để đối tượng giao tiếp với phần mã còn lại của bạn.

Mẫu thiết kế cũng là một trường hợp đặc biệt của sự đóng gói. Bạn phải đóng gói những gì bạn cho là dễ thay đổi nhất Một số mẫu thiết kế xoay quanh ý tưởng là trích ra những phần mã dễ thay đổi nhất hoặc phần cần phải bảo trì nhiều và đóng gói chúng vào một đối tượng riêng để dàng dàng xử lý hơn. Xuyên suốt cuốn sách này, bạn sẽ nhìn thấy nhiều về sự đóng gói và cách thức bất ngờ mà mẫu thiết kế giải quyết các vấn đề thường gặp.

Đa hình là gì?

Một nền tảng khác của lập trình hướng đối tượng là "tính đa hình". Đó là khả năng khi chúng ta lập trình một chức năng, chức năng đó có thể làm việc với nhiều loại đối tượng khác nhau, tùy thuộc vào kiểu thực sự của đối tượng đó trong thực tế. Ví dụ, bạn có thế viết mã để xử lý tất cả các hình dạng khác nhau như hình tròn, hình chữ nhật, tam giác.. vân vân. Mặc dù chúng có hình dạng khác nhau, chúng có chung một số hành động, ví dụ như chúng có chung chức năng là ĐượcVẽRa.

Sử dụng tính đa hình, bạn có thể viết code để thực hiện nhiều hành động trên loại hình dáng mà bạn định làm việc và sau đó quyết định hình dạng thực tế nào sẽ được sử dụng khi chạy chương trình. Đa hình (nhiều hình thứ) có nghĩa là mã bạn viết ra có thể sử dụng được nhiều kiểu của đối tượng mà bạn không phải viết lại mã.

Sau đây là ví dụ. Bạn tạo ra một lớp (class) *Shape* với phương thức chung là *draw*

```
class Shape
{
  public void draw()
  {
    System.out.println("Drawing a shape.");
  }
}
```

Sau đó bạn có thể mở rộng một lớp mới, lớp *Rectangle*, từ lớp *Shape*, và cho phép nó vẽ một hình chữ nhật như sau:

```
class Rectangle extends Shape
{
  public void draw()
  {
    System.out.println("Drawing a rectangle.");
  }
}
```

Bạn muốn vẽ một hình? Không vấn đề gì. Bạn viết một ít mã để tạo một đối tượng tên shape và gọi phương thức draw:

```
public class Polymorphism
{

  public static void main(String[] args)
  {
    Shape shape = new Shape();

    shape.draw();
  }
}
```

Kết quả khi chạy chương trình:

```
Drawing a shape.
```

Muốn vẽ hình chữ nhật sử dụng cùng mã trên? Không vấn đề gì. Đây là sự kỳ diệu của tính "đa hình", hãy tạo lại biến shape với kiểu rectangle và chạy lại đoạn code trên:

```
public class Polymorphism
{

  public static void main(String[] args)
  {
    Shape shape = new Shape();
    shape = new Rectangle();

    shape.draw();
  }
}
```

Kết quả là:

```
Drawing a rectangle.
```

Trong trường hợp thứ nhất, bạn đã nạp một đối tượng shape vào biến shape và gọi phương thức draw. Trong trường hợp thứ hai, bạn đã lấy một đối tượng rectangle và nạp nó vào cùng biến shape (mặc dù là bạn đã khai báo nó như là 1 đối tượng shape) và sau đó gọi phương thức draw.

Vậy là bạn đã cùng sử dụng một biến shape, để lưu giữ một đối tượng shape, một đối tượng rectangle, chương trình vẫn hoạt động vì rectangle được thừa kế từ shape. Đây là cách bạn quyết định kiểu đối tượng nào được nạp vào biến shape khi chạy chương trình và khi đó toàn bộ mã của bạn vẫn không hề thay đổi.

Kế thừa là gì?

Đặc điểm cuối cùng và nổi bật của lập trình hướng đối tượng là tính kế thừa. Là qui trình mà một lớp có thể thừa hưởng toàn bộ phương thức và thuộc tính của một lớp khác. Bạn có thể nhìn thấy sự kế thừa trong ví dụ trước, bắt đầu từ lớp Shape:

```
class Shape
{
  public void draw()
  {
    System.out.println("Drawing a shape.");
  }
}
```

Sau đó lớp Rectangle kế thừa từ lớp Shape như sau:

```
class Rectangle extends Shape
{
  public void draw()
  {
    System.out.println("Drawing a rectangle.");
  }
}
```

Đa hình thường xuất hiện khi bạn làm việc với mẫu thiết kế bởi vì mẫu thiết kế có xu hướng ủng hộ “kết hợp” hơn là “kế thừa” (Bạn sử dụng “kết hợp” khi đối tượng của bạn

chứa đối tượng khác thay vì thừa hưởng từ chúng). Kế thừa chính là mối quan hệ "Is-a" (là một). Ta có thể nói Rectangle "is-a" Shape.

Mẫu thiết kế- lập trình hướng đối tượng thường sử dụng việc "kết hợp" hơn là "kế thừa". Khi bạn sử dụng "kết hợp", mã của bạn chứa đựng một đối tượng khác, hơn là thừa hưởng từ chúng. Phương pháp này tỏ ra mềm dẻo, uyển chuyển để thích ứng với nhiều loại đối tượng trong cùng một cách, cùng một đoạn mã. Mẫu thiết kế thường dựa trên tính đa hình.

"Kết hợp" hay "Đa hình":

Thử nghiệm đầu tiên khi thiết kế một chiếc xe hơi mới.

Vậy ai đã nói với bạn rằng "Kết hợp" thì tốt hơn "Kế thừa". Có lẽ để ví dụ sau làm sáng tỏ vấn đề. Các lập trình viên tại MegaGigaCo (phần đầu chương) đều biết về sự "kế thừa" và họ bắt đầu việc thiết kế xe hơi mới bất chấp lời cảnh báo của bạn cho đến khi bạn có cơ hội nói chuyện với họ. Họ biết họ đang phải thiết kế một loạt xe, vì vậy họ bắt đầu tạo ra một lớp cơ sở tên *Vehicle* với một phương thức tên là *go* , phương thức này xuất hiện lên dòng chữ *Now I'm driving*.

```
public abstract class Vehicle
{
  public Vehicle()
  {
  }

  public void go()
  {
    System.out.println("Now I'm driving.");
  }
}
```

Sau đó họ tao tiếp một lớp mới, như là lớp *StreetRacer*, sử dụng *Vehicle* làm lớp cơ sở như sau:

```
public class StreetRacer extends Vehicle
{
  public StreetRacer()
  {
  }
}
```

Chương trình tới đây vẫn tốt đẹp. Bạn có thể cho chạy chương trình với lớp *StreetRacer* như sau:

```
 public static void main(String[] args)
 {
   StreetRacer streetRacer = new StreetRacer();

   streetRacer.go();
       .
       .
       .
}
```

Kết quả nhận được

```
Now I'm driving.
```

Bạn cũng có thể chạy cùng lúc street racer và formula one racer với cùng một cách như sau:

```
 public static void main(String[] args)
 {
   StreetRacer streetRacer = new StreetRacer();
   FormulaOne formulaOne = new FormulaOne();

   streetRacer.go();
   formulaOne.go();
       .
       .
       .
}
```

Và kết quả là

```
Now I'm driving.
Now I'm driving.
```

“Không tồi”. Giám đốc và ban điều hành nói. “Vậy cần gì phải sử dụng mẫu thiết kế” Họ hỏi mà mắt nhìn chằm chằm vào bạn. Nhưng sau đó họ nhận được một hợp đồng sản xuất máy bay trực thăng Helicopter. Máy bay trực thăng à? Họ lý luận, thì cũng là một phương tiện vận chuyển. Vì vậy họ tạo một lớp *Helicopter* , được mở rộng ra từ lớp *Vehicle* :

```
public class Helicopter extends Vehicle
{
  public Helicopter()
  {
  }
}
```

Nhưng lại xuất hiện một vấn đề. Nếu như bạn sử dụng helicopter trong cùng một điều kiện như xe hơi:

```
  public static void main(String[] args)
  {
    StreetRacer streetRacer = new StreetRacer();
    FormulaOne formulaOne = new FormulaOne();
    Helicopter helicopter = new Helicopter();

    streetRacer.go();
    formulaOne.go();
    helicopter.go();
        .
        .
        .
}
```

Bạn sẽ nhận được 3 phương tiện như sau: một xe street racer, một xe Formula One, một máy bay helicopter như sau:

```
Now I'm driving.
Now I'm driving.
Now I'm driving.
```

Có gì đó không ổn, Giám đốc nói một cách hồ nghi. Tại sao helicopter (máy bay trực thăng) mà lại đang chạy? Hình như nó đang bay thì mới đúng? Tuy nhiên vấn đề thực sự tồi tệ khi công ty MegaGigaCo nhận được một hợp đồng chế tạo máy bay phản lực Jet, khi đó chúng cũng được kế thừa từ lớp *Vehicle* :

```
public class Jet extends Vehicle
{
  public Jet()
  {
  }
}
```

Khi bạn cho chạy bốn phương tiện trên: một xe street racer, một xe formula one, một máy bay trực thăng helicopter, một máy bay phản lực jet, bạn nhận được kết quả sau:

```
Now I'm driving.
Now I'm driving.
Now I'm driving.
Now I'm driving.
```

"Chắc chắn là đã có sai sót ở đây" Vị giám đốc lên tiếng. Máy bay phản lực Jet thì không chạy trên đường, chúng ở trên không. Chúng bay và rất nhanh. Không vấn đề gì, các lập trình viên trong công ty đáp. Chúng tôi sẽ ghi đè(override) lên phương thức *go* của lớp *Helicopter* và lớp *Jet* để sửa chữa chúng. Họ chỉnh sửa lại như sau:

```
public class Helicopter extends Vehicle
{
  public Helicopter()
  {
  }

  public void go()
  {
    System.out.println("Now I'm flying.");
  }
}
```

Giờ lớp máy bay trực thăng *Helicopter* đã bay được.

"OK". Giám đốc nói "Tuy nhiên vào tuần sau, ban giám đốc họp và quyết định phải chuyển từ *"Now I'm flying"* sang *"Now, I'm flying 200mph"* và nhiều sự thay đổi tồi tệ kế tiếp…

Có một vấn đề nảy sinh ở đây, bạn giải thích. Các lập trình viên đã thể hiện một chức năng đơn giản – là lái một chiếc xe hay một chiếc phi cơ – qua nhiều lớp con. Đó có thể chưa là một vấn đề lớn nhưng nếu bạn xử lý các công việc này một cách khá thường xuyên, thì việc phải chỉnh sửa mọi lớp con như vậy sẽ trở thành một vấn đề bảo trì khá nghiêm trọng.

Bạn nói tiếp: có thể là "sự kế thừa" không phải là câu trả lời cho tình huống này. Nơi mà bạn cần phải thay đổi chức năng thường xuyên ở các lớp con. Bạn cần phải chỉnh sửa, bảo trì phần lớn các đoạn mã ở các lớp con khi có sự thay đổi. Và khi có càng nhiều lớp kế thừa liên quan, chúng cũng cần được phải bảo trì khi có sự thay đổi, và khi đó bạn phải cập nhật các phương thức *go* mãi mãi.

Vấn đề bạn phải giải quyết ở đây là làm sao tránh được việc thay đổi ở các lớp con. Nếu bạn không tránh được điều này, bạn sẽ phải chỉnh sửa rất nhiều file để cập nhật mã của bạn.

Có lẽ có một cách tốt hơn để xử lý vấn đề này hơn là sử dụng sự "kế thừa".

"Hey" một lập trình viên nói, "Sao anh không sử dụng giao diện interface thay cho sự kế thừa inheritance? Anh có thể cài đặt một giao diện *IFly* và cho giao diện đó một phương thức *go* và để cho lớp *Helicopter* hiện thực giao diện đó như sau:

```
public class Helicopter implements IFly
{
  public Helicopter()
  {
  }

  public void go()
  {
    System.out.println("Now I'm flying.");
  }
}
```

"Không tốt" bạn nói. Anh vẫn chưa giải quyết ổn thỏa vấn đề. Mỗi lớp và lớp con vẫn phải hiện thực cho riêng nó một giao diện, cũng giống như trường hợp của sự kế thừa. Bởi vì giao diện thì không cài đặt nội dung, bạn vẫn phải viết code cho từng lớp, điều này có nghĩa là chẳng có sử dụng lại được một đoạn code nào cả.

Nắm vững sự thay đổi từ "is-a" sang "has-a"

Mọi việc đều thay đổi. Trong thời buổi thương mại phát triển, mọi thứ thay đổi nhanh chóng. Vì vậy việc lập kế hoạch cho sự thay đổi là rất đáng giá. Nếu bạn có một vấn đề nhỏ cần phải có một giải pháp nhỏ, bạn có thể không cần phải lập một kế hoạch lớn lao cho sự thay đổi. Nhưng nếu bạn làm việc trong một dự án nghiêm túc, với một khối lượng công việc đáng kể, thì đúng là lúc bạn nên nhìn lại về một kế hoạch nghiêm túc khi có sự thay đổi. Các đoạn mã mà bạn viết hôm nay, sẽ phải chỉnh sửa lại để phù hợp với những yêu cầu phát triển trong tương lai. Hầu hết các nhà phát triển không chú ý tới vấn đề này, và sau đó họ luôn luôn hối tiếc. Vậy câu hỏi đặt ra là dự án phải lớn tới đâu, để bạn quan tâm đến vấn đề thay đổi. Đó là sự đánh giá của riêng bạn, một phần của nghệ thuật lập trình. Bằng cách nắm vững phương pháp xử lý sự thay đổi, bạn sẽ biết rõ hơn khi nào thì nên thực hiện nó.

Có một dấu hiệu đáng chú ý ở đây: Phân chia các đoạn mã dễ thay đổi trong chương trình riêng biệt với phần còn lại. Và làm cho chúng càng độc lập càng tốt cho sự bảo trì nâng cấp. Bạn cũng nên cố gắng tái sử dụng những phần này càng nhiều càng tốt.

Điều này có nghĩa là nếu ứng dụng của bạn có một phần bị thay đổi, bạn có thể đem nó riêng ra, sau đó thay đổi từng phần riêng biệt một cách dễ dàng trong khi vẫn không bị ảnh hưởng bởi những tác dụng phụ của nó.

Và đây là cách để lập kế hoạch cho sự thay đổi, và vì sao "kế thừa" lại không thể giải quyết tốt các sự thay đổi này. Với sự kế thừa, lớp cơ sở và các lớp con có một mối quan hệ "is-a". Ví dụ , lớp *Helicopter* có quan hệ "is-a" với lớp *Vehicle*, điều này có nghĩa *Helicopter* thừa kế mọi thứ từ *Vehicle*, và nếu bạn phải chỉnh sửa các phương thức này, bạn sẽ gặp phải vấn đề bảo trì nó trong tương lai. Lớp cơ sở xử lý phương thức theo một cách, và lớp kế thừa lại thay đổi nó, và lớp kế tiếp lại thay đổi nó thêm một lần nữa. Và cuối cùng bạn có một lô một lốc các biến thể của cùng 1 phương thức qua các lớp con.

Mặc khác, nếu bạn có thể trích những đoạn code dễ thay đổi và đóng gói chúng vào đối tượng, bạn có thể sử dụng các đối tượng này khi cần. Nhiệm vụ mới là xử lý trên các đối tượng này. Bạn đã không để việc xử lý lây lan qua các lớp con. Làm như vậy sẽ cho phép bạn chỉnh sửa mã của bạn bằng việc tạo ra "sự kết hợp" composites các đối tượng. Với composites "kết hợp" này, bạn có thể dễ dàng chọn ra và sử dụng đối tượng cần thiết. Một quan hệ "has-a" mới được tạo ra. Một chiếc xe street racer sẽ có một "has-a" cách để di chuyển, đã được đóng gói vào đối tượng. Một máy bay trực thăng sẽ có một cách riêng để di chuyển, và cũng được đóng gói vào đối tượng. Từng đối tượng sẽ thực hiện hành động của riêng nó.

Một đối tượng, một nhiệm vụ thường là có ý nghĩa hơn là việc kế thừa các lớp, và tạo ra hàng tá các lớp con. Nói cách khác, chúng ta sắp xếp lại dựa trên nhiệm vụ của lớp, chứ không phải trên sự kế thừa.

Sử dụng kế thừa sẽ tự động cài đặt mọi thuộc tính một cách nghiêm ngặt, bao gồm cả quan hệ "is-a", là thứ gây ra các rắc rối khi bảo trì cũng như khi mở rộng. Nếu bạn đặt kế hoạch cho sự thay đổi, bạn nên nghĩ tới quan hệ "has-a" , nơi mà mã của bạn bao gồm nhiều đối tượng mà có thể dễ dàng cập nhật khi có sự thay đổi xảy ra.

Gợi ý: Khi có kế hoạch cho sự thay đổi, hãy thay thế quan hệ "is-a" thành quan hệ "has-a" và đặt các đoạn mã dễ thay đổi vào các đối tượng trong ứng dụng này hơn là kế thừa chúng.

KẾ HOẠCH CHỈNH SỬA

Làm thế nào mà ý tưởng phân chia các đoạn mã dễ thay đổi sẽ hoạt động trong ví dụ *Vehicle/StreetRacer/Helicopter* đã nhắc trước đây. Theo ý kiến của giám đốc điều hành, phần được thay đổi nhiều nhất là phương thức *go* , do đó chúng ta sẽ tách nó ra. Trong thuật ngữ về thiết kế mẫu, mỗi cách hiện thực một phương thức được gọi là 1 thuật toán(algorithm) hay có thể gọi là 1 chiến lược (strategy). Vì vậy bạn có thể tạo một tập hợp các giải thuật để sử dụng cho các biến của bạn như *StreetRacer, FormulaOne, Helicopter,* và *Jet* . Làm như thế để phân chia các đoạn mã dễ thay đổi vào trong thuật toán. Từng thuật toán sẽ hoàn thành 1 nhiệm vụ.

Cách tạo thuật toán

Để chắc chắn mọi thuật toán đều hiện thực cùng một phương thức (phương thức *go* ở trên). Bạn cần phải tạo một giao diện interface cho nó (ND: Interface là một khái niệm rất hay trong OOP, mà khi có dịp chúng ta sẽ thảo luận về nó) như sau:

```
public interface GoAlgorithm
{
  public void go();
}
```

Giao diện GoAlgorithm có một phương thức duy nhất *go.* Để chắc chắn rằng mọi thuận toán có thể được sử dụng bởi bất kì lớp *Vehicle* nào, ta cần phải hiện thực interface này. Thuật toán đầu tiên *GoByDrivingAlgorithm* , sẽ hiển thị văn bản *"Now I'm driving"*. Và đây là mã của thuật toán:

```
public class GoByDrivingAlgorithm implements GoAlgorithm
{
  public void go()
  {
    System.out.println("Now I'm driving.");
  }
}
```

Ngoài ra, thuật toán *GoByFlying*, sẽ hiển thị văn bản *Now I'm flying*. Mã như sau:

```
public class GoByFlying implements GoAlgorithm
{
  public void go() {
    System.out.println("Now I'm flying.");
  }
}
```

Và cuối cùng, thuật toán *GoByFlyingFast*, sẽ được sử dụng bởi máy bay phản lực, hiển thị dòng văn bản *Now I'm flying fast*

```
public class GoByFlyingFast implements GoAlgorithm
{
  public void go()
  {
    System.out.println("Now I'm flying fast.");
  }
}
```

Tuyệt vời. Bạn vừa phân chia các thuật toán của mình ra khỏi phần mã. Bạn đang thực hiện thao tác thực thi quan hệ "has-a" hơn là quan hệ "is-a". Bây giờ bạn đã có thể đưa các thuật toán này vào sử dụng.

SỬ DỤNG THUẬT TOÁN

Bạn đang có một số thuật toán, bạn có thể tạo các đối tượng và sử dụng quan hệ "has-a" thay cho "is-a". Sau khi bạn tạo một đối tượng từ một thuật toán, bạn cần phải lưu trữ đối tượng ở đâu đó. Vì vậy hãy thêm vào lớp cơ sở *Vehicle*, một phương thức mới *SetGoAlgorithm*. Phương thức này sẽ lưu trữ thuật toán mà bạn muốn sử dụng. Mã như sau:

```
public abstract class Vehicle
{
  private GoAlgorithm goAlgorithm;

  public Vehicle()
  {
  }

  public void setGoAlgorithm (GoAlgorithm algorithm)
  {
    goAlgorithm = algorithm;
  }
     .
     .
     .
}
```

Bây giờ khi bạn muốn sử dụng một thuật toán cụ thể nào đó ở lớp kế thừa, tất cả việc cần làm là gọi phương thức *setGoAlgorithm* với một đối tượng thuật toán đúng, theo cách như sau:

```
setGoAlgorithm(new GoByDrivingAlgorithm());
```

Phương thức *go* của lớp *Vehicle* có chút thay đổi. Trước đây là:

```
public void go()
{
  System.out.println("Now I'm driving.");
}
```

Tuy nhiên, bây giờ nó phải gọi phương thức đã được định nghĩa ở các lớp thuật toán. Mã mới như sau:

```
public abstract class Vehicle
{
  private GoAlgorithm goAlgorithm;

  public Vehicle()
  {
  }

  public void setGoAlgorithm (GoAlgorithm algorithm)
  {
    goAlgorithm = algorithm;
  }

  public void go() {
    goAlgorithm.go();
  }
}
```

Bây giờ thì tất cả những gì phải làm là chọn đúng thuật toán mà bạn muốn sử dụng cho phương tiện nào đó. Ví dụ với street racer sẽ là thuật toán *GoByDrivingAlgorithm*

```
public class StreetRacer extends Vehicle
{
  public StreetRacer()
  {
    setGoAlgorithm(new GoByDrivingAlgorithm());
  }
}
```

Xe Formula One cũng sử dụng cùng một thuật toán trên, mã như sau:

```
public class FormulaOne extends Vehicle
{
  public FormulaOne()
  {
    setGoAlgorithm(new GoByDrivingAlgorithm());
  }
}
```

Nhưng máy bay trực thăng helicopter sẽ sử dụng thuật toán *GoByFlyingAlgorithm*:

```
public class Helicopter extends Vehicle
{
  public Helicopter()
  {
    setGoAlgorithm(new GoByFlyingAlgorithm());
  }
}
```

Và máy bay phản lực Jet sẽ sử dụng thuật toán *GoByFlyingFastAlgorithm*

```
public class Jet extends Vehicle
{
  public Jet()
  {
    setGoAlgorithm(new GoByFlyingFastAlgorithm());
  }
}
```

OK. Đã đến lúc chạy thử chương trình. Biên dịch và chạy thử chương trình như sau:

```
public class StartTheRace
{

  public static void main(String[] args)
  {
    StreetRacer streetRacer = new StreetRacer();
    FormulaOne formulaOne = new FormulaOne();
    Helicopter helicopter = new Helicopter();
    Jet jet = new Jet();

    streetRacer.go();
    formulaOne.go();
    helicopter.go();
    jet.go();

  }
}
```

Kết quả:

```
Now I'm driving.
Now I'm driving.
Now I'm flying.
Now I'm flying fast.
```

Kết quả đúng như mong đợi. Tuy nhiên bây giờ bạn đã sử dụng mối quan hệ "has-a" thay vì quan hệ kế thừa "is-a". Từ lúc này bạn có thể sử dụng các thuật toán xuyên suốt chương trình, bất cứ đâu, vì nó đã không còn nằm trong các lớp *StreetRacer* hay *Helicopter* nữa.

Kỹ thuật này thay thế cho cách tạo các lớp con và sử dụng kế thừa. Nếu bạn sử dụng một quan hệ kế thừa "is-a", bạn sẽ bắt đầu sự rắc rối cho việc kiểm soát được các phương thức trong lớp cơ sở và các lớp con – trong ví dụ là bạn phải nạp đè lên phương thức *go* cho lớp *Helicopter* và *Jet*. Nếu bạn sử dụng mô hình "has-a", bạn có thể tạo ra một dòng họ các thuật toán một cách rõ ràng, và sau đó bạn chọn một thuật toán thích hợp để sử dụng.

Theo cách này, bạn đã có thể khắc phục được vấn đề mà sự kế thừa đã gây ra cho hầu hết các lập trình viên: nếu bạn phải giải quyết một chức năng cụ thể nào đó qua nhiều thế hệ của một lớp, và chức năng này liên tục thay đổi, bạn sẽ phải chỉnh sửa rất nhiều mã của mình. Mặt khác, khi bạn tập trung chức năng đó vào một thuật toán duy nhất, việc thay đổi nó sẽ dễ dàng hơn rất nhiều.

Quay lại ví dụ trên, khi ban giám đốc muốn thay đổi từ *"Now I'm flying"* sang *"Now I'm flying at 20 mph"*. Đơn giản, bạn chỉ cần chỉnh sửa thuật toán *GoByFlying:*

```
public class GoByFlying implements GoAlgorithm
{
  public void go() {
    System.out.println("Now I'm flying at 200 mph.");
  }
}
```

Và bây giờ tất cả mã của bạn đã tự động được cập nhật, bạn không cần thiết phải đi tìm và chỉnh sửa từng lớp con như trước nữa. Theo cách này, bạn đã tập trung sự xử lý một chức năng vào một đối tượng thuật toán duy nhất, bạn sẽ dễ dàng quản lý đối tượng này trong trường hợp yêu cầu chức năng bị thay đổi.

Chọn lựa thuật toán khi thực thi chương trình

“Đợi một chút,” Giám đốc MegaGigaCo nói. “Có việc xảy ra, máy bay phản lực không chỉ bay nhanh, đầu tiên nó chạy trên đường băng một lúc, và khi đáp xuống mặt đất, nó lại tiếp tục chạy trên đường băng nữa. Vì vậy chúng ta phải chỉnh sửa chức năng cho nó lại : đầu tiên là chạy trên đường băng, rồi bay, rồi chạy tiếp?”

“Đó là về mặt lý thuyết,” các lập trình viên rên rỉ “Nhưng điều đó làm chúng ta phải viết thêm nhiều đoạn mã nữa”

“Không sao cả” Bạn nói. “Đó là một trong những điểm kỳ diệu của việc sử dụng một đối tượng thuật toán bên ngoài. Bạn có thể thay đổi nó khi bạn thực thi chương trình”

Khi bạn viết mã cho một chức năng trong một lớp, bạn không thể thay đổi nó khi thực thi chương trình. Tuy nhiên khi bạn sử dụng một đối tượng thuật toán bên ngoài với mối quan hệ “has-a”, bạn dễ dàng thay đổi chức năng đó lúc chương trình hoạt động. Nói cách khác một quan hệ “has-a” cho phép bạn dễ dàng thay đổi hơn một quan hệ “is-a” đặc biệt khi chương trình đang hoạt động.

Và đây là ví dụ cho việc sử dụng linh hoạt các thuật toán, cũng như việc thay đổi nó khi chương trình đang chạy. Bạn có thể tạo một máy ban phản lực, có thể chạy trên đường băng với thuật toán *GoByDrivingAlgorithm*, như mã sau:

```
public class RealJet
{

  public static void main(String[] args)
  {
    Jet jet = new Jet();

    jet.setGoAlgorithm(new GoByDrivingAlgorithm());
        .
        .
        .
}
```

Để máy bay phản lực chạy được trên đường băng, bạn gọi phương thức *go*:

```
public class RealJet
{

  public static void main(String[] args)
  {
    Jet jet = new Jet();

    jet.setGoAlgorithm(new GoByDrivingAlgorithm());
    jet.go();
        .
        .
        .
}
```

Bạn có thể cài đặt thuật toán mới *setGoAlgorithm* cho máy bay phản lực, để thay đổi phương thức *go* một cách linh động, và sau đó gọi lại phương thức *go* để thấy sự khác biệt.

```
public class RealJet
{

  public static void main(String[] args)
  {
    Jet jet = new Jet();

    jet.setGoAlgorithm(new GoByDrivingAlgorithm());
    jet.go();

    jet.setGoAlgorithm(new GoByFlyingFastAlgorithm());
    jet.go();

    jet.setGoAlgorithm(new GoByDrivingAlgorithm());
    jet.go();
  }
}
```

Và đây là kết quả: máy bay phản lực, chạy trên đường băng, rồi bay, rồi chạy trên đường, không vấn đề gì cả.

```
Now I'm driving.
Now I'm flying fast.
Now I'm driving.
```

Bạn thấy đó, việc chuyển đổi một thuật toán lúc thực thi chương trình rất dễ dàng. Nói cách khác, nếu bạn để việc xử lý thuật toán vào nội tại một lớp, bạn sẽ không thể thay đổi nó lúc chạy chương trình. Nhưng khi bạn cài đặt một chiến lược “Strategy”, bạn sẽ

dễ dàng thay đổi nó khi chạy chương trình. Tất cả những điều trên mang chúng ta đến một mẫu thiết kế "Strategy", hay được gọi là mẫu "chiến lược".

MẪU "STRATEGY" – Mẫu chiến lược

Mẫu chiến lược là mẫu thiết kế chúng ta học đầu tiên trong quyển sách này, và thực tế là chúng ta đã cùng nhau đi xuyên suốt qua chương này để hiểu về nó. Ý nghĩa thực sự của mẫu chiến lược là bạn tách rời phần xử lý một chức năng cụ thể ra khỏi đối tượng của bạn. Sau đó tạo ra một tập hợp các thuật toán để xử lý chức năng đó và lựa chọn thuật toán nào mà bạn thấy đúng đắn nhất khi thực thi chương trình. Mẫu thiết kế này thường được sử dụng để thay thế cho sự kế thừa, khi bạn muốn chấm dứt việc theo dõi và chỉnh sửa một chức năng qua nhiều lớp con.

Chúng ta có thể nhìn thấy vấn đề tổng quát như sau. Đầu tiên mọi việc đều ổn, bạn có một đối tượng, một chức năng

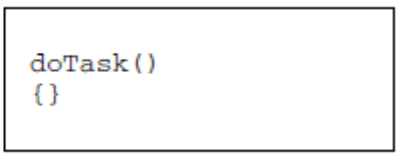


Figure 2-1: One object, one task.

Một thời gian sau đó, do yêu cầu đặc biệt, bạn cần có thêm một lớp mới, bạn kế thừa lớp cũ, và ghi đè lên phương thức đã được thừa hưởng. Bạn đang dàn trải việc xử lý chức năng qua nhiều lớp con như hình:

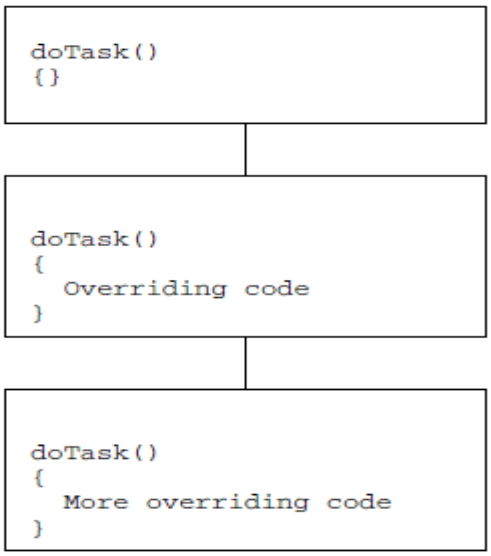


Figure 2-2: Adding tasks requires rewriting code.

Mẫu “Strategy”, mẫu chiến lược nói rằng: bạn cần phải tách những phần dễ thay đổi và đóng gói chúng vào các đối tượng và bạn có thể sử dụng các đối tượng này khi cần. Bây giờ bạn có thể chỉnh sửa mã của mình thông qua việc tạo sự “kết hợp” các đối tượng. Khi chương trình thực thi, bạn chỉ cần sử dụng đúng đối tượng mà bạn cần. Như hình sau:

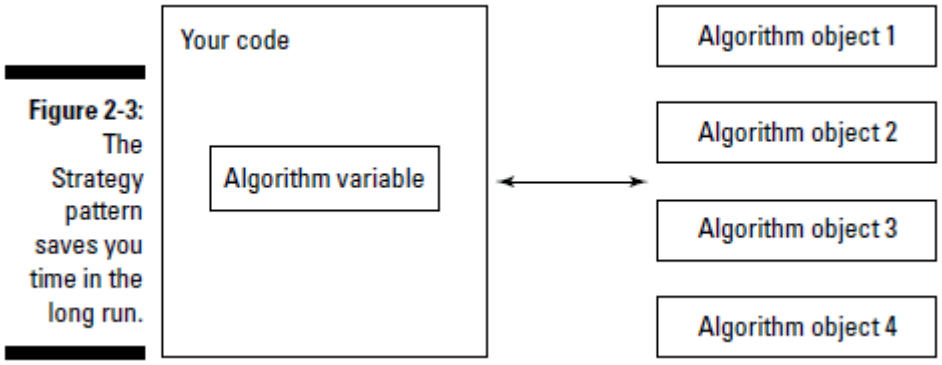


Figure 2-3: The Strategy pattern saves you time in the long run.

Cuốn sách GoF đã nói rằng mẫu Strategy, mẫu chiến lược như sau: “Định nghĩa một tập hợp các thuật toán, đóng gói chúng thành từng loại một, và giúp chúng có thể hoán đổi cho nhau. Mẫu chiến lược giúp các thuật toán độc lập hơn khi được sử dụng.

Mẫu chiến lược chỉ ra rằng, đôi khi, nó sẽ được áp dụng tốt cho mục đích hướng chức năng. Và nó đặc biệt quan trọng khi bạn muốn thực hiện công việc nâng cấp, bảo trì cho các đoạn mã dễ thay đổi của bạn một cách riêng biệt với toàn bộ mã của chương trình, hoặc khi bạn muốn thay đổi thuật toán sử dụng khi chương trình được thực thi.

Gợi ý: Bạn nên sử dụng mẫu Strategy khi có những tình huống sau:

- Bạn có một đoạn mã dễ thay đổi, và bạn tách chúng ra khỏi chương trình chính để dễ dàng bảo trì
- Bạn muốn tránh sự rắc rối, khi phải hiện thực một chức năng nào đó qua quá nhiều lớp con.
- Bạn muốn thay đổi thuật toán sử dụng khi chạy chương trình

Chúng tôi không cung cấp mã nguồn cho bạn. Thay vào đó, bạn hãy làm quen với việc suy nghĩ, và khi ý tưởng tới đó là lúc bạn đã nắm vững mẫu thiết kế này. Điều này cũng giống như một công việc phải thực hiện cho mẫu “Chiến lược”.

Việc hiểu biết cách thức làm việc của các mẫu thiết kế khác nhau cũng giúp bạn có cơ hội thảo luận với đồng nghiệp khác. Hầu hết các lập trình viên chuyên nghiệp đều biết một số mẫu thiết kế cơ bản. Và khi mọi người trong nhóm của bạn nói tới mẫu chiến lược, mọi người gật đầu ra vẻ hiểu biết, thì bạn cũng có thể làm như vậy.

DP4Dummies – Chương 3: Decorator, Factory

CHƯƠNG 3: TẠO VÀ MỞ RỘNG MỘT ĐỐI TƯỢNG VỚI MẪU DECORATOR VÀ FACTORY

Trong chương này, chúng ta sẽ thảo luận về một số nội dung sau:

- Giữ vững nguyên tắc viết mã "Open-Close" hay "Luôn mở cho việc mở rộng, nhưng đóng cho việc sửa đổi"
- Giới thiệu về mẫu trang trí Decorator
- Các ví dụ về mẫu trang trí Decorator
- Xây dựng các đối tượng với mẫu nhà máy Factory
- Đóng gói việc khởi tạo đối tượng bằng mẫu nhà máy Factory
- Sử dụng các phương thức khởi tạo nhà máy Factory Method

Bạn đang làm nhân viên tư vấn Thiết Kế Mẫu tại công ty Giganto Computer, với mức lương khá cao và bạn đang ở trong căn tin công ty.

"Hôm nay có món gì?" bạn hỏi tay đầu bếp khó chịu đang đứng sau bếp nướng.

"Cho một cái hamburger," bạn nói và xoay xoay cái khay trong tay.

Người đầu bếp mang cái hamburger đến bàn tính tiền, không quên hỏi lại "Có thêm thịt rán không?"

"Chắc chắn rồi", bạn nói.

Người đầu bếp xóa phiếu ăn cũ trên máy tính tiền và làm lại phiếu ăn.

"Hamburger và thịt rán". Vừa nói anh ta vừa gõ vào máy tính tiền.

"Cho thêm một ít pho mát" Bạn nói.

Người đầu bếp ném một ánh nhìn khó chịu , xóa cái phiếu ăn, mổ mổ cái bàn phím và nói “Hamburger với pho mát và thịt nướng. Ok. Đủ rồi chứ?”

“Hmm”, bạn nói, nhìn quét qua cái thực đơn “Hay là thêm một chút thịt xông khói?”

Người đầu bếp nhìn chằm chằm vào bạn và dường như định văng ra một vài câu khó chịu gì đó nhưng vẫn nhập phiếu ăn vào máy.

“Hey”, bạn nói. “Anh chắc chắn là được lợi nhiều hơn từ việc sử dụng mẫu thiết kế trang trí Decorator chứ hả?”

“Vâng”, anh đầu bếp trả lời, tự hỏi về những gì bạn nói “Tôi đã nói điều này cả ngàn lần rồi”

Bạn cầm cái Hamburger pho mát thịt xông khói với vẻ hạnh phúc và nói “Thêm một vài lát cà chua nữa thì tuyệt!”

Chương này nói về hai mẫu thiết kế quan trọng, nó sẽ lấp đầy những thiếu sót trong việc lập trình hướng đối tượng cơ bản, đặc biệt là ở khả năng kế thừa. Đây là hai mẫu trang trí Decorator và mẫu nhà máy Factory.

Mẫu trang trí Decorator là lựa chọn hoàn hảo cho tình huống tôi vừa nêu ở trên bởi vì ta đang nói về khả năng mở rộng chức năng cho một lớp có sẵn. Sau khi viết một lớp, bạn có thể thêm phần trang trí Decorator (các lớp mở rộng) để mở rộng lớp này. Khi đó bạn không phải sửa đổi lên lớp gốc. Kết quả là cái Hamburger của bạn trở thành Hamburger pho mát, rồi Hamburger pho mát thịt xông khói, mọi thứ thật dễ dàng.

Nguyên lý Open-Close – “Luôn Open cho việc mở rộng và Closed cho việc sửa đổi”

Một trong những khía cạnh quan trọng nhất trong quá trình phát triển một ứng dụng là các nhà phát triển và lập trình viên phải đối đầu với sự thay đổi, và đó là lý do vì sao các mẫu thiết kế này lại được giới thiệu trước tiên. Có thể nói các Mẫu Thiết Kế sẽ giúp bạn giải quyết được các sự thay đổi, và bạn có thể dễ dàng chuyển đổi mã nguồn của mình cho các trường hợp mới và bất khả kháng. Như tôi đã nói qua trong suốt cuốn sách này,lập trình viên thường tiêu tốn thời gian cho việc mở rộng và thay đổi mã nguồn hơn là phát triển mã nguồn gốc.

Mẫu chiến lược Strategy đã được giới thiệu trước đây trong chương II, giúp bạn xử lý những sự thay đổi bằng cách cho phép bạn chọn lựa một thuật toán thích hợp từ một tập hợp thuật toán bên ngoài hơn là phải viết lại mã nguồn. Mẫu trang trí Decorator cũng tương tự vậy, nó cho phép bạn viết tiếp mã nguồn, tránh việc sửa đổi lên mã nguồn gốc, trong khi vẫn đáp ứng được yêu cầu thay đổi. Đó là điểm chính yếu tôi muốn nhấn mạnh.

Ghi nhớ:Hãy làm cho mã nguồn của bạn đáp ứng được nguyên tắc "Luôn đóng cho sự chỉnh sửa, và luôn mở cho việc mở rộng" càng nhiều càng tốt. Nói cách khác, hãy thiết kế mã nguồn sao cho không cần phải thay đổi gì nhiều nhưng luôn có thể mở rộng khi cần.

Đây là một ví dụ cho việc viết mã nguồn luôn đóng cho sự thay đổi.

Công ty mà bạn đang làm tư vấn, công ty GigantoComputer, quyết định làm một cái máy vi tính mới. Đây là mã nguồn của lớp *Computer*:

```
public class Computer
{
  public Computer()
  {
  }

  public String description()
  {
    return "You're getting a computer.";
  }
}
```

Khi một đối tượng computer được khởi tạo. Phương thức *description* sẽ trả về văn bản *"You're getting a computer."* . Tới giờ mọi việc vẫn tốt đẹp. Nhưng một số khách hàng quyết định rằng họ muốn có một cái đĩa cứng trong máy tính. "Không vấn đề gì cả" Các lập viên trong công ty đáp. " Chúng ta chỉ cần chỉnh sửa mã nguồn lại một chút như sau:"

```
public class Computer
{
  public Computer()
  {
  }

  public String description()
  {
    return "You're getting a computer and a disk.";
  }
}
```

Bây giờ khi một đối tượng computer được tạo và bạn gọi phương thức *description*, bạn sẽ nhận được văn bản *"You're getting a computer and a disk."* Nhưng một vài khách hàng vẫn chưa hài lòng. Họ muốn thêm một cái màn hình nữa. Và thế là các lập trình viên phải chỉnh sửa tiếp như sau:

```
public class Computer
{
  public Computer()
  {
  }

  public String description()
  {
    return "You're getting a computer and a disk and a monitor.";
  }
}
```

Bây giờ, khi bạn tạo một computer và gọi phương thức *description* bạn sẽ thấy

```
You're getting a computer and a disk and a monitor.
```

Bạn có thể thấy vấn đề ở đây: Các lập trình viên phải thay đổi mã nguồn mỗi khi khách hàng thay đổi yêu cầu của họ. Rõ ràng, đó là vấn đề chính.

Và bạn, với cương vị là tư vấn mẫu thiết kế, sẽ chỉnh sửa nó.

MẪU TRANG TRÍ DECORATOR?

Tôi phải nhắc lại một lần nữa: càng nhiều càng tốt, hãy viết mã nguồn của bạn đóng cho việc sửa đổi, nhưng mở cho việc mở rộng. Trong chương II, bạn đã biết cách làm việc với mẫu chiến lược Strategy. Đó là, bạn đóng gói mã nguồn vào các thuật toán riêng biệt để sử dụng dễ dàng, hơn là việc xử lý chúng thông qua các lớp con.

Mẫu trang trí Decorator có một cách tiếp cận khác. Thay vì sử dụng một thuật toán bên ngoài, mẫu thiết kế này sử dụng một phương pháp "bao bọc" mã nguồn của bạn để mở rộng chúng.

Ghi nhớ: Định nghĩa chính thức của mẫu trang trí Decorator trong sách của GoF có viết: "Gắn kết thêm một số tính năng cho đối tượng một cách linh động. Mẫu trang trí Decorator cung cấp một phương pháp linh hoạt hơn là sử dụng lớp con để mở rộng chức năng cho đối tượng"

Mẫu thiết kế này được gọi là "Người trang trí" Decorator nhưng dường như đó là tên gọi rườm rà. Một cái tên tốt hơn cho mẫu này có thể là "Người tăng thêm" Augmentor hay "Người mở rộng" Extender bởi vì nó cho phép bạn: tăng thêm hay mở rộng một lớp một cách linh động khi chương trình được thực thi. Tuy nhiên, như bạn thấy trong chương này, thuật ngữ "Người trang trí" Decorator còn giúp bạn hiểu rõ hơn khái niệm "đóng cho việc chỉnh sửa, mở cho việc mở rộng". Khi bạn làm hành động bao bọc mã nguồn để mở rộng thêm chức năng, bạn không cần thiết chỉnh sửa lại mã nguồn cũ, bạn chủ yếu tập trung vào việc trang trí nó.

Và đây là cách mà nó làm việc. Bạn bắt đầu với một cái máy tính computer đơn giản sau:

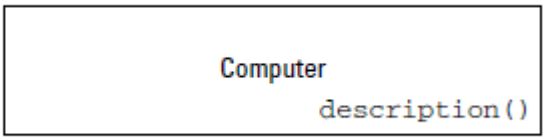


Figure 3-1: You start with a core computer.

Khi bạn gọi phương thức *description*, bạn nhận được kết quả "*You're getting a computer"*. Bây giờ bạn muốn thêm ít phần cứng, một ổ cứng mới chẳng hạn. Trong trường hợp này, bạn có thể thêm một lớp bao bọc wrapper như sau:

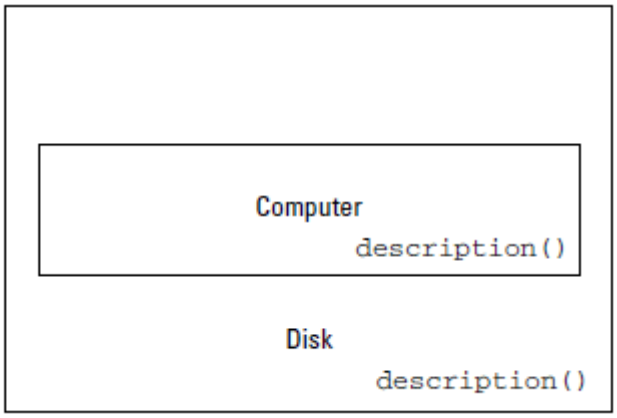


Figure 3-2: Next, you add a disk drive.

Bây giờ khi bạn gọi phương thức *description* của lớp bao bọc wrapper, nó sẽ gọi phương thức *description* của đối tượng computer để nhận được kết quả "*You're getting a computer* " và đối tượng ổ cứng *disk* sẽ trả về kết quả "*and a disk".* Kết quả bạn nhận được *"You're getting a computer and a disk"*

Nếu bạn muốn thêm vài thứ nữa vào lớp máy tính *Computer*, bạn hãy đặt nó vào lớp bao bọc *wrapper*, ví dụ như thêm vào cái màn hình *Monitor*:

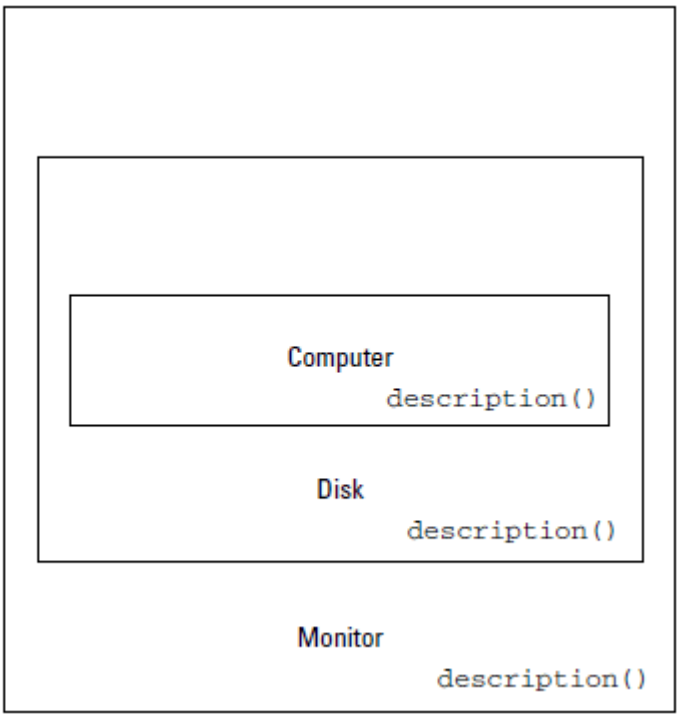


Figure 3-3: Finally, you add a monitor to the computer.

Bây giờ khi bạn gọi phương thức *description*, mọi việc sẽ xảy ra như sau:

- Đối tượng computer, sẽ thực hiện phương thức *description* để tạo ra kết quả *"You're getting a computer"*

- Đối tượng disk, sẽ thực hiện tiếp phương thức trên để thêm vào *"and a disk"*

- Đối tượng monitor, tiếp tục thực hiện phương thức *description* để thêm vào *"and a monitor"*

- Kết quả là bạn nhận được *"You're getting a computer and a disk and a monitor"*

VÍ DỤ VỀ MẪU TRANG TRÍ DECORATOR

Bạn bắt đầu viết một lớp máy tính Computer đơn giản, với một phương thức *description* trả về kết quả "*computer"* như sau:

```
public class Computer
{
  public Computer()
  {
  }

  public String description()
  {
    return "computer";
  }
}
```

OK. Bạn đã hoàn thành cái máy tính đơn giản. Bây giờ làm sao để tạo một lớp trang trí? Những lớp này hoạt động như là một lớp bao bọc cho lớp *Computer*, điều này có nghĩa là phải có một biến để lưu trữ một đối tượng *computer*. Một cách đơn giản để tạo lớp bao bọc wrapper là mở rộng lớp *Computer*.

Tạo dựng một Lớp trang trí Decorator

Bạn có thể bắt đầu bằng việc tạo một lớp trừu tượng được mở rộng từ lớp *Computer* (nhớ rằng lớp trừu tượng sẽ không thể sử dụng trực tiếp được, bạn phải kế thừa từ lớp này, và tạo ra lớp mới để sử dụng). Đây là mã nguồn:

```
public abstract class ComponentDecorator extends Computer
{
  public abstract String description();
}
```

Lớp mới này, *ComponentDecorator* , có một phương thức trừu tượng tên *description*. Bởi vì lớp này là trừu tượng nên bạn không thể tạo đối tượng từ nó. Điều đó có nghĩa là bạn đã chắn chắn mọi lớp bao bọc wrapper kế thừa từ lớp này phải nhất quán, và khi đó mọi lớp kế thừa sẽ có một phương thức description riêng khác nhau.

Thêm vào một đĩa cứng *Disk*

Đây là lớp bao bọc *Disk* , sẽ thêm một ổ cứng vào máy tính. Lớp này sẽ mở rộng từ lớp trừu tượng *ComponentDecorator*

```
public class Disk extends ComponentDecorator
{
    .
    .
    .
}
```

Bởi vì đây là một lớp bao bọc, nó cần phải biết đang bao bọc thứ gì. Vì vậy bạn đưa cho nó một đối tượng *computer* ngay khi nó khởi tạo. Lớp bao bọc *Disk* sẽ lưu trữ một đối tượng tên *computer*

```
public class Disk extends ComponentDecorator
{
  Computer computer;

  public Disk(Computer c)
  {
    computer = c;
  }
   .
   .
   .
}
```

Bây giờ bạn cần hiện thực phương thức Description. (Lưu ý: khi bạn kế thừa một lớp trừu tượng trong Java, bạn cần hiện thức tất cả các phương thức trừu tượng của lớp đó). Phương thức mới này sẽ gọi phương thức *description* của lớp *computer* và thêm vào dòng chữ *"and a disk"* như sau:

```
public class Disk extends ComponentDecorator
{
  Computer computer;

  public Disk(Computer c)
  {
    computer = c;
  }

  public String description()
  {
    return computer.description() + " and a disk";
  }
}
```

Vậy là bạn đã bao bọc đối tượng *computer*, và khi bạn gọi phương thức *description* của đối tượng *disk* này, nó sẽ gọi phương thức *description* của lớp computer, đồng thời thêm vào dòng chữ *"and a disk"*. Kết quả bạn sẽ có *"computer and a disk"*

Thêm vào một ổ CD

Bạn cũng có thể thêm vào một ổ CD theo cùng cách trên. Đây là mã nguồn

```
public class CD extends ComponentDecorator
{
  Computer computer;

  public CD(Computer c)
  {
    computer = c;
  }

  public String description()
  {
    return computer.description() + " and a CD";
  }
}
```

Thêm vào một màn hình monitor

Tất nhiên bạn cũng có thêm vào một màn hình theo cùng một cách như sau:

```
public class Monitor extends ComponentDecorator
{
  Computer computer;

  public Monitor(Computer c)
  {
    computer = c;
  }

  public String description()
  {
    return computer.description() + " and a monitor";
  }
}
```

OK. Bạn đã có đầy đủ các lớp. Giờ là lúc chạy thử nghiệm chương trình.

Đầu tiên bạn tạo đối tượng c*omputer* như sau:

```
public class Test
{
  public static void main(String args[])
  {
    Computer computer = new Computer();
        .
        .
        .
  }
}
```

Sau đó bạn bao bọc đối tượng *computer* để thêm vào một đĩa cứng

```
public class Test
{
  public static void main(String args[])
  {
    Computer computer = new Computer();

    computer = new Disk(computer);
        .
        .
        .
  }
}
```

Bây giờ hãy thêm vào một monitor:

```
public class Test
{
  public static void main(String args[])
  {
    Computer computer = new Computer();

    computer = new Disk(computer);
    computer = new Monitor(computer);
        .
        .
        .
  }
}
```

Sau đó, bạn có thêm vào không chỉ một ổ CD, mà là hai ổ CD chẳng hạn. Không vấn đề gì lớn lao cả. Cuối cùng bạn gọi phương thức *description* của lớp bao bọc để xem kết quả:

```
public class Test
{
  public static void main(String args[])
  {
    Computer computer = new Computer();

    computer = new Disk(computer);
    computer = new Monitor(computer);
    computer = new CD(computer);
    computer = new CD(computer);

    System.out.println("You're getting a " + computer.description()
      + ".");
  }
}
```

OK. Khi chạy chương trình bạn nhận được kết quả.

```
You're getting a computer and a disk and a monitor and a CD and a CD.
```

Không tồi. Bạn đã mở rộng một đối tượng gốc thật đơn giản bằng cách bao bọc nó trong nhiều lớp trang trí decorator khác nhau, tránh việc phải chỉnh sửa trong mã nguồn gốc. Và đó là Mẫu Thiết Kế Trang Trí Decorator.

CẢI TIẾN TOÁN TỬ NEW VỚI MẪU THIẾT KẾ NHÀ MÁY FACTORY

Tại đây, công ty MegaGigaCo, bạn được trả lương cao cho kỹ năng thiết kế mẫu chuyên nghiệp của mình, bạn đang tạo một đối tượng kết nối cơ sở dữ liệu mới. Hãy xem toán tử new trong Java làm việc này như thế nào?

```
Connection connection = new OracleConnection();
```

“Không tồi,” bạn nghĩ, sau khi hoàn thành đoạn mã việc lớp OracleConnection. Bây giờ bạn đã có thể kết nối với cơ sở dữ liệu Oracle.

“Nhưng,” Giám đốc điều hành la lên ,”làm thế nào để kết với máy chủ cơ sở dữ liệu Microsof SQL Server?”

“Được”, bạn nói “Bình tĩnh, để tôi suy nghĩ một lát”. Bạn ra khỏi phòng để ăn trưa và sau đó quay lại tìm giám đốc và ban quản trị. Mọi người nóng lòng chờ đợi và hỏi “Mọi việc đã xong chưa?”

Bạn trở lại làm việc và tạo ra một lớp mới dùng để kết nối cơ sở dữ liệu, lớp SQLServerConnection.

```
Connection connection = new SqlServerConnection();
```

“Tốt lắm” Vị giám đốc nói. “Umh, vậy làm sao để kết nối với MySQL? Chúng ta muốn nó là kết nối mặc định”. “Woa”, bạn hơi bối rối. Tuy nhiên bạn vẫn làm thêm một kết nối với MySQL như sau:

```
Connection connection = new MySqlConnection();
```

Hiện tại bạn đã có ba loại kết nối cơ sở dữ liệu như sau: Oracle, SQL Server và MySQL. Vì vậy bạn chỉnh sửa mã nguồn cho phù hợp với các biến như “Oracle”, “SQL Server” hay bất cứ biến nào khác như sau:

```
Connection connection;

if (type.equals("Oracle")){
  connection= new OracleConnection();
}
else if (type.equals("SQL Server")){
  connection = new SqlServerConnection();
}
else {
  connection = new MySqlConnection();
}
```

Mọi việc đều ổn, bạn nghĩ. Tuy nhiên có tới 200 chỗ trong mã nguồn cần phải tạo kết nối cơ sở dữ liệu. Vì vậy đã tới lúc đưa đoạn mã này vào một phương thức riêng, phương thức createConnection, qua truyền cho nó loại kết nối mà bạn muốn như sau:

```
public Connection createConnection(String type)
{
  .
  .
  .
}
```

Phương thức có thể trả về loại kết nối mong muốn, tùy thuộc vào giá trị tham số truyền vào:

```
public Connection createConnection(String type)
{
  if (type.equals("Oracle")){
    return new OracleConnection();
  }
  else if (type.equals("SQL Server")){
    return new SqlServerConnection();
  }
  else {
    return new MySqlConnection();
  }
}
```

Tuyệt, bạn nghĩ. Không có gì khó khăn ở đây.

“Tin xấu” Vị giám đốc nói lớn trong khi chạy ào vào phòng làm việc của bạn. “Chúng ta cần phải chỉnh sửa lại mã nguồn để xử lý các kết nối an toàn cho tất cả máy chủ cơ sở dữ liệu. Hội đồng quản trị của khu vực Western yêu cầu như vậy”

Bạn đưa vị giám đốc ra khỏi phòng và ngồi suy nghĩ. Tất cả mã nguồn chắc phải chỉnh sửa lại. Phương thức mới createConnection, phần chính của mã nguồn, sẽ phải chỉnh sửa lại.

Trong chương II của quyển sách này. Bạn đã được biết dấu hiệu phải sử dụng mẫu thiết kế: “Đó là tách rời phần mã nguồn dễ thay đổi nhất ra khỏi phần mã chính của bạn. Và cố gắng sử dụng lại những phần này càng nhiều càng tốt.”

Có lẽ đây là lúc nghĩ về việc tách rời phần mã nguồn dễ thay đổi ra khỏi chương trình chính, phần tạo kết nối cơ sở dữ liệu connection, và đóng gói nó vào một đối tượng. Và

đối tượng đó chính là mẫu nhà máy Factory. Đối tượng là một nhà máy, được viết trong mã nguồn, nhằm tạo ra các đối tượng kết nối connection.

Vì sao bạn nghĩ tới mẫu thiết kế nhà máy Factory. Đây là những gợi ý:

- Bạn sử dụng toán tử new để tạo đối tượng OracleConnection
- Sau đó lại sử dụng tiếp toán tử new để tạo đối tượng SQLServerConnection, và sau đó là MySQLConnection. Nói cách khác, bạn đã sử dụng toán tử new để tạo nhiều đối tượng thuộc các lớp khác nhau, điều này làm mã nguồn của bạn trở nên lớn hơn và bạn buộc phải lặp lại điều này nhiều lần trong toàn bộ mã nguồn.
- Sau đó bạn đưa đoạn mã đó vào trong một phương thức
- Bởi vì yêu cầu vẫn còn có thể thay đổi nhanh chóng, nên cách tốt nhất là đóng gói chúng vào một đối tượng nhà máy factory. Theo cách làm này, bạn đã tách phần mã dễ thay đổi riêng biệt ra và giúp phần mã nguồn còn lại giữ vững nguyên tắc "đóng cho việc sửa đổi"

Chúng ta có thể nói rằng, toán tử new vẫn tốt trong mọi trường hợp, nhưng khi mã tạo dựng đối tượng bị liên tục thay đổi, ta nên nghĩ đến việc đóng gói chúng bằng mẫu thiết kế nhà máy factory.

XÂY DỰNG MẪU NHÀ MÁY FACTORY ĐẦU TIÊN

Nhiều lập trình viên biết cách thức mà đối tượng nhà máy factory làm việc. Họ nghĩ đơn giản rằng, bạn có một đối tượng làm nhiệm vụ tạo ra đối tượng khác. Đó là cách mà đối tượng factory thường được tạo ra và sử dụng, tuy nhiên nó còn làm được nhiều hơn thế. Chúng ta hãy nhìn vào cách thông thường khi tạo một đối tượng nhà máy factory trước, sau đó xem xét định nghĩa chính xác từ sách của GOF, theo định nghĩa mà đối tượng nhà máy factory sẽ có nhiều điểm khác, nhiều sự uyển chuyển hơn.

Tạo dựng đối tượng nhà máy Factory

Ví dụ đầu tiên, FirstFactory, sẽ làm việc theo cách hiểu thông thường nhất. Lớp FirstFactory đóng gói đối tượng xây dựng connection, và bạn truyền giá trị tham số theo đúng loại muốn tạo đó là "Oracle" hay "SQL Server" hay loại gì khác. Đây là cách bạn tạo một đối tượng sử dụng nhà máy factory :

```
FirstFactory factory;

factory = new FirstFactory("Oracle");
```

Bây giờ, bạn có thể sử dụng đối tượng nhà máy factory mới tạo này, để tạo đối tượng kết nối connection, bằng cách gọi phương thức tên createConnection như sau:

```
FirstFactory factory;

factory = new FirstFactory("Oracle");

Connection connection = factory.createConnection();
```

Vậy bạn đã tạo lớp nhà máy FirstFactory như thế nào? Hãy xem mã sau:

```
public class FirstFactory
{
  protected String type;

  public FirstFactory(String t)
  {
    type = t;
  }
    .
    .
    .
}
```

Đầu tiên bạn truyền kiểu kết nối vào phương thức khởi tạo của lớp FirstFactory.

Lớp FirstFactory chứa đựng một phương thức createConnection dùng để tạo ra một đối tượng kết nối connection thật sự. Đây là nơi bạn phải chỉnh sửa mã nguồn nhiều nhất tùy theo loại kết nối muốn tạo, mã như sau:

```
public class FirstFactory
{
  protected String type;

  public FirstFactory(String t)
  {
    type = t;
  }

  public Connection createConnection()
  {
    if (type.equals("Oracle")){
      return new OracleConnection();
    }
    else if (type.equals("SQL Server")){
      return new SqlServerConnection();
    }
    else {
      return new MySqlConnection();
    }
  }
}
```

Kết quả, bạn đã có một lớp nhà máy factory.

Tạo một lớp kết nối Connection trừu tượng

Hãy nhớ rằng một trong những mục tiêu của chúng ta khi viết mã, là làm sao việc thay đổi phần chính của mã nguồn càng ít càng tốt. Với mục tiêu đó, hãy nhìn đoạn mã sau làm việc, khi ta sử dụng một đối tượng connection được tạo bởi đối tượng nhà máy factory:

```
FirstFactory factory;

factory = new FirstFactory("Oracle");

Connection connection = factory.createConnection();

connection.setParams("username", "Steve");

connection.setParams("password", "Open the door!!!)";

connection.initialize();

connection.open();
        .
        .
```

Bạn có thể thấy rằng, đối tượng kết nối connection được tạo bởi nhà máy factory, được sử dụng khắp nơi trong mã nguồn. Để sử dụng cùng một đoạn mã cho tất cả các loại kết nối khác nhau (Oracle,MySQL…), đoạn mã cần phải được viết theo tính "đa hình", có nghĩa là tất cả các đối tượng connection, đều có cùng một giao diện interface, hay cùng kế thừa từ một lớp cơ sở. Theo cách đó, bạn có thể sử dụng cùng một biến cho mọi loại đối tượng kết nối.

Trong ví dụ, tôi tạo một lớp trừu tượng connection, để các lớp khác kế thừa nó. Lớp này gồm một phương thức khởi dựng, và một phương thức description (trả về mô tả của loại đối tượng). Mã như sau:

```
public abstract class Connection
{
  public Connection()
  {
  }

  public String description()
  {
    return "Generic";
  }
}
```

OK. Mọi việc có vẻ tốt đẹp. Bây giờ bạn đã tạo một lớp trừu tượng cơ sở cho các lớp kết nối khác kế thừa. Bạn cần phải kế thừa tất cả các đối tượng kết nối connection tạo ra từ lớp nhà máy factory.

Tạo lớp kế nối connection

Có ba lớp kết nối connection mà nhà máy Factory có thể tạo ra, phù hợp với loại kết nối mà Vị giám đốc mong muốn: OracleConnection , SqlServerConnection , MySQLConnection . Như chúng ta vừa nói, cần phải kế thừa từ lớp trừu tượng vừa tạo. Và mỗi loại trong chúng đều có phương thức decription trả về mô tả của từng loại kết nối một. Đây là mã nguồn của lớp OracleConnection:

```
public class OracleConnection extends Connection
{
  public OracleConnection()
  {
  }

  public String description()
  {
    return "Oracle";
  }
}
```

Đây là lớp SqlServerConnection, cũng kế thừa từ lớp trừu tượng Connection:

```
public class SqlServerConnection extends Connection
{
  public SqlServerConnection()
  {
  }

  public String description()
  {
    return "SQL Server";
  }
}
```

Và lớp MySqlConnection cũng tương tự:

```
public class MySqlConnection extends Connection
{
  public MySqlConnection()
  {
  }

  public String description()
  {
    return "MySQL";
  }
}
```

Tuyệt với. Mọi việc hoàn tất. Giờ là lúc thử nghiệm chúng. Đầu tiên ta tạo lớp nhà máy, truyền tham số khởi dựng là Oracle:

```
public class TestConnection
{
  public static void main(String args[])
  {
    FirstFactory factory;

    factory = new FirstFactory("Oracle");

    Connection connection = factory.createConnection();
        .
        .
        .
  }
}
```

Để kiểm tra lại đối tượng connection được tạo có phải là Oracle không, ta gọi phương thức description như sau:

```
public class TestConnection
{
  public static void main(String args[])
  {
    FirstFactory factory;

    factory = new FirstFactory("Oracle");

    Connection connection = factory.createConnection();

    System.out.println("You're connecting with " +
      connection.description());
  }
}
```

Kết quả bạn nhận được

```
You're connecting with Oracle
```

Không tồi. Đó là những gì bạn mong đợi.

Ghi nhớ: Theo sách GoF, mẫu thiết kế phương thức nhà máy Factory Method được định nghĩa "Định nghĩa một giao diện để tạo một đối tượng, nhưng cho phép các lớp con quyết định cách thức thể hiện nó. Phương thức nhà máy Factory cho phép một lớp trì hoãn việc hiện thực của nó qua các lớp con"

Điểm mấu chốt ở đây là phần "để lớp con quyết định". Cho tới bây giờ, lớp nhà máy Factory mà bạn vừa tạo, vẫn chưa cho phép các lớp con quyết định cách thể hiện, trừ việc cho kế thừa và ghi đè lại phương thức của lớp Connection cơ sở.

Mẫu thiết kế phương thức nhà máy Factory Method của GoF đem đến cho bạn khả năng uyển chuyển hơn phương pháp truyền thống rất nhiều. Cách làm của GoF là: bạn định nghĩa cách phương thức nhà máy Factory làm việc, và cho phép các lớp con hiện thực implement một nhà máy factory thật sự.

Chúng ta đã nói rằng, Hội đồng quản trị khu vực Western bất ngờ gọi điện và yêu cầu họ không thích lớp nhà máy FirstFactory, họ muốn có thể tạo ra các kết nối bảo mật đến máy chủ cơ sở dữ liệu, không chỉ là một kết nối thông thường. Điều này có nghĩa là họ phải viết lại lớp nhà máy FirstFactory mỗi khi bạn thay đổi nó, để họ có thể tạo ra một kết nối bảo mật.

Đây là vấn đề của các lập trình viên. Mỗi khi bạn cập nhật lại lớp FirstFactory, các lập trình viên khác phải viết lại mã của họ để thích hợp với yêu cầu của họ. Họ đang gọi và yêu cầu rằng họ muốn kiểm soát được quá trình nhiều hơn.

Tốt thôi, bạn nói. Đó chính là vấn đề mẫu thiết kế Factory áp dụng, giao quyền kiểm soát cho các lớp con. Để thấy cách mẫu này hoạt động, bạn thay đổi cách tạo đối tượng kết nối connection, sử dụng kỹ thuật của GoF, bạn sẽ làm cho khu vực Western của công ty MegaGigaCo hài lòng.

Gợi ý: Bạn vẫn còn băn khoăn về cách sử dụng của mẫu nhà máy Factory của GoF? Mẫu Factory được sử dụng khi bạn muốn chuyển giao toàn bộ quyền điều khiển các lớp con cho các lập trình viên khác.

TẠO MỘT NHÀ MÁY FACTORY THEO CÁCH CỦA GoF

Làm cách nào để "cho phép các lớp con toàn quyền hiện thực cách lớp con thể hiện" khi tạo một đối tượng nhà máy factory. Cách mà bạn phải làm là định nghĩa lớp nhà máy factory như là một lớp trừu tượng abstract hay giao diện interface, và để cho các lớp con hiện thực implement nó.

Nói cách khác, bạn tạo ra một khung sườn cho lớp nhà máy Factory tại trụ sở của MegaGigaCo, và việc hiện thực lớp này sẽ do các lớp con đảm nhiệm.

Tạo lớp nhà máy trừu tượng factory

Việc tạo lớp trừu tượng factory rất dễ dàng. Lớp này được gọi là ConnectionFactory

```
public abstract class ConnectionFactory
{
        .
        .
        .
}
```

Bên cạnh một phương thức khởi dựng rỗng, phương thức quan trọng nhất ở đây là phương thức nhà máy createConnection. Ta phải làm cho phương thức mang tính trừu tượng, để các lớp con hiện thực nó. Phương thức này nhận một đối số, đó là loại kết nối cần tạo:

```
public abstract class ConnectionFactory
{
  public ConnectionFactory()
  {
  }

  protected abstract Connection createConnection(String type);
}
```

Và đó là tất cả những gì bạn cần. Sự đặc tả cho đối tượng nhà máy factory. Bây giờ khu vựa Western sẽ hài lòng vì họ có thể hiện thực một đối tượng nhà máy cụ thể thích hợp với họ từ lớp trừu tượng trên.

Tạo một lớp nhà máy factory cụ thể

Bạn đã bay tới khu vực Western của công ty MegaGigaCo, để giúp họ xử lý vấn đề tạo đối tượng. Bạn giải thích ”Tôi hiểu rằng các anh muốn được quyền điều khiển nhiều hơn đối với các đối tượng kết nối”

“Vâng” các lập trình viên của Western nói. “Chúng tôi muốn có thể làm việc với các kết nối bảo mật. Chúng tôi đã tạo một vài lớp mới, lớp SecureOracleConnection, SecureSqlServerConnection và SecureMySqlConnection để tạo ra các kết nối bảo mật.

“OK” bạn nói. “tất cả những gì các bạn phải làm là mở rộng lớp trừu tượng mới của tôi, tên là ConnectionFactory khi các bạn muốn tạo đối tượng nhà máy factory cho các bạn. Hãy chắc chắn là các bạn sẽ hiện thực phương thức createConnection. Sau đó bạn có

thể tùy ý viết mã cho phương thức createConnection để tạo đối tượng theo đúng cách bảo mật mà bạn muốn”

Các lập trình viên của Western nói. “Wa, thật dễ dàng. Chúng tôi sẽ tạo lớp factory mới với tên SecureFactory, và nó sẽ kế thừa từ ConnectionFactory như sau:

```
public class SecureFactory extends ConnectionFactory
{
     .
     .
     .
}
```

“Tiếp theo,” các lập viên của khu vực Western nói “Chúng chỉ cần hiện thực lớp createConnection mà lớp trừu tượng ConnectionFactory yêu cầu:

```
public class SecureFactory extends ConnectionFactory
{
  public Connection createConnection(String type)
  {
       .
       .
       .
  }
}
```

“Cuối cùng” các lập trình viên nói “ chúng ta chỉ cần tạo các đối tượng từ các lớp vừa tạo , lớp SecureOracleConnection , SecureSqlServerConnection và SecureMySqlConnection , tùy thuộc vào kiểu dữ liệu được truyền vào hàm createConnection:

```
public class SecureFactory extends ConnectionFactory
{
  public Connection createConnection(String type)
  {
    if (type.equals("Oracle")){
      return new SecureOracleConnection();
    }
    else if (type.equals("SQL Server")){
      return new SecureSqlServerConnection();
    }
    else {
      return new SecureMySqlConnection();
    }
  }
}
```

“Thật đơn giản” Họ nói.

Sự khác biệt giữa cách tạo mẫu nhà máy factory thông thường và cách của GoF là cách của GoF chỉ đặc tả lớp nhà máy factory và để cho các lớp con xử lý nội dung chi tiết.

Tạo các lớp kết nối bảo mật

Để hiểu rõ cách thức GoF tạo mẫu factory, bạn cần tạo các lớp cụ thể cho đối tượng nhà máy connect mới, lớp SecureOracleConnection, SecureSqlServerConnection và lớp SecureMySqlConnection. Thật dễ dàng để tạo chúng ,bắt đầu từ lớp SecureOracleConnection, với hàm description trả về văn bản "Oracle Secure":

```
public class SecureOracleConnection extends Connection
{
  public SecureOracleConnection()
  {
  }

  public String description()
  {
    return "Oracle secure";
  }
}
```

Tiếp theo là lớp SecureSqlServerConnection, với hàm description trả về văn bản "SQL Server Secure"

```
public class SecureSqlServerConnection extends Connection
{
  public SecureSqlServerConnection()
  {
  }

  public String description()
  {
    return "SQL Server secure";
  }
}
```

Và lớp SecureMySqlConnection, với hàm description trả về văn bản "MySql Secure":

```
public class SecureMySqlConnection extends Connection
{
  public SecureMySqlConnection()
  {
  }

  public String description()
  {
    return "MySQL secure";
  }
}
```

Vậy là đã hoàn tất phần mã nguồn. Giờ là lúc cho chương trình chạy.

Thực thi chương trình

Để kiểm tra mã nguồn, hãy tạo đối tượng SecureFactory và sử dụng nó để tạo đối tượng SecureOracleConnection. Mã như sau:

```
public class TestFactory
{
  public static void main(String args[])
  {
    SecureFactory factory;

    factory = new SecureFactory();
```

Tất cả những gì bạn cần phải làm là sử dụng hàm createConnection của đối tượng nhà máy factory để tạo các kết nối an toàn. Mã như sau:

```
public class TestFactory
{
  public static void main(String args[])
  {
    SecureFactory factory;

    factory = new SecureFactory();

    Connection connection = factory.createConnection("Oracle");

    System.out.println("You're connecting with " +
      connection.description());
  }
}
```

Khi chạy chương trình, đúng như mong đợi, bạn nhận được văn bản sau, chứng tỏ rằng bạn đang sử dụng một kết nối Oracle bảo mật:

```
You're connecting with Oracle secure
```

Đây cũng là kết quả mà bạn nhận được từ ví dụ FirstFactory mà chúng ta đã nói trong phần trước, ngoại trừ một điều là bạn cho phép khu vựa Western tự mình hiện thực loại nhà máy factory mà họ mong muốn. Bạn đặc tả một lớp nhà máy bằng cách tạo ra một lớp trừu tượng hay một giao diện interface để các lớp con sử dụng, và người khác sẽ tự mình quyết định lớp đó thực hiện như thế nào. Không còn việc sử dụng một đối tượng nhà máy cụ thể, nay tập hợp các lớp con quyết định việc thể hiện chúng như thế nào.

DP4Dummies – Chương 4: Observer, Chain of Responsibility

Chương 4: "Chuyện gì đang diễn ra" mẫu Observer và mẫu Chain of Responsibility

Trong chương này, chúng ta sẽ nói về:

- Sử dụng mẫu Observer (Mẫu quan sát)
- Tạo một đối tượng quan sát Observer và đối tượng bị quan sát Subject
- Sử dụng lớp Observer Java
- Sử dụng giao diện Observer Java
- Sử dụng mẫu Chain of Responsibility

Ông chủ bước vào văn phòng và nói: "Ai đó đang chỉnh sửa dữ liệu trên máy chủ, sao tôi không được thông báo?"

"Bằng cách nào?" Bạn hỏi lại: "Ông muốn chúng tôi thông báo khi có bất kỳ sự thay đổi nào trên máy chủ à?" Bạn biết ông chủ vừa mới tiếp cận hệ thống này, nhưng không ngờ ông chủ lại hỏi những câu kì lạ vậy.

"Đúng vậy". Ông chủ nhấn mạnh. "Tôi muốn được thông báo khi có bất kì sự thay đổi nào trên dữ liệu ở máy chủ. Tôi muốn biết việc gì đang diễn ra quanh đây!"

"Ý ông muốn nói là gởi ông một báo cáo?"

"Đúng"

“Ok” Bạn nói “Tôi có một ý tốt hơn. Ông nghĩ sao khi tôi sử dụng mẫu thiết kế Observer (Người quan sát), và đăng ký ông với máy chủ dữ liệu như là một người quan sát dữ liệu Database Observer.

“Là sao?” Ông chủ hỏi.

“Ông sẽ được thông báo bất cứ khi nào có sự thay đổi dữ liệu trên máy chủ.” Bạn nói “Không cần báo cáo gì cả. Tất cả đều được viết tự động trong mã nguồn”

“Đó là tất cả những gì tôi muốn,” Ông chủ nói và đi khỏi.

Bạn mỉm cười với chính mình và tự hỏi. Không biết khuôn mặt ông chủ sẽ thế nào khi hàng ngày nhận được 200,000 thông báo về sự thay đổi dữ liệu ở máy chủ. Thế nhưng, với mẫu Observer, việc viết mã sẽ không có khó khăn gì.

Chương này nói về một đối tượng cần quan sát. Khi đối tượng có thay đổi, nó sẽ thông báo cho tất cả các Observer (đối tượng quan sát) những thay đổi đó. Chúng ta nói về hai mẫu, mẫu Observer và mẫu Chain of Responsibility.

Mẫu Observer cho phép nhiều đối tượng quan sát Observer nhận được thông báo, khi đối tượng bị quan sát có thay đổi. Từng Observer phải đăng ký với chủ thể, khi chủ thể thay đổi, nó sẽ thông báo tới tất cả các Observer. Các Observer được thông báo cùng lúc.

Mẫu Chain of Responsibility tương tự vậy, nhưng các Observer được thông báo theo thứ tự trước sau. Hết Observer này tới Observer khác, giúp cho từng Observer xử lý thông báo của theo cách của riêng mình.

Thông báo cho Observer với mẫu Observer.

Ông chủ muốn được báo cáo mỗi khi dữ liệu trên máy chủ có thay đổi. Bạn cũng muốn ghi nhận lại sự thay đổi này. Người thực hiện việc thay đổi cũng muốn biết việc thay đổi có thành công hay không?

Vì vậy, bạn cần có một tập hợp nhiều Observer để quan sát những gì đang diễn ra. Điều đó đúng với tên gọi của mẫu là “Người quan sát”. Cho phép đối tượng bị quan sát (máy chủ), thông báo tới các đối tượng quan sát (ông chủ, người sử dụng…) những gì đang diễn ra.

Mẫu thiết kế Observer nói về việc gởi thông báo cập nhật tới các đối tượng quan sát. Bạn có thể thêm hoặc xóa 1 đối tượng quan sát Observer trong khi hệ thống thực thi. Khi có sự thay đổi diễn ra, tất cả các Observer đều nhận được thông báo.

Theo sách của GOF (Design Patterns: Elements of Reusable Object-Oriented Software, xuất bản năm 1995) định nghĩa mẫu Observer như sau: "Xây dựng mối quan hệ một nhiều giữa các đối tượng, và khi đối tượng chủ thay đổi, tất cả các đối tượng phụ thuộc sẽ được thông báo một cách tự động"

Đây là cách thức mẫu Observer hoạt động. Một observer đăng ký với chủ thể như hình dưới

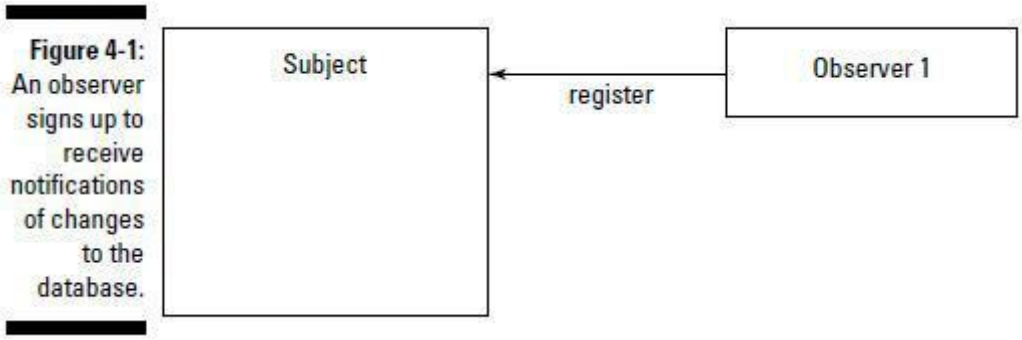


Figure 4-1: An observer signs up to receive notifications of changes to the database.

Chủ thể lưu lại thông tin của Observer này. Sau đó một observer khác, Observer 2 cũng đăng ký với chủ thể như hình dưới:

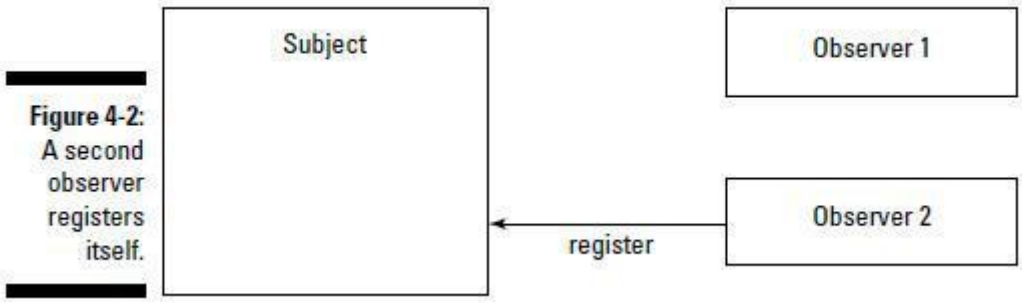


Figure 4-2: A second observer registers itself.

Vào thời điểm này, chủ thể (máy chủ dữ liệu), đang lưu trữ thông tin của 2 Observer. Khi có sự thay đổi trên máy chủ (dữ liệu bị thay đổi…), nó sẽ tự động gởi thông báo cho 2 Observer như hình dưới:

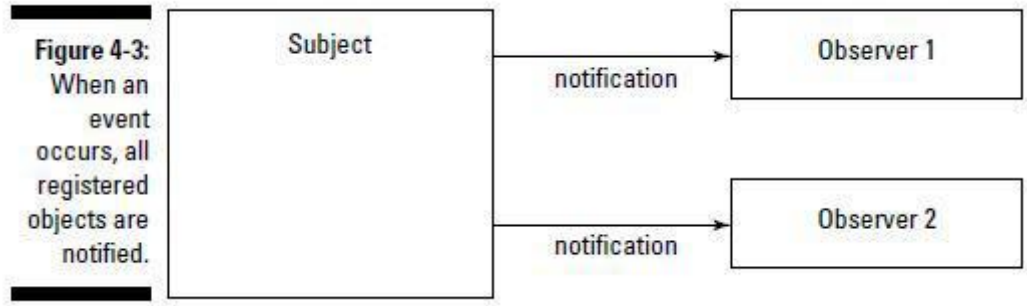


Figure 4-3: When an event occurs, all registered objects are notified.

Đối tượng quan sát Observer (ông chủ, người quản trị…) có thể gỡ đăng ký ra khỏi chủ thể (máy chủ) bất cứ lúc nào nó muốn. Lúc đó nó sẽ không nhận được thông báo từ chủ thể nữa. Theo như hình sau:

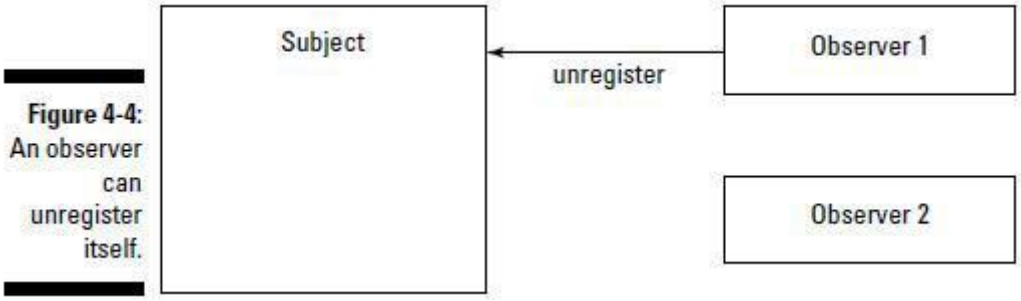


Figure 4-4: An observer can unregister itself.

Bây giờ Observer 1 sẽ không nhận được bất kỳ thông báo nào từ máy chủ nữa. Khi máy chủ có sự thay đổi, nó chỉ gởi thông báo cho Observer 2 như hình sau:

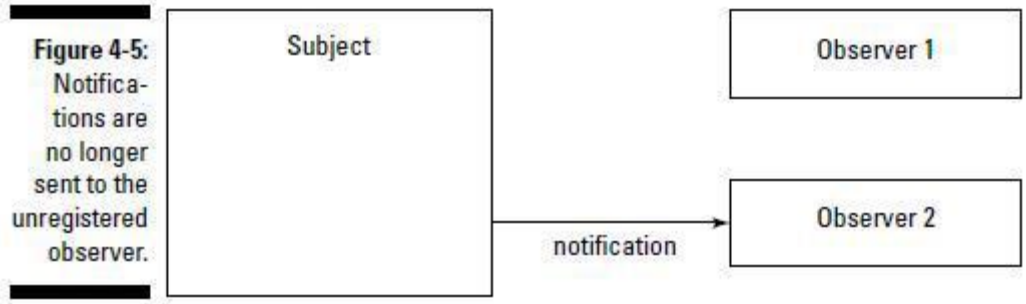


Figure 4-5: Notifications are no longer sent to the unregistered observer.

Tuy nhiên, Observer 1 có thể tái đăng ký với máy chủ bất cứ khi nào cần thiết. Khi đó nó lại được đưa vào danh sách nhận thông báo.

Cách tạo một giao diện chủ thể (Subject Interface)

(ND: Interface là một khái niệm trừu tượng trong lập trình hướng đối tượng, là một khái niệm tổng quát hơn, thô sơ hơn của một lớp. Nó định nghĩa hình dáng của một lớp. Còn được coi như một hợp đồng. Thông thường nó quy định lớp nào hiện thức interface này phải có những thuộc tính nào, phương thức nào…)

Khi bạn hiện thực một mẫu thiết kế, phương pháp tốt nhất thường là bắt đầu từ việc tạo một interface. Điều này đảm bảo cho những lớp bạn định tạo luôn phù hợp với mẫu thiết kế. Đặc biệt là trong trường hợp bạn phải tạo nhiều lớp khác nhau, việc tạo một interface gần như bắt buộc. Hiện thực(implement) một interface giúp cho mã của bạn sáng sủa và dễ hiểu hơn.

Khi viết mã cho mẫu Observer, bạn tạo một interface hay abstract class cho đối tượng Observer. Điều này giúp cho việc tạo nhiều Observer luôn nhất quán với nhau.

(ND: Abstract class, còn gọi là lớp trừu tượng, là một khái niệm trong lập trình hướng đối tượng, nó nằm ở giữa interface và class. Nó chi tiết hơn Interface, nhưng tổng quát cao hơn Class. Interface chỉ định nghĩa hình dáng của lớp, Abstract class vừa định nghĩa hình dáng một lớp, vừa chứa đựng cả phần hiện thực của lớp)

Trong ví dụ sau, tôi sẽ tạo một interface cho chủ thể Subject (Đối tượng cần quan sát), interface này liệt kê các phương thức của Subject cần có. Subject này cần phải có ít nhất 3 hàm. Hàm registerObserver để đăng ký Observer, hàm removeObserver để gỡ đăng ký 1 Observer, và hàm notifyObservers để thông báo cho tất cả các Observer biết khi có sự thay đổi trên Subject.

```
public interface Subject
{
  public void registerObserver(Observer o);
  public void removeObserver(Observer o);
  public void notifyObservers();
}
```

Interface trên liệt kê các hàm mà chủ thể (máy chủ dữ liệu), cần phải có. Tiếp theo, ta sẽ định nghĩa 1 interface cho Observer.

Tạo một giao diện quan sát Observer Interface

Việc này tương đối dễ dàng, interface cho Observer chỉ cần 1 hàm, hàm này sẽ thực hiện khi Observer nhận được thông báo từ máy chủ. Tôi đặt tên cho nó là hàm cập nhật update. Trong ví dụ này, bạn chuyển những thay đổi của máy chủ (như xóa, sửa, tạo …) và tên record đã thay đổi vào hàm update này.

```
public interface Observer
{
  public void update(String operation, String record);
}
```

(ND: tôi giữ nguyên chữ record trong văn bản gốc. Thông thường các database bao gồm nhiều bảng, mỗi bảng bao gồm nhiều hàng, nhiều cột. Một record chính là 1 hàng trong bảng. Khi lập trình ta thường ánh xạ 1 bảng thành 1 lớp. 1 hàng trong bảng, chính là 1 đối tượng. Khi ta thay đổi đối tượng này, chính là đang thay đổi 1 record. Tôi luôn phân vân khi dịch một từ chuyên ngành tin học sang tiếng việt, vì thực sự chúng ta chưa xây dựng một từ điển chuẩn cho tin học, điều này cần phải huy động lực lượng quốc gia. Tôi thấy rằng để nguyên từ tiếng anh thì dễ hiểu và chuẩn mực. Bạn nghĩ sao khi tôi dịch từ implement thành hiện thực, instance thành thể hiện, record thành bản ghi…Có gì đó không ổn, nên có lúc tôi giữ nguyên từ tiếng anh, có lúc tôi chuyển ngữ, mong rằng các đọc giả thông cảm bỏ qua)

Khi observer hiện thực hàm update này, máy chủ có thể chuyển tới observer tên record và những thay đổi trên record đó.

Mọi việc bắt đầu tốt đẹp. Tiếp theo tôi sẽ tạo một đối tượng Database(máy chủ dữ liệu), làm đối tượng bị quan sát, nó sẽ lưu trữ thông tin về các observer (đối tượng quan sát), và thông báo cho observer biết khi có gì thay đổi diễn ra.

Tạo một đối tượng cần quan sát – chủ thể Subject

Chủ thể cho phép một Observer đăng ký, và phải thông báo cho Observer biết khi có sự kiện xảy ra. Theo như giao diện Subject Interface ở trên, có 3 hàm mà Subject cần phải hiện thực là registerObserver, removeObserver, notifyObserver.

DP4Dummies – Chương 5: Singleton, Flyweight

CHƯƠNG V: TỪ MỘT CHO TỚI NHIỀU – MẪU DUY NHẤT SINGLETON VÀ MẪU FLYWEIGHT

Trong chương này, chúng ta sẽ đi qua các nội dung sau:

- Sử dụng mẫu duy nhất Singleton
- Ví dụ về Singleton
- Đồng bộ hóa để loại bỏ các vấn đề rắc rối trong đa luồng.
- Một cách tốt hơn để xử lý đa luồng
- Sử dụng mẫu "hạng ruồi" flyweight

Là một nhà tư vấn được trả lương hậu hĩnh tại MegaGigaco, bạn phải xử lý các sự cố về hiệu năng hệ thống. "Hệ thống hình như ngày càng chậm chạp hơn." Các lập trình viên nói:

"Hmm," bạn nói, "Tôi lưu ý các bạn rằng chúng ta đang có một đối tượng dữ liệu có kích thước khá lớn, khoảng 20Mb"

"Vâng", họ nói.

"Cùng một thời điểm, các bạn sử dụng bao nhiêu đối tượng này?"

"Khoảng 219", các lập trình viên nói

"Trời, vậy các bạn sử dụng 219 đối tượng 20Mb trong lúc chương trình hoạt động?" Bạn nói. "Chẳng lẽ không ai thấy được vấn đề ở đây à?"

"Không", họ đồng thanh nói.

Bạn nói với họ "Các bạn sử dụng quá nhiều tài nguyên hệ thống. Các bạn có hàng trăm đối tượng to lớn mà máy tính phải xử lý. Các bạn có thật sự cần tất cả chúng?"

"Vâng…" họ nói.

"Tôi nghĩ là không," bạn nói. "Tôi sẽ sửa chữa vấn đề này bằng cách sử dụng mẫu duy nhất Singleton."

Chương này nói về việc kiểm soát số lượng đối tượng mà bạn phải tạo ra trong mã nguồn của mình. Có hai mẫu thiết kế đặc biệt giúp ích cho bạn: mẫu duy nhất Singleton và mẫu "hạng ruồi" flyweight.

Với mẫu duy nhất Singleton, bạn luôn đảm bảo rằng chỉ có duy nhất một đối tượng cho một lớp cụ thể trong suốt ứng dụng. Với mẫu "hạng ruồi" flyweight, bạn cũng có duy nhất một đối tượng cho một lớp, nhưng khi nhìn vào mã của bạn, ta có thể thấy giống như đang có nhiều đối tượng vậy. Đây là một thủ thuậ t khéo léo.

Tạo một đối tượng duy nhất với mẫu duy nhất Singleton.

Tôi bắt đầu với mẫu Singleton và xử lý rắc rối mà lập trình viên MegaGigaCo gặp phải. Họ muốn chắn chắc rằng chỉ tạo duy nhất một đối tượng cho một lớp cụ thể mặc cho người khác có cố gắng tạo bao nhiêu đối tượng đi nữa.

Các lập trình viên đang tạo ra hàng trăm đối tượng Database trong mã nguồn, và rắc rối là từng đối tượng này có kích thước rất lớn. Đâu là giải pháp? Mẫu duy nhất Singeton là câu trả lời.

Mẫu duy nhất Singleton chắc chắn rằng bạn có thể khởi tạo chỉ duy nhất một đối tượng cho một lớp. Nếu bạn không sử dụng mẫu thiết kế này, toán tử new như thường sử dụng, sẽ tạo ra liên tiếp nhiều đối tượng mới như sau:

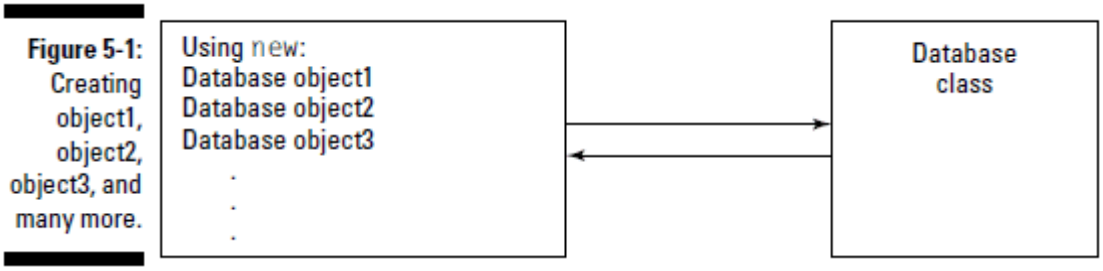


Figure 5-1: Creating object1, object2, object3, and many more.

Ghi nhớ: Để chắc chắn rằng bạn chỉ có duy nhất một đối tượng, mặc cho người khác có hiện thực bao nhiêu phiên bản đi nữa, hãy sử dụng mẫu duy nhất Singleton. Cuốn sách GoF nói rằng, mẫu Singleton "Đảm bảo rằng một lớp chỉ có duy nhất một thể hiện và cung cấp một biến toàn cục để truy cập nó"

Bạn sử dụng mẫu Singleton khi bạn muốn hạn chế việc sử dụng tài nguyên (thay vì việc tạo không hạn chế số lượng đối tượng) hoặc khi bạn cần phải xử lý một đối tượng nhạy

cảm, mà dữ liệu của nó không thể chia sẻ cho mọi thể hiện, như registry của Windows chẳng hạn.

Gợi ý: Ngoài đối tượng bản ghi registry, bạn có thể sử dụng mẫu Singleton khi bạn muốn hạn chế số lượng các thể hiện được tạo bởi vì bạn muốn chia sẻ dữ liệu của các đối tượng này. Ví dụ như khi bạn có một đối tượng cửa sổ window hay hộp thoại dialog, cần phải hiện thị và thay đổi dữ liệu, bạn sẽ không muốn tạo nhiều thể hiện của đối tượng này, vì bạn sẽ bị bối rối trong việc phải truy cập dữ liệu của thể hiện nào.

Việc tạo một đối tượng duy nhất cũng rất quan trọng khi bạn sử dụng đa luồng và khi bạn không muốn sự đụng độ dữ liệu xảy ra. Ví dụ bạn đang làm việc với một đối tượng cơ sở dữ liệu, và các thể hiện khác cũng làm việc trên cùng cơ sở dữ liệu đó, việc đụng độ có thể gây ra các vấn đề nghiêm trọng. Tôi sẽ thảo luận cách làm việc với mẫu Singleton và đa luồng trong chương này.

Bất cứ khi nào bạn thật sự cần duy nhất một thể hiện của một lớp, hãy nghĩ tới mẫu Singleton (thay vì dùng toán tử new).

Tạo lớp cơ sở dữ liệu Database dựa trên kiểu Singleton

Giờ là lúc bạn bắt tay vào viết mã nguồn. Bạn sẽ tạo một lớp tên Database mà các lập trình viên trong công ty sẽ sử dụng. Lớp này có một hàm khởi dựng đơn giản, như mã sau:

```
public class Database
{
  private int record;
  private String name;

  public Database(String n)
  {
    name = n;
    record = 0;
  }
    .
    .
    .
}
```

Bạn cần phải thêm vào hai hàm editRecord, cho phép bạn chỉnh sửa một bản ghi, và hàm getName, trả về tên gọi của Database.

```
public class Database
{
  private int record;
  private String name;

  public Database(String n)
  {
    name = n;
    record = 0;
  }

  public void editRecord(String operation)
  {
    System.out.println("Performing a " + operation +
      " operation on record " + record +
      " in database " + name);
  }

  public String getName()
  {
    return name;
  }
}
```

Tới giờ mọi việc vẫn tốt đẹp. Bất cứ khi nào bạn tạo một đối tượng bằng toán tử new, một đối tượng mới sẽ được tạo ra. Nếu bạn tạo 3 database, bạn sẽ có 3 đối tượng như sau:

```
Database dataOne = new Database("Products");
        .
        .
        .
Database dataTwo = new Database("Products Also");
        .
        .
        .
Database dataThree = new Database("Products Again");
        .
        .
        .
```

Làm sao để bạn có thể tránh việc tạo một đối tượng mới khi sử dụng toán tử new? Đây là một giải pháp – làm cho hàm khởi dụng từ toàn cục (public) trở thành cục bộ (private)

```
  private Database(String n)
  {
    name = n;
    record = 0;
  }
```

Điều này ngăn cản mọi người sử dụng hàm khởi dựng, ngoài trừ chính trong lớp này gọi tới. Nhưng đợi một chút, có gì không ổn ở đây? Ai ở trên trái đất này lại cần có một hàm khởi dựng riêng tư vậy? Làm sao bạn có thể tạo một đối tượng khi bạn không thể gọi hàm khởi tạo nó?

Bạn đã làm cho hàm khởi dựng trở nên riêng tư và cách duy nhất để phần còn lại của thế giới khởi tạo đối tượng đó là thêm vào một hàm tạo đối tượng, và gọi nó khi bạn chắn chắn muốn tạo một đối tượng duy nhất cho lớp này.

Hãy xem đoạn mã sau:

```
public class Database
{
  private int record;
  private String name;
  private Database(String n)
  {
    name = n;
    record = 0;
  }
    .
    .
    .
```

OK. Đầu tiên bạn ngăn chặn việc khởi tạo bằng toán tử new. Và bây giờ cách duy nhất là tạo một hàm để gọi việc khởi tạo đối tượng, thông thường hàm này có tên getInstance (hay createInstance hoặc một cái tên cụ thể như createDatabase cũng được). Chú ý rằng hàm này được gán phạm vi công cộng và là một phương thức tĩnh để bạn có thể truy cập tới nó thông qua tên lớp (ví dụ như Database.getInstance())

(ND: public và static là hai khái niệm trong OOP. Public giúp hàm có thể được sử dụng từ bất kì lớp khác, static giúp ta có thể sử dụng hàm trực tiếp từ tên lớp, không cần thông qua một đối tượng lớp.)

```
public class Database
{
  private int record;
  private String name;

  private Database(String n)
  {
    name = n;
    record = 0;
  }

  public static Database getInstance(String n)
  {
  }
    .
    .
    .
}
```

Hàm này sẽ trả về một đối tượng Database , nhưng hàm chỉ hoạt động khi có ít nhất một đối tượng đã tồn tại. Vì thế đầu tiên ta cần kiểm tra đối tượng này , tôi gọi nó là singleObject , xem nó đã tồn tại chưa ? Nếu chưa, tôi sẽ tạo nó. Và sau đó trả giá trị nó về cho hàm.

```
public class Database
{
  private static Database singleObject;
  private int record;
  private String name;

  private Database(String n)
  {
    name = n;
    record = 0;
  }

  public static Database getInstance(String n)
  {
      if (singleObject == null){
        singleObject = new Database(n);
      }

      return singleObject;
  }
    .
    .
    .
}
```

Vấn đề đã được giải quyết. Bây giờ chỉ có duy nhất một đối tượng Database tồn tại trong cùng một thời điểm. (Vấn đề đa luồng ta sẽ giải quyết trong phần sau của chương). Việc gọi hàm getInstance sẽ cho ta một đối tượng như hình sau:

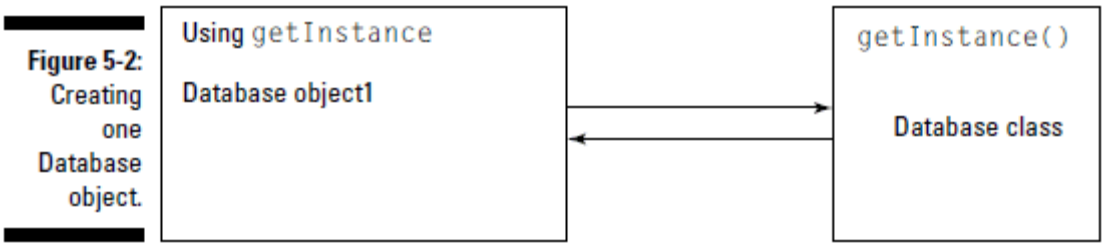


Figure 5-2: Creating one Database object.

Khi bạn gọi getInstance lần nữa, bạn sẽ nhận được cùng một đối tượng như lần đầu.

Không quan tâm đến việc bạn gọi bao nhiêu lần getInstance, bạn luôn nhận được cùng một đối tượng. Đó chính là cách bạn phải làm với mẫu singleton.

Chạy thử ví dụ với mẫu Singleton

Bắt đầu bằng việc tạo một đối tượng Database với tên là products, sau đó gọi hàm getName:

```
public class TestSingleton
{
  public static void main(String args[])
  {
    Database database;

    database = Database.getInstance("products");

    System.out.println("This is the " +
      database.getName() + " database.");
        .
        .
        .
  }
}
```

Sau đó bạn tiếp tục tạo một đối tượng Database với tên là employees, và gọi lại hàm getName để kiểm tra:

```
public class TestSingleton
{
  public static void main(String args[])
  {
    Database database;

    database = Database.getInstance("products");

    System.out.println("This is the " +
      database.getName() + " database.");

    database = Database.getInstance("employees");

    System.out.println("This is the " +
      database.getName() + " database.");
  }
}
```

Tuy nhiên đối tượng Database đã được tạo, vì vậy trong lần thứ hai, hàm getInstance vẫn trả về đối tượng Database cũ, và kết quả là bạn nhận được thông báo:

```
This is the products database.
This is the products database.
```

Quá rõ ràng. Bạn đã nhận được duy nhất một đối tượng cho dù đã thực hiện việc tạo hai lần. Cách thức bạn làm việc như sau: ngăn cản việc khởi tạo bằng toán tử new, và tạo một hàm mới để tạo đối tượng theo ý bạn. Đó chính là cách mẫu Singleton hoạt động.

Đừng quên vấn đề đa luồng

Hãy xem hàm getInstance trong ví dụ trên:

```
  public static Database getInstance(String n)
  {
      if (singleObject == null){
        singleObject = new Database(n);
      }

      return singleObject;
  }
```

Có một lỗ hổng tiềm tàng ở đây, tuy nhỏ nhưng là một lỗ hổng rõ ràng, đó là khi làm việc với đa luồng. Hãy nhớ rằng, bạn muốn đảm bảo rằng chỉ có duy nhất một đối tượng Database tồn tại. Nhưng khi bạn có nhiều luồng chương trình chạy cùng lúc, bạn sẽ gặp rắc rối. Cụ thể là, hãy chú ý đoạn mã kiểm tra sự tồn tại của đối tượng Database:

```
public static Database getInstance(String n)
{
    if (singleObject == null){
      singleObject = new Database(n);
    }

    return singleObject;
}
```

Nếu có hai luồng cùng thực hiện hàm kiểm tra này một lúc, hai luồng này đều thỏa điều kiện của hàm if (tức chưa có đối tượng nào được tạo), và điều này có nghĩa là cả hai luồng đều tạo ra một đối tượng Database.

Làm sao để chỉnh sửa chỗ này? Một cách dễ dàng là sử dụng từ khóa synchronized (đồng bộ) trong Java, xem đoạn mã sau:

```
public class DatabaseSynchronized
{
  private static DatabaseSynchronized singleObject;
  private int record;
  private String name;
  private DatabaseSynchronized(String n)
  {
    name = n;
    record = 0;
  }

  public static synchronized DatabaseSynchronized getInstance(String n)
  {
      if (singleObject == null){
        singleObject = new DatabaseSynchronized(n);
      }

      return singleObject;
  }

  public void editRecord(String operation)
  {
    System.out.println("Performing a " + operation +
      " operation on record " + record +
      " in database " + name);
  }

  public String getName()
  {
    return name;
  }
}
```

Sử dụng từ khóa synchronized sẽ khóa việc truy cập vào hàm getInstance, trong khi hàm getInstance được chạy. Bất cứ luồng nào muốn gọi hàm getInstance, đều phải đợi hàm này hoạt động xong. Sử dụng kỹ thuật đồng bộ hóa synchronized là cách dễ nhất để thực thi việc đơn luồng trong gọi hàm, và kỹ thuật này đã giải quyết được vấn đề đa luồng.

Chạy thử chương trình với giải pháp đồng bộ hóa:

Bởi vì việc gọi hàm getInstance đã được đồng bộ hóa, bạn có gọi hàm từ nhiều luồng khác nhau. Xem mã sau:

```
public class TestSingletonSynchronized implements Runable
{
  public static void main(String args[])
  {
    TestSingletonSynchronized t = new TestSingletonSynchronized();
  }

  public TestSingletonSynchronized()
  {
    DatabaseSynchronized database;

    database = DatabaseSynchronized.getInstance("products");
        .
        .
        .
}
```

Đoạn mã cũng cho phép chạy một tiến trình mới để cố gắng tạo mới một đối tượng DatabaseSynchronized:

```
public class TestSingletonSynchronized implements Runable
{
  Thread thread;

  public static void main(String args[])
  {
    TestSingletonSynchronized t = new TestSingletonSynchronized();
  }

  public TestSingletonSynchronized()
  {
    DatabaseSynchronized database;

    database = DatabaseSynchronized.getInstance("products");

    thread = new Thread(this, "second");
    thread.start();

    System.out.println("This is the " +
      database.getName() + " database.");
  }
    .
    .
    .
}
```

Tiến trình mới cố gắng tạo một đối tượng mới DatabaseSynchronized với tên employees.

```
public class TestSingletonSynchronized implements Runable
{
  Thread thread;

  public static void main(String args[])
  {
    TestSingletonSynchronized t = new TestSingletonSynchronized();
  }

  public TestSingletonSynchronized()
  {
    DatabaseSynchronized database;
    database = DatabaseSynchronized.getInstance("products");

    thread = new Thread(this, "second");
    thread.start();

    System.out.println("This is the " +
      database.getName() + " database.");
  }

    public void run()
    {
      DatabaseSynchronized database =
        DatabaseSynchronized.getInstance("employees");

      System.out.println("This is the " +
        database.getName() + " database.");
    }
}
```

Nhưng như bạn có thể nhìn thấy, khi chương trình thực thi, chỉ duy nhất một đối tượng DatabaseSynchronized tồn tại. Đó là đối tượng products.

```
This is the products database.
This is the products database.
```

Kể từ khi bạn sử dụng kỹ thuật đồng bộ hóa trên hàm getInstance, bạn không còn lo lắng về vấn đề đa luồng nữa. Chỉ duy nhất một luồng được gọi hàm getInstance. Nó ngăn chặn việc tạo đối tượng bằng một bức tường an toàn, việc kiểm tra ở trên cho thấy, nếu đối tượng muốn tạo đã tồn tại, hàm sẽ không tạo, ngược lại, sẽ tạo đối tượng cho lớp.

Thoạt nhìn, điều này thật tuyệt vời, đồng bộ hóa hàm getInstance và giải quyết được vấn đề đa luồng, bảo vệ mã nguồn chống lại việc xung đột khi có nhiều tiến trình cùng tạo một đối tượng.

Tuy nhiên, vẫn còn một câu hỏi, việc đồng bộ hóa đã giải quyết vấn đề, nhưng đó có phải là cách tốt nhất. Việc đồng bộ hóa gây tốn tài nguyên hệ thống, Java buộc phải theo dõi từng tiến trình để cho xử lý khi nào thì cho phép tiến trình truy cập hàm getInstance, khi nào thì không cho phép.

Đồng bộ hóa hàm getInstance đã làm việc, nhưng với chi phí đáng kể. Có cách nào tốt hơn để giải quyết vấn đề này?

Tối ưu hóa việc xử lý đa luồng

Vấn đề khi bạn cố gắng xử lý từ khóa synchronized là đoạn mã kiểm tra việc tạo đối tượng sẽ bị phá hỏng bởi các tiến trình khác. Một cách tốt hơn để làm việc này là đảm bảo rằng đoạn mã kiểm tra không còn quan trọng nữa.

“Mọi việc thế nào?”Các lập trình viên hỏi trong kinh ngạc. “Nếu bạn không kiểm tra việc tạo đối tượng, làm sao bạn có thể chắc chắn là bạn không tạo thêm một đối tượng mới?”

Bạn giải thích “Bằng cách loại bỏ toàn bộ mã tạo đối tượng ra khỏi hàm getInstance. Tôi sẽ viết lại mã để chắc chắn chỉ một đối tượng được tạo ra. Và đối tượng sẽ được tạo ra trước khi bất cứ luồng chương trình nào có thể nắm bắt được nó.

“Hmm”, các lập trình viên nói “Nghe có vẻ nó sẽ hoạt động”

Ý tưởng như sa:– Tạo đối tượng duy nhất mà bạn muốn ngay khi mã nguồn được nạp lần đầu tiên vào Java Virtual Machine (máy ảo Java, bộ máy biên dịch và thi hành Java). Không để hàm getInstance tạo đối tượng nữa. Chỉ cho phép hàm trả về đối tượng vừa tạo. Mã như sau:

```
public class DatabaseThreaded
{
  private static DatabaseThreaded singleObject =
    new DatabaseThreaded("products");
  private int record;
  private String name;

  private DatabaseThreaded(String n)
  {
    name = n;
    record = 0;
  }

        .
        .
        .
}
```

Tốt, giờ bạn đã tạo đối tượng duy nhất rồi. Tất cả những gì cần làm là cho hàm getInstance trả về đối tượng này

```
public class DatabaseThreaded
{
  private static DatabaseThreaded singleObject =
    new DatabaseThreaded("products");
  private int record;
  private String name;

  private DatabaseThreaded(String n)
  {
    name = n;
    record = 0;
  }

  public static synchronized DatabaseThreaded getInstance(String n)
  {
      return singleObject;
  }

  public void editRecord(String operation)
  {
    System.out.println("Performing a " + operation +
      " operation on record " + record +
      " in database " + name);
  }

  public String getName()
  {
    return name;
  }
}
```

Như các bạn có thể thấy, thật đơn giản, đối tượng singleton đã được tạo ra trước khi bất cứ luồng chương trình nào có thể nắm bắt được. Tuyệt vời.

Cách làm việc của giải pháp xử lý "tiền tiến trình"

Chương trình trên có hoạt động không? Như với giải pháp đồng bộ hóa, bạn có thể để phiên bản này làm việc bằng cách tạo đối tượng DatabaseThreaded bằng cách gọi hàm getInstance.

```
public class TestSingletonThreaded implements Runable
{
  public static void main(String args[])
  {
    TestSingletonThreaded t = new TestSingletonThreaded();
  }

  public TestSingletonThreaded()
  {
    DatabaseThreaded database;

    database = DatabaseThreaded.getInstance("products");
      .
      .
      .
}
```

Và bạn có thể sử dụng một tiến trình khác để cố gắng tạo ra một đối tượng DatabaseThreaded khác.

```
public class TestSingletonThreaded implements Runable
{
  Thread thread;

  public static void main(String args[])
  {
    TestSingletonThreaded t = new TestSingletonThreaded();
  }

  public TestSingletonThreaded()
  {
    DatabaseThreaded database;

    database = DatabaseThreaded.getInstance("products");

    thread = new Thread(this, "second");
    thread.start();

    System.out.println("This is the " +
      database.getName() + " database.");
  }

    public void run()
    {
      DatabaseThreaded database;

      database = DatabaseThreaded.getInstance("employees");

      System.out.println("This is the " +
        database.getName() + " database.");
    }
}
```

Và khi chương trình hoạt động. Bạn thấy rằng bạn đang nhận được cùng một đối tượng.

```
This is the products database.
This is the products database.
```

Đây là giải pháp tốt hơn việc đồng bộ hóa hàm getInstance. Không thể tạo nhiều hơn một đối tượng, nên sẽ tránh được sự xung đột giữa các luồng chương trình. Bạn đã gỡ bỏ mã nguồn đồng bộ hóa bằng cách đưa chúng ra hỏi hàm getInstance.

Chú ý: Nếu bạn sử dụng Java với phiên bản nhỏ hơn 1.2, có một rắc rối với trình thu dọn rác. Nếu không có một tham chiếu với đối tượng singleton, trình thu gom rác sẽ thu gom luôn đối tượng này. Lỗi này đã được sửa lại trên phiên bản 1.2

Có một vấn đề cần chú ý. Nếu bạn sử dụng bộ nạp đa lớp và sử dụng đối tượng singleton, bạn sẽ gặp lỗi. Bởi vì mỗi lớp sử dụng một không gian tên khác nhau, bạn có

thể đối mặt với việc tạo nhiều đối tượng singleton. Vì thế khi bạn sử dụng nhiều lớp, hãy chắc chắn rằng mã nguồn có kiểm tra đối chiếu giữa các lớp, để đảm bảo rằng chỉ có duy nhất một đối tượng singleton tồn tại trong một thời điểm.

Mẫu ”hạng ruồi” flyweight giúp cho một đối tượng trông giống nhiều đối tượng.

Mẫu singleton nói về việc tạo một đối tượng duy nhất. Có một mẫu thiết kế khác cũng hạn chế việc tạo đối tượng, nhưng lần này nó sẽ đem đến một cách thức khác trong việc viết mã. Đó là mẫu “hạng ruồi” flyweight.

Mẫu thiết kế này gọi là “hạng ruồi” flyweight nguyên do thay vì phải làm việc với nhiều đối tượng độc lập, to lớn, bạn giảm bớt kích thước chúng bằng việc tạo một tập hợp các đối tượng dùng chung nhỏ hơn, gọi là flyweight mà bạn có thể cài đặt vào lúc thực thi chương trình để chúng trông giống như những đối tượng lớn hơn. Mỗi đối tượng to lớn có thể tiêu tốn nhiều tài nguyên hệ thống, bằng cách tách những điểm giống nhau của các đối tượng này, và dựa trên việc cấu hình thời gian thực để mô phỏng lại các đối tượng lớn, bạn đã làm giảm bớt gánh nặng lên tài nguyên hệ thống.

Bạn có thể đem những phần riêng biệt ra khỏi mã nguồn của những đối tượng to lớn và tạo ra những đối tượng flyweight. Khi làm điều này, bạn đã chấm dứt việc sử dụng nhiều đối tượng có chung các đặc điểm, và giảm xuống việc chỉ sử dụng một đối tượng, có thể cài đặt khi chương trình thực thi, mô phỏng lại cả tập hợp các đối tượng to lớn ban đầu

Ghi nhớ: Sách GoF đã định nghĩa mẫu flyweight như sau: “Sử dụng việc chia sẻ để giúp cho việc xử lý các đối tượng lớn một cách hiệu quả” Họ cũng nói rằng: “Một mẫu flyweight là một đối tượng chia sẻ mà có thể sử dụng trong đồng thời nhiều ngữ cảnh. Mẫu flyweight hoạt động như một đối tượng độc lập trong mỗi thời điểm.

Đây là những gì diễn ra. Bạn bắt đầu với một tập hợp nhiều đối tượng to lớn trong mã nguồn. Bạn gỡ bỏ những phần dùng chung, đóng gói chúng vào một đối tượng chia sẻ, một flyweight, đối tượng này hoạt động như một khuôn mẫu. Đối tượng khuôn mẫu này có thể được cài đặt vào lúc thực thi chương trình bằng cách chuyển các đặc điểm dùng chung vào đối tượng flyweight để nó xuất hiện giống như một hay nhiều đối tượng lớn ban đầu. Bạn có thể thấy như hình vẽ sau:

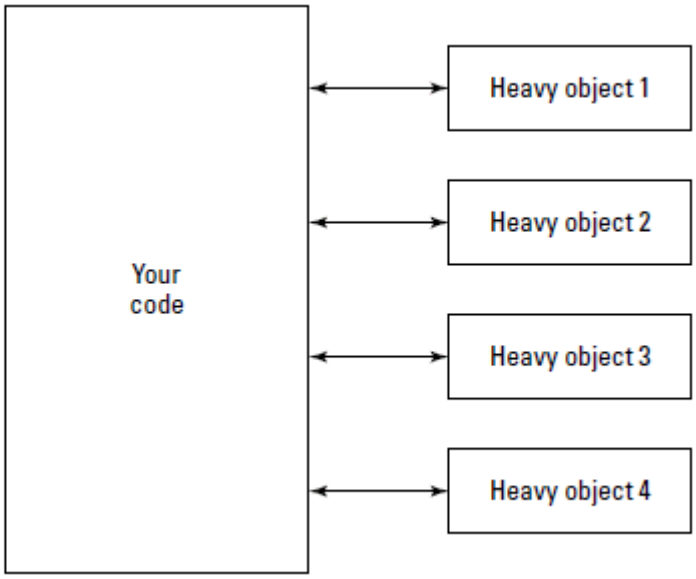


Figure 5-4: Code full of heavy, large objects.

Từ những đối tượng to lớn, bạn tạo một đối tượng nhỏ hơn gọi là flyweight (trong ví dụ này là một flyweight, tùy nhiên tùy thuộc vào ứng dụng mà bạn có thể có nhiều flyweight), mà bạn có thể cài đặt vào lúc chương trình hoạt động như hình sau:

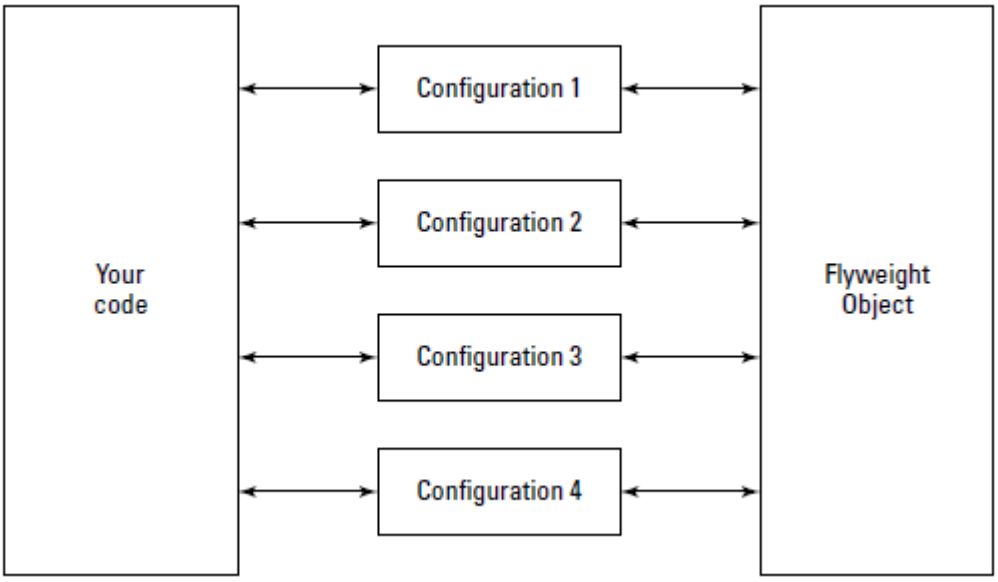


Figure 5-5: Using a flyweight object.

Bất cứ khi nào bạn phải xử lý một lượng lớn các đối tượng, mẫu Flyweight sẽ xuất hiện trong tâm trí bạn. Nếu bạn có thể tách những nội dung giống nhau cần thiết từ những đối tượng này, và tạo một flyweight, hoặc nhiều flyweight, mà hoạt động giống những khuôn mẫu, thì đó chính là cách mẫu flyweight hoạt động

Ví dụ rằng, ở cương vị một chuyên gia thiết kế mẫu, bạn được chọn để giảng dạy về mẫu thiết kế cho một lớp học. Chương trình mà bạn cần có để theo dõi hồ sơ học viên cho từng học viên có thể là những đối tượng thật sự lớn. Bạn quyết định đã đến lúc tiết kiệm tài nguyên hệ thống. Đó là công việc của mẫu Flyweight.

Tạo một học viên

Để tạo mã nguồn cho một đối tượng học viên, bạn quyết định cài đặt nó như một đối tượng Flyweight với tên Student. Đối tượng này được cấu hình sao cho trông giống nhiều học viên mà bạn muốn. Vì vậy bạn thêm vào các hàm thiết lập thông tin và trả thông tin, chẳng hạn tên học viên, mã số, và điểm.

Bạn cùng có thể muốn so sánh học lực của các học viên với nhau, nên bạn thêm một hàm getStanding, có thể trả về mối tương quan của học lưc học viên và điểm trung bình . Mã như sau:

```
public class Student
{
  String name;
  int id;
  int score;
  double averageScore;

  public Student(double a)
  {
    averageScore = a;
  }

  public void setName(String n)
  {
      name = n;
  }

  public void setId(int i)
  {
      id = i;
  }

  public void setScore(int s)
  {
      score = s;
  }

  public String getName()
  {
      return name;
  }

  public int getID()
  {
      return id;
  }

  public int getScore()
  {
      return score;
  }

  public double getStanding()
  {
      return (((double) score) / averageScore - 1.0) * 100.0;
  }
}
```

Lưu ý rằng hàm getStanding sẽ trả về sự khác biệt phần trăm điểm số của sinh viên so với điểm trung bình.

Chạy thử mẫu Flyweight

Để sử dụng mẫu flyweight, bạn phải lưu trữ dữ liệu mà bạn muốn cấu hình cho flyweight. Trong trường hợp này bạn muốn cài đặt cho đối tượng Student giống như một tập hợp các học viên có thật, vì vậy bạn có thể lưu trữ dữ liệu sinh viên (như tên, mã số, điểm) bằng một dãy như sau:

```
public class TestFlyweight
{
  public static void main(String args[])
  {
    String names[] = {"Ralph", "Alice", "Sam"};
    int ids[] = {1001, 1002, 1003};
    int scores[] = {45, 55, 65};
        .
        .
        .
```

Để so sánh học viên này với học viên khác, bạn cần xác định điểm trung bình (tổng điểm chia cho số sinh viên), mã như sau:

```
public class TestFlyweight
{
  public static void main(String args[])
  {
   String names[] = {"Ralph", "Alice", "Sam"};
   int ids[] = {1001, 1002, 1003};
   int scores[] = {45, 55, 65};

   double total = 0;
   for (int loopIndex = 0; loopIndex < scores.length; loopIndex++){
     total += scores[loopIndex];
   }

   double averageScore = total / scores.length;
       .
       .
       .
```

Trong ví dụ này, bạn cần duy nhất một đối tượng flyweight Student, bạn sẽ truyền giá trị điểm số trung bình qua hàm khởi dựng của Student như sau:

```
public class TestFlyweight
{
  public static void main(String args[])
  {
    String names[] = {"Ralph", "Alice", "Sam"};
    int ids[] = {1001, 1002, 1003};
    int scores[] = {45, 55, 65};

    double total = 0;
    for (int loopIndex = 0; loopIndex < scores.length; loopIndex++){
      total += scores[loopIndex];
    }

    double averageScore = total / scores.length;

    Student student = new Student(averageScore);
        .
        .
        .
```

Bây giờ bạn phải cài đặt đối tượng flyweight theo ý muốn, thay vì phải tạo từng đối tượng riêng biệt cho từng sinh viên một. Hãy xem cách thức vòng lặp sau thực hiện:

```
public class TestFlyweight
{
  public static void main(String args[])
  {
    String names[] = {"Ralph", "Alice", "Sam"};
    int ids[] = {1001, 1002, 1003};
    int scores[] = {45, 55, 65};

    double total = 0;
    for (int loopIndex = 0; loopIndex < scores.length; loopIndex++){
      total += scores[loopIndex];
    }

    double averageScore = total / scores.length;

    Student student = new Student(averageScore);

    for (int loopIndex = 0; loopIndex < scores.length; loopIndex++){
      student.setName(names[loopIndex]);
      student.setId(ids[loopIndex]);
      student.setScore(scores[loopIndex]);

      System.out.println("Name: " + student.getName());
      System.out.println("Standing: " +
        Math.round(student.getStanding()));
      System.out.println("");
    }
  }
}
```

Chạy mã trên và bạn nhận được kết quả mong muốn. Đối tượng flyweight đã được cấu hình cho từng học viên, thể hiện được tên và xếp hạng:

```
Name: Ralph
Standing: -18

Name: Alice
Standing: 0

Name: Sam
Standing: 18
```

Thay vì sử dụng ba đối tượng đầy đủ, bạn chỉ cần sử dụng một đối tượng. Cũng gần giống mẫu Singleton, tuy nhiên ý tưởng đằng sau mẫu Flyweight là kiểm soát việc tạo dựng đối tượng, và số lượng đối tượng theo ý bạn muốn.

Xử lý vấn đề đa luồng

Mẫu flyweight được sử dụng để kiểm soát việc tạo dựng đối tượng, nhưng bạn lưu ý rằng nó cũng bị chung một vấn đế với mẫu Singleton mà chúng ta đã nhắc tới. Nếu mã nguồn của bạn có sử dụng đa luồng, bạn có thể tránh việc tạo ra quá nhiều đối tượng flyweight bằng cách tách rời quá trình tạo đối tượng ra khỏi toán tử new như đã từng làm với mẫu Singleton. Bạn có thể tạo đối tượng flyweight ngay khi lớp được nạp lần đầu tiên, ngăn cản việc truy xuất hàm khởi dựng bằng cách gán cho nó một truy cập cục bộ, và cho phép việc tạo đối tượng thông qua hàm getInstance.

```
public class StudentThreaded
{
  String name;
  int id;
  int score;
  double averageScore;
  private static StudentThreaded singleObject =
    new StudentThreaded();

  private StudentThreaded()
  {
  }

  public void setAverageScore(double a)
  {
    averageScore = a;
  }

  public void setName(String n)
  {
      name = n;
  }

  public void setId(int i)
  {
      id = i;
  }

  public void setScore(int s)
  {
      score = s;
  }

  public String getName()
  {
      return name;
  }
  public int getID()
  {
      return id;
  }

  public int getScore()
  {
      return score;
  }
  public double getStanding()
  {
      return (((double) score) / averageScore - 1.0) * 100.0;
  }

  public static StudentThreaded getInstance()
  {
      return singleObject;
  }

}
```

Ví dụ sau, cho thấy khi làm việc với phiên bản mới, việc xử lý đa luồng đã được giải quyết

```
public class TestFlyweightThreaded implements Runable
{
  Thread thread;

  public static void main(String args[])
  {
    TestFlyweightThreaded t = new TestFlyweightThreaded();
  }

  public TestFlyweightThreaded()
  {
    String names[] = {"Ralph", "Alice", "Sam"};
    int ids[] = {1001, 1002, 1003};
    int scores[] = {45, 55, 65};

    double total = 0;
    for (int loopIndex = 0; loopIndex < scores.length; loopIndex++){
      total += scores[loopIndex];
    }

    double averageScore = total / scores.length;

    StudentThreaded student = StudentThreaded.getInstance();

    student.setAverageScore(averageScore);
    student.setName("Ralph");
    student.setId(1002);
    student.setScore(45);

    thread = new Thread(this, "second");
    thread.start();

    System.out.println("Name: " + student.getName() +
      ", Standing: " + Math.round(student.getStanding()));
  }

  public void run()
  {
    StudentThreaded student = StudentThreaded.getInstance();

    System.out.println("Name: " + student.getName() +
      ", Standing: " + Math.round(student.getStanding()));
  }
}
```

Chạy đoạn mã trên, kết quả như sau, bạn sẽ nhận được cùng một đối tượng khi chạy cùng lúc hai luồng:

```
Name: Ralph, Standing: -18
Name: Ralph, Standing: -18
```

Chú ý: Vậy sử dụng mẫu flyweight có hạn chế nào không? Vấn đề chính là bạn sẽ mất thêm thời gian để cài đặt một đối tượng flyweight và nếu bạn phải cài đặt bao bọc mọi thứ, bạn có thể sẽ làm giảm hiệu năng hệ thống nhiều hơn mong đợi. Một hạn chế nữa là: bởi vì bạn tách một lớp mẫu chung ra khỏi đối tượng để tạo flyweight, bạn phải thêm vào một lớp khác trong việc lập trình, và có thể gây ra sự khó khăn trong việc bảo trì và mở rộng.

DP4Dummies – Chương 6: Adapter, Facade

Chương 6: ĐƯA MỘT CÁI CHỐT HÌNH TRÒN VÀO MỘT LỖ HÌNH VUÔNG VỚI MẪU CHUYỂN ĐỔI ADAPTER VÀ MẪU MẶT TIỀN FAÇADE

Trong chương này, chúng ta sẽ đi qua các nội dung sau:

- Sử dụng mẫu chuyển đổi Adapter
- Tạo một adapter
- Chuyển đổi một đối tượng Ace thành đối tượng Acme
- Xử lý các rắc rối với việc chuyển đổi
- Sử dụng mẫu façade

Đôi khi một đối tượng không thật sự phù hợp với những gì ta mong muốn. Một lớp có thể thay đổi, hoặc một đối tượng trở nên khó khăn khi sử dụng. Chương này sẽ là giải pháp tốt cho vấn đề trên khi sử dụng hai mẫu thiết kế: mẫu chuyển đổi adapter và mẫu façade. Mẫu adapter cho phép bạn chuyển đổi một đối tượng để cung cấp cho một lớp khác có thể sử dụng chúng. Mẫu façade cũng tương tự vậy, nó thay đổi vẻ ngoài của một đối tượng, nhưng có một chút khác biệt: bạn sử dụng mẫu này để đơn giản hóa các chức năng của nó, làm cho nó dễ làm việc với đối tượng hay lớp khác.

Kịch bản của mẫu Adapter

“Được,” trưởng nhóm phát triển MegaGigaCo nói, khi đang bước vào phòng, “Thu dọn mọi thứ, quản lý đã ra lệnh rằng chúng ta phải chuyển đổi cơ sở hạ tầng sang một hệ thống mới, chúng ta vừa mua từ công ty của cháu Giám đốc”

“Hmm”, các lập trình viên nói, “Đó là một rắc rối. Giao diện khách hàng trực tuyến của chúng ta cho phép khách hàng sử dụng phần mềm từ công ty Ace và đóng gói chúng trong một lớp tên Ace. Làm sao để chúng ta có thể chuyển đổi chúng cho phù hợp với hệ thống mới?

“Chỉ có thể là một đối tượng Acme mới”, trưởng nhỏm nói “không phải là đối tượng Ace”

“Oh, không”, mọi người nói “Làm sao có thể như vậy được”

Bạn có thể thấy rắc rối ở đây. Hiện tại, hệ thống phù hợp với một đối tượng Ace như hình vẽ

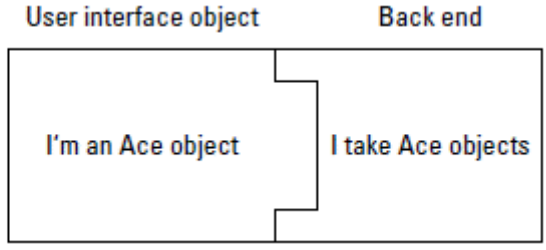


Figure 6-1: These two objects fit together well.

Nhưng khi hệ thống thay đổi, nó yêu cầu một đối tượng Acme (không phải Ace), vì vậy đối tượng Ace không thích hợp nữa. Xem hình sau:

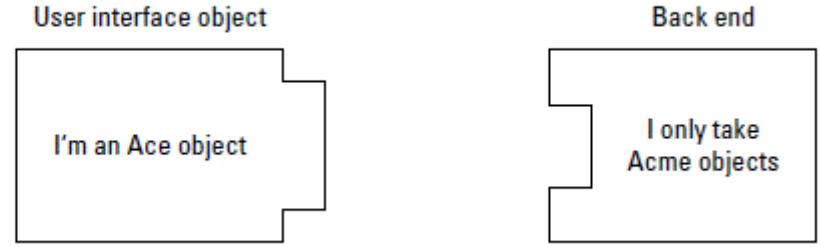


Figure 6-2: The user interface no longer works with the back end.

“Tôi có một giải pháp”. Bạn nói. Mọi người quay sang, và bạn nói tiếp ”Dĩ nhiên, với tư cách là một tư vấn viên, tôi sẽ tính chi phí cao cho việc này”

“OK”, trưởng nhóm nói.

“Bạn cần sử dụng mẫu chuyển đổi Adapter”, bạn giải thích, “Mẫu adapter cho phép bạn chuyển đổi một lớp hoặc một đối tượng sang một lớp hoặc đối tượng mới mà bạn mong muốn”. Bạn vẽ ra giải pháp lên tấm bảng như hình sau:

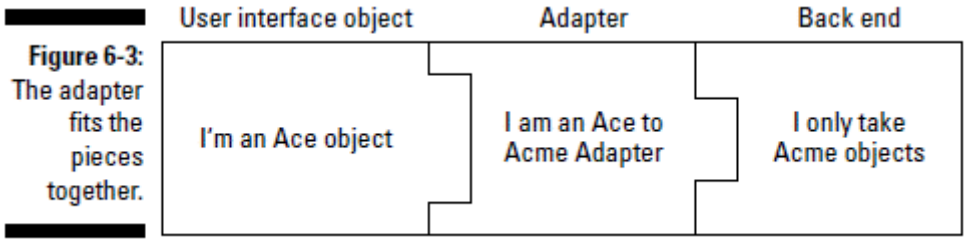


Figure 6-3: The adapter fits the pieces together.

“Ah” cả nhóm phát triển phần mềm nói “Chúng tôi đang hiểu ra vấn đề”

“Tốt” bạn nói “Vậy phải trả phí cho tôi”

Chỉnh sửa rắc rối khi kết nối với mẫu Adapter:

Mẫu thiết kế Adapter cho phép bạn sửa đổi một giao diện giữa đối tượng và một lớp mà không phải sửa đổi trực tiếp lên chúng. Khi bạn làm việc với một ứng dụng mua sẵn, sản phẩm bạn nhận được thường không tương thích với những gì bạn thật sự muốn.

Đây là phần đặc biệt quan trọng trong phát triển trực tuyến. Ngày càng nhiều các công ty tạo ra các sản phẩm cho những công ty lớn, họ đang bỏ qua các phần mềm cho những công ty nhỏ. Và điều đó thật đáng xấu hổ vì việc tương thích của hệ thống, phần mềm từ công ty nhỏ không thể giao tiếp với một hoặc nhiều thành phần khác trong toàn hệ thống. Nhưng việc chuyển sang một giải pháp đắt tiền thì không phải lúc nào cũng cần thiết. Thông thường, giải pháp có thể là một bộ chuyển đổi nhỏ.

Cách tốt nhất để xem mẫu Adapter làm việc là thông qua ví dụ. Hiện tại giao tiếp người dùng công ty MegaGigaCo mà tôi đã nhắc tới trong phần trước của chương này, dữ liệu người dùng được đóng gói trong lớp Ace. Lớp này quản lý tên của khách hàng, với hai hàm sau:

- `setName`
- `getName`

Nhưng theo bạn biết, công ty MegaGigaCo đang chuyển sang sử dụng phần mềm Acme, mà cách thức quản lý khách hàng có khác một chút. Vấn đề ở đây là phần mềm Acme cần một đối tượng Acme. Đối tượng này có tới bốn hàm, chứ không phải hai, dùng để quản lý tên khách hàng. Chúng là:

- `setFirstName`
- `setLastName`
- `getFirstName`
- `getLastName`

Vì vậy bạn cần một bộ chuyển đổi để chắc chắn rằng hệ thống mới Acme có thể xử lý được đối tượng Ace. Bộ chuyển đổi này gọi hai hàm của đối tượng Ace và mở rộng chúng thành bốn hàm mà đối tượng Acme yêu cầu, như hình sau:

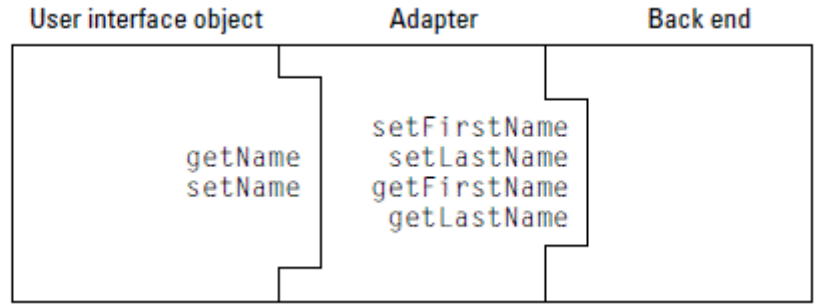


Figure 6-4: The Ace and Acme adapter.

Đây chính là cách thức làm việc của mẫu Adapter

Ghi nhớ: Sách của GoF định nghĩa mẫu Adapter như sau: "Chuyển đổi giao tiếp của một lớp sang một kiểu giao tiếp khác mà khách hàng mong muốn. Mẫu adapter cho phép các lớp có thể làm việc với nhau cho dù giao tiếp của chúng không tương thích nhau"

Thông qua định nghĩa chính thức của mẫu Adapter nói về các lớp, mẫu này bao gồm hai phần chính như sau: một cho đối tượng, một cho lớp. Chúng ta sẽ xem xét cả hai trong chương này.

Bạn sử dụng mẫu Adapter khi bạn cố gắng đưa một cái chốt hình vuông vào cái lỗ hình tròn. Nếu một lớp có giao tiếp không tương thích, bạn có thểm thêm vào một bộ chuyển đổi – giống như bộ chuyển đổi điện trong những chuyến du lịch toàn cầu – để có thể đạt được yêu cầu mong muốn

Mẫu này đặc biệt tốt trong trường hợp bạn đang làm việc với mã nguồn cũ mà yêu cầu là không được thay đổi mã cũ, trong khi phần mềm giao tiếp với mã nguồn cũ này lại thay đổi.

Giờ đã đến lúc để mẫu Adapter làm việc.

Tạo một đối tượng Ace:

Trước khi công ty con của MegaGigaCo phá hỏng mọi thứ trong phòng bạn, đối tượng Ace đang quản lý khách hàng với hai hàm như sau: setName và getName – đây là một giao diện với hai hàm sau:

```
public interface AceInterface
{
  public void setName(String n);
  public String getName();
}
```

Đối tượng Ace sẽ được tạo ra từ lớp AceClass, lớp này hiện thực giao diện trên

```
public class AceClass implements AceInterface
{
    .
    .
    .
}
```

Với hai hàm setName và getName đơn giản như sau:

```
public class AceClass implements AceInterface
{
  String name;

  public void setName(String n)
  {
    name = n;
  }

  public String getName()
  {
    return name;
  }
}
```

Đó là tất cả những gì bạn cần trong hệ thống cũ. Một đối tượng Ace được trả về. Tuy nhiên hiện tại, công ty bạn chuyển sang hệ thống Acme, hệ thống mới cần giao tiếp với đối tượng Acme. (Một lần nữa xin cảm ơn công ty con MegaGigaCo)

Tạo đối tượng Acme

Đối tượng Acme cần quản lý tên khách hàng với bốn hàm sau: setFirstName, setLastName, getFirstName và getLastName. Đây là giao diện của nó, AcmeInterface. Mã như sau:

```
public interface AcmeInterface
{
  public void setFirstName(String f);
  public void setLastName(String l);
  public String getFirstName();
  public String getLastName();
}
```

Đối tượng Acme được tạo từ lớp Acme, hiện thực giao diện AcmeInterface như sau:

```
public class AcmeClass implements AcmeInterface
{
    .
    .
    .
}
```

Và đây là bốn hàm của lớp Acme

```
public class AcmeClass implements AcmeInterface
{
  String firstName;
  String lastName;

  public void setFirstName(String f)
  {
    firstName = f;
  }

  public void setLastName(String l)
  {
    lastName = l;
  }

  public String getFirstName()
  {
    return firstName;
  }

  public String getLastName()
  {
    return lastName;
  }
}
```

Tới lúc này, bạn đã có đối tượng Ace, và đối tượng Acme. Bây giờ bạn cần một bộ chuyển đổi để gắn đối tượng Ace và hệ thống mới Acme.

Tạo đối tượng chuyển đổi Ace-to-Acme

Bạn muốn tạo một bô chuyển đổi để giúp cho ứng dụng có thể làm việc với đối tượng Ace (cho dù nó mong muốn một đối tượng Acme), vì vậy bạn phải tạo một đối tượng chuyển đổi. Đối tượng chuyển đổi này làm việc với sự "kết hợp" composition (xem chương hai để biết thêm về "kết hợp" composition) – một bộ chuyển đổi lưu trữ chính đối tượng mà nó muốn chuyển đổi.

Tiếp tục với ví dụ trong chương này, tôi đặt tên cho bộ chuyển đổi là AceToAcmeAdapter và bởi vì nó trông giống một đối tượng Acme, nó sẽ hiện thực giao diện AcmeInterface:

```
public class AceToAcmeAdapter implements AcmeInterface
{
    .
    .
    .
}
```

Bộ chuyển đổi này sử dụng đối tượng kết hợp “composition” để lưu giữ đối tượng chuyển đổi, đó là một đối tượng AceClass. Bạn có thể chuyển đối tượng này tới lớp thông qua hàm khởi dựng. Mã như sau:

```
public class AceToAcmeAdapter implements AcmeInterface
{
  AceClass aceObject;

  public AceToAcmeAdapter(AceClass a)
  {
    aceObject = a;
  }
  .
  .
  .
}
```

Điểm khác biệt giữa đối tượng Ace và Acme là đối tượng Ace chứa tên khách hàng như là một chuỗi duy nhất, trong khi đối tượng Acme lưu trữ tên và họ khách hàng riêng biệt. Để chuyển đổi giữa đối tượng Ace và Acme, tôi tách phần tên trong đối tượng Ace ra thành tên và họ. Bạn vẫn có thể nhận được tên khách hàng lưu trữ trong đối tượng Ace khi sử dụng hàm getName. Mã như sau:

```
public class AceToAcmeAdapter implements AcmeInterface
{
  AceClass aceObject;
  String firstName;
  String lastName;

  public AceToAcmeAdapter(AceClass a)
  {
    aceObject = a;
    firstName = aceObject.getName().split(" ")[0];
    lastName = aceObject.getName().split(" ")[1];
  }
}
```

Bây giờ bạn đã có tên và họ của khách hàng. Để tương thích với đối tượng Acme, bạn phải hiện thực các hàm của Acme như setFirstName, setLastName, getFirstName và getLastName. Mã như sau:

```
public class AceToAcmeAdapter implements AcmeInterface
{
  AceClass aceObject;
  String firstName;
  String lastName;

  public AceToAcmeAdapter(AceClass a)
  {
    aceObject = a;
    firstName = aceObject.getName().split(" ")[0];
    lastName = aceObject.getName().split(" ")[1];
  }

  public void setFirstName(String f)
  {
    firstName = f;
  }

  public void setLastName(String l)
  {
    lastName = l;
  }

  public String getFirstName()
  {
    return firstName;
  }

  public String getLastName()
  {
    return lastName;
  }
}
```

Tuyệt vời. Bạn đã có một bộ chuyển đổi. Nó làm việc ra sao?

Cho chạy thử bộ chuyển đổi:

Thông qua phần trên, bạn đã chuyển đổi một đối tượng Ace để chúng trông giống như một đối tượng Acme. Giờ là lúc để thấy cách mẫu Adapter làm việc. Bắt đầu bằng việc tạo một đối tượng Ace chứa thông tin khách hàng tên Cary Grant.

```
public class TestAdapter
{
  public static void main(String args[])
  {
    AceClass aceObject = new AceClass();

    aceObject.setName("Cary Grant");
      .
      .
      .
}
```

Sau đó bạn chuyển đối tượng Ace sang cho đối tượng AceToAcmeAdapter

```
public class TestAdapter
{
  public static void main(String args[])
  {
    AceClass aceObject = new AceClass();

    aceObject.setName("Cary Grant");

    AceToAcmeAdapter adapter = new AceToAcmeAdapter(aceObject);
      .
      .
      .
}
```

Và tiếp tục, bạn có thể sử dụng các hàm của Acme như getFirstName và getLastName một cách dễ dàng, không rắc rối gì:

```
public class TestAdapter
{
  public static void main(String args[])
  {
    AceClass aceObject = new AceClass();

    aceObject.setName("Cary Grant");

    AceToAcmeAdapter adapter = new AceToAcmeAdapter(aceObject);

    System.out.println("Customer's first name: " +
      adapter.getFirstName());
    System.out.println("Customer's last name: " +
      adapter.getLastName());
  }
}
```

Kết quả nhận được:

```
Customer's first name: Cary
Customer's last name: Grant
```

Đó là những gì bạn mong muốn khi bạn sử dụng đối tượng Acme thật sự. Bạn đã sử dụng một đối tượng Acme từ một đối tượng Ace nhờ bộ chuyển đổi. Đó là cách thức làm việc của mẫu Adapter. Một Adapter sử dụng một "composition" để lưu trữ một đối tượng mà nó muốn chuyển đổi, và khi các hàm của bộ chuyển đổi được gọi, nó sẽ thay đổi các hàm trong đối tượng gốc sang các hàm của đối tượng mới. Đoạn mã gọi một bộ chuyển đổi sẽ không cần quan tâm tới cách thức làm việc của bộ chuyển đổi, vì bộ chuyển đổi đã thực hiện bên trong và trả về giá cần thiết cho hệ thống.

Sử dụng một đối tượng kết hợp "composition" để bao bọc đối tượng chuyển đổi là một thiết kế hướng đối tượng tốt, như những gì chúng ta đã nói tới trong chương hai. Và chú ý rằng nếu bạn kế thừa một đối tượng adapter, bộ bao bọc adapter sẽ có thể quản lý các đối tượng kế thừa với sự thay đổi ít nhất.

ĐƠN GIẢN HÓA CUỘC SỐNG VỚI MẪU FACADE

Một mẫu thiết kế tương tự với mẫu Adapter là mẫu Façade. Hai mẫu này làm việc theo cùng một cách, nhưng mục đích sử dụng của chúng khác nhau. Mẫu adapter chuyển đổi mã nguồn để làm việc được với mã nguồn khác. Nhưng mẫu Façade cho phép bạn bao bọc mã nguồn gốc để nó có thể giao tiếp với mã nguồn khác dễ dàng hơn.

Ví dụ như, có một người thiết kế ra một cái máy in, và đưa cho bạn một cách tự hào. "Làm sao để máy có thể in?" bạn hỏi

"Đầu tiên," anh ta nói với bạn, "gọi hàm khởi động."

"Được" bạn nói. "Bây giờ nó in chưa?"

"Chưa, bạn phải gọi turnFanOn"

"OK, giờ nó sẽ in chứ", bạn hỏi

"Chưa, hãy gọi hàm làm nóng máy warmUp"

"Được rồi. Giờ nó sẽ in, phải không?"

"Vẫn chưa. Bạn phải gọi hàm getData để đưa dữ liệu từ máy vi tính tới máy in"

“OK, hàm getData. Còn gì nữa không?”

“Hàm định dạng dữ liệu formatData”

“Và gì nữa?”

“Hàm kiểm tra đầu mực checkToner, hàm kiểm tra giấy checkPaperSupply, hàm kiểm tra hệ thống runInternalDiagnostic, hàm …”

“Khoan đã”, bạn nói, vậy viết cho tôi một hàm đại diện façade cho tất cả cái đống lộn xộn này. Hàm façade này sẽ gọi tất cả các hàm khác bên trong nó, và đơn giản hóa một cách đáng kể việc giao tiếp. “Chỉ vậy thôi”

“Đó là cái gì?”, người thiết kế máy in hỏi

“Hàm in ấn print”, bạn nói. “Chỉ cần gọi hàm in ấn print, và máy in hoạt động. Không cần làm gì khác.”

“Hey”, anh ta nói “ đó có thể là một ý tưởng tuyệt vời . Bây giờ bạn có thể thêm một hàm prepareToCallThePrintMethod , một hàm callThePrintMethod , một hàm cleanupAfterPrinting , một hàm…”

“Anh đúng là hết thuốc chữa” bạn nói.

Mẫu thiết kế Façade tạo ra một giao diện OOP dễ dàng để sử dụng. Nó là một vấn đề thiết kế cơ bản – nếu một đối tượng hay một lớp quá khó để giao tiếp, mẫu Façade tạo ra cho bạn một giao diện để giao tiếp dễ dàng hơn. Đây là định nghĩa chính thức của GoF về mẫu Façade: “Cung cấp một giao tiếp duy nhất cho tập hợp các giao tiếp của hệ thống. Façade định nghĩa một giao tiếp cao hơn để giúp các hệ thống con dễ dàng sử dụng”

Thông thường, bạn sử dụng mẫu Façade khi bạn làm việc với những mã nguồn đã được đóng gói một cách cẩu thả. Không phải ai cũng là một chuyên gia OOP, và bạn cũng nhanh chóng thấy được rằng, khi bạn làm việc trong một môi trường phát triển phần mềm thương mại rộng lớn. Khi bạn trở nên mệt mỏi để giao tiếp với những giao diện được thiết kế rắc rối và nhận thấy rằng, bạn nhận được x,y thay vì một cái z đơn giản hơn. Đó là lúc bạn nên sử dụng một giao diện mới.

(ND: từ “giao diện” trong các đoạn văn trên có nguyên văn tiếng anh là interface – tôi vẫn chưa biết dùng từ tiếng việt nào phù hợp, có thể hiểu nó là một nền tảng, cách thức giao tiếp giữa các đối tượng)

Ý tưởng rất đơn giản; một façade làm đơn giản hóa một giao tiếp interface (sử dụng nghĩa chung của từ “interface”, không phải giao diện interface trong Java) giữa một lớp hay một đối tượng.

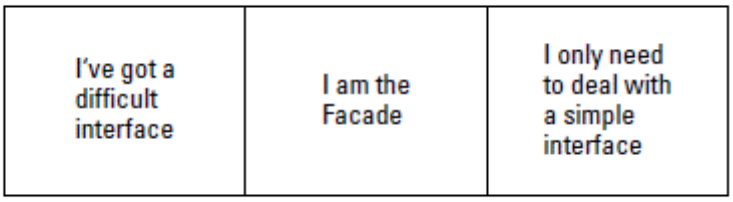


Figure 6-7: The Facade pattern makes an interface less complicated.

Bạn thường sử dụng mẫu thiết kế Façade khi bạn muốn mã nguồn đơn giản hơn nhưng lại không thể chỉnh sửa mã nguồn cũ. Thông qua việc sử dụng mẫu Façade bạn có thể giải quyết vấn đề, nó thêm vào một lớp khác bên trên, và nếu mã nguồn của lớp bên dưới thay đổi, bạn cũng phải thay đổi luôn mã nguồn của mẫu Façade.

Có một định nghĩa OOP làm việc ở đây, đôi khi còn được gọi là “Nguyên tắc về sự hiểu biết ít nhất” , đôi lúc được gọi là “Luật của Demeter” , đôi khi được gọi là “sự đóng gói hiệu quả”. Đây là ý tưởng nâng cao sự hiệu quả của OOP, bạn không muốn những thực thể riêng biệt (lớp hay đối tượng) phải biết quá nhiều về nhau. Càng ít càng tốt, bạn có thể che dấu chi tiết của từng lớp hay đối tượng và làm cho sự liên kết của chúng lỏng lẻo càng nhiều càng tốt. (Xem chương bốn để bíết thêm về “tháo lỏng các mối liên kết” loose coupling) . Nếu một đối tượng cần phải biết quá nhiều về đối tượng khác, đó chính là lúc cần sử dụng mẫu Façade.

Ghi chú: Hãy tháo lỏng các mối liên kết càng nhiều càng tốt.

Làm việc với một đối tượng khó khăn

Đây là một ví dụ minh họa cách thức mẫu Façade làm việc. Công ty bạn vừa mua một công ty đối thủ, người quản lý đang rất hân hoan.

“Hmm” bạn nói “ Chúng ta đang gặp rắc rối với vấn đề tương thích?. Sản phẩm của họ quá khác biệt với chúng ta”

“Phi lý”, Ông chủ lớn nói “Mọi việc không thể đơn giản hơn”

“Được,” bạn nói “Làm sao ông có thể đặt tên cho nó”

“Không thể đơn giản hơn. Cứ gọi hàm setFirstNameCharacter. Nó sẽ đặt kí tự đầu cho tên”

“OK, vậy kí tự thứ hai của tên thì sao?”

“Chỉ cần gọi hàm setSecondNameCharacter. Không thể đơn giản hơn”

“OK, để tôi tiếp tục.” bạn nói ”Vậy để đặt tên cho một sản phẩm, ông gọi hàm setFirstNameCharacter để thiết lập kí tự đầu tiên cho tên , sau đó gọi hàm setSecondNameCharacter để thiết lập kí tự thứ hai , và bằng cách đó ông tiếp tục gọi hàm setFiveMillionNameCharacter để thiết lập kí thứ thứ năm triệu cho tên?”

“Không”, Ông chủ lớn nói “Bạn chỉ có thể đặt tên với bảy kí tự”

“À,” bạn nói “Không thể đơn giản hơn”

“Đúng”, ông chủ nói

Đây là đoạn mã, sau khi bạn hợp nhất với sản phẩm của công ty mới, lớp DifficultProduct

```
public class DifficultProduct
{
  public DifficultProduct()
  {
  }
  .
  .
  .
}
```

Bạn đặt tên cho sản phẩm này từng kí tự một, sử dụng hàm setFirstNameCharacter, setSecondNameCharacter, setThirdNameCharacter, và hàm cho kí tự thứ tư, thứ năm… như sau:

```
public class DifficultProduct
{
  char nameChars[] = new char[7];

  public DifficultProduct()
  {
  }

  public void setFirstNameCharacter(char c)
  {
    nameChars[0] = c;
  }

  public void setSecondNameCharacter(char c)
  {
    nameChars[1] = c;
  }

  public void setThirdNameCharacter(char c)
  {
    nameChars[2] = c;
  }

  public void setFourthNameCharacter(char c)
  {
    nameChars[3] = c;
  }

  public void setFifthNameCharacter(char c)
  {
    nameChars[4] = c;
  }

  public void setSixthNameCharacter(char c)
  {
    nameChars[5] = c;
  }

  public void setSeventhNameCharacter(char c)
  {
    nameChars[6] = c;
  }
  .
  .
  .
}
```

Và để lấy được tên sản phẩm, bạn gọi hàm getName, trả về một chuỗi.

```
public class DifficultProduct
{
  char nameChars[] = new char[7];

  public DifficultProduct()
  {
  }

  public void setFirstNameCharacter(char c)
  {
    nameChars[0] = c;
  }

  public void setSecondNameCharacter(char c)
  {
    nameChars[1] = c;
  }

  public void setThirdNameCharacter(char c)
  {
    nameChars[2] = c;
  }

  public void setFourthNameCharacter(char c)
  {
    nameChars[3] = c;
  }

  public void setFifthNameCharacter(char c)
  {
    nameChars[4] = c;
  }

  public void setSixthNameCharacter(char c)
  {
    nameChars[5] = c;
  }

  public void setSeventhNameCharacter(char c)
  {
    nameChars[6] = c;
  }

  public String getName()
  {
    return new String(nameChars);
  }
}
```

Đây là cách tạo một máy in, đặt tên theo từng kí tự một:

```
DifficultProduct difficultProduct = new DifficultProduct();

difficultProduct.setFirstNameCharacter('p');
difficultProduct.setSecondNameCharacter('r');
difficultProduct.setThirdNameCharacter('i');
difficultProduct.setFourthNameCharacter('n');
difficultProduct.setFifthNameCharacter('t');
difficultProduct.setSixthNameCharacter('e');
difficultProduct.setSeventhNameCharacter('r');
```

“Thấy không?” ông chủ nói “không thể dễ dàng hơn”

“Quá sức phi lý.” bạn nói ”Tôi sẽ viết một mẫu Façade”

Tạo một mẫu Façade đơn giản:

Ông chủ đồng ý cách của bạn và bạn bắt đầu viết một façade đơn giản như sau:

```
public class SimpleProductFacade
{
  public SimpleProductFacade()
  {
  }
  .
  .
  .
}
```

Façade này phải bao bọc đối tượng (đối tượng DifficultProduct trong ví dụ). Thông thường, bạn viết một façade mà cho phép façade chỉnh sửa giao diện bên ngoài của đối tượng. Bạn cũng có thể chuyển tham số cấu hình vào hàm khởi dựng của façade, nhưng điều đó không cần thiết trong ví dụ này, ta chỉ cần tạo mới đối tượng DifficultProduct như sau:

```
public class SimpleProductFacade
{
  DifficultProduct difficultProduct;

  public SimpleProductFacade()
  {
    difficultProduct = new DifficultProduct();
  }
  .
  .
  .
}
```

Rắc rối với việc sử dụng đối tượng DifficultProduct nguyên thủy là cách mà ta thiết lập tên cho nó, sử dụng vụng về một loạt các hàm setFirtNameCharacter, setSecondNameCharacter, setThirdNameCharacter và vân vân. Để sửa chữa , bạn quyết định cung cấp một façade với một hàm đơn giản là setName để đặt tên cho đối tượng. Mã như sau:

```
public class SimpleProductFacade
{
  DifficultProduct difficultProduct;

  public SimpleProductFacade()
  {
    difficultProduct = new DifficultProduct();
  }

  public void setName(String n)
  {
    char chars[] = n.toCharArray();

    if(chars.length > 0){
      difficultProduct.setFirstNameCharacter(chars[0]);
    }

    if(chars.length > 1){
      difficultProduct.setSecondNameCharacter(chars[1]);
    }

    if(chars.length > 2){
      difficultProduct.setThirdNameCharacter(chars[2]);
    }

    if(chars.length > 3){
      difficultProduct.setFourthNameCharacter(chars[3]);
    }

    if(chars.length > 4){
      difficultProduct.setFifthNameCharacter(chars[4]);
    }

    if(chars.length > 5){
      difficultProduct.setSixthNameCharacter(chars[5]);
    }

    if(chars.length > 6){
      difficultProduct.setSeventhNameCharacter(chars[6]);
    }
  }
  .
  .
  .
}
```

Có một hàm không cần dùng Façade, đó là hàm getName, mã như sau:

```
public class SimpleProductFacade
{
  DifficultProduct difficultProduct;

  public SimpleProductFacade()
  {
    difficultProduct = new DifficultProduct();
  }

  public void setName(String n)
  {
    char chars[] = n.toCharArray();

    if(chars.length > 0){
      difficultProduct.setFirstNameCharacter(chars[0]);
    }
    .
    .
    .
    if(chars.length > 6){
      difficultProduct.setSeventhNameCharacter(chars[6]);
    }
  }

  public String getName()
  {
    return difficultProduct.getName();
  }
}
```

Bây giờ bạn đã bao bọc một đối tượng khó giao tiếp và hợp nhất các hàm rắc rối thành hàm dễ sử dụng. Hãy xem cách thức hoạt động của mẫu Façade

Chạy thử mẫu Façade

Đầu tiên tạo một đối tượng SimpleProductFacade, sau đó thiết lập tên đối tượng “printer” với hàm setName và nhận lại với hàm getName.

```
public class TestFacade
{
  public static void main(String args[])
  {
    TestFacade t = new TestFacade();
  }

  public TestFacade()
  {
    SimpleProductFacade simpleProductFacade =
      new SimpleProductFacade();

    simpleProductFacade.setName("printer");

    System.out.println("This product is a " +
      simpleProductFacade.getName());
  }
}
```

Và đây là kết quả:

```
This product is a printer
```

Bạn đã chinh phục được đối tượng khó khăn với giao diện khó sử dụng bằng một façade.

DP4Dummies – Chương 7: Template, Builder

CHƯƠNG 7:TẠO HÀNG LOẠT ĐỐI TƯỢNG VỚI MẪU TEMPLATE (Khuôn Mẫu)VÀ MẪU BUILDER (Thợ Xây)

Trong chương này, chúng ta đi qua một số nội dung sau:

- Sử dụng mẫu Template
- Tạo rô bốt sử dụng mẫu template
- Kế thừa mẫu template
- Hiểu biết sự khác biệt giữa mẫu Template và mẫu Builder
- Sử dụng mẫu Builder

“Tin tốt”, giám đốc GigundoCorp – một công ty mới mà bạn đang nhận trách nhiệm tư vấn – vừa nói trong khi chạy vào phòng họp “Chúng ta đã nhận được hợp đồng đó”

“Hợp đồng nào?”, mọi người hỏi

“Hợp đồng về những con rô bốt tự động lắp ráp xe hơi”, vị giám đốc nói.

“Ồ, thì ra là hợp đồng đó” Mọi người nói.

“Giờ thì về phòng và viết chương trình thôi”, vị giám đốc vừa nói và xua đuổi mọi người ra khỏi phòng họp

“Chờ một lát”, bạn nói “Chúng ta có nên dành chút thời gian cho vấn đề thiết kế không? Ví dụ: có khả năng chúng ta sẽ tạo một loại khác của rô bốt trong tương lai chẳng hạn”

“Chắc chắn rồi”, vị giám đốc nói. “Chúng ta có một tá hồ sơ dự thầu ngoài đó. Nhưng không có thời gian nghĩ về nó đâu. Chúng ta cần phải bắt đầu tạo những con rô bốt tự động trước”

“Vâng”, các lập trình viên rên rỉ và mọi người trở về phòng của mình.

“Có điều gì đó mách bảo với tôi rằng họ đang mắc phải sai lầm”, bạn tự nhủ trong căn phòng trống rỗng, rải rác những ly Styrofoam trống rỗng lăn lóc khắp sàn.

Chương này nói này về hai mẫu thiết kế giúp bạn có một cách thức khéo léo hơn trong việc tạo dựng các đối tượng: mẫu Template Method và mẫu Builder. Mẫu Template Method cho phép các lớp con định nghĩa lại các bước tạo đối tượng, rất thích hợp cho việc tạo ra các chủng loại rô bốt khác nhau. Mẫu Builder giúp bạn uyển chuyển hơn trong việc tạo đối tượng vì nó tách rời quá trình khởi tạo ra khỏi bản thân đối tượng. Cả hai mẫu sẽ được thảo luận trong chương này

Tạo con rô bốt đầu tiên

Các lập trình viên của GigundoCorp đã xào nấu ra phần mềm của họ trong vài ngày và nó vừa đủ đơn giản. Lớp robot bắt đầu với một hàm khởi tạo như sau:

```
public class Robot
{
  public Robot()
  {
  }
    .
    .
    .
}
```

Và có một số hành động mà robot có thể thực hiện, ví dụ như, để khởi động robot, bạn gọi hàm bắt đầu Start, để robot làm việc, bạn gọi hàm lắp ráp assemble, để kiểm tra sản phẩm, bạn gọi hàm kiểm tra test, và vân vân…

```
public class Robot
{
  public Robot()
  {
  }

  public void start()
  {
    System.out.println("Starting....");
  }

  public void getParts()
  {
    System.out.println("Getting a carburetor....");
  }

  public void assemble()
  {
    System.out.println("Installing the carburetor....");
  }

  public void test()
  {
    System.out.println("Revving the engine....");
  }

  public void stop()
  {
    System.out.println("Stopping....");
  }
}
```

Và tất cả những gì bạn cần là một phương thức, tên là *go* , nó sẽ làm cho robot làm việc bằng cách gọi các hàm *start, getParts, assemble, test* và *stop* như sau:

```
public class Robot
{
  public Robot()
  {
  }

  public void go()
  {
    start();
    getParts();
    assemble();
    test();
    stop();
  }

  public void start()
  {
    System.out.println("Starting....");
  }

  public void getParts()
  {
    System.out.println("Getting a carburetor....");
  }

  public void assemble()
  {
    System.out.println("Installing the carburetor....");
  }

  public void test()
  {
    System.out.println("Revving the engine....");
  }

  public void stop()
  {
    System.out.println("Stopping....");
  }
}
```

Bạn có thể nhanh chóng viết chương trình kiểm tra. Đầu tiên tạo một robot và gọi hàm *go* như sau:

```
public class TestRobot
{
  public static void main(String args[])
  {
    Robot robot = new Robot();

    robot.go();
  }
}
```

Và khi chạy chương trình, bạn nhận được kết quả:

```
Starting....
Getting a carburetor....
Installing the carburetor....
Revving the engine....
Stopping....
```

“Tuyệt vời”, giám đốc điều hành phấn khích. “Phần thưởng luôn ở xung quanh. Tôi đã nói với anh rằng họ không cần cái thứ mẫu thiết kế vớ vẩn”. Các lập trình viên của công ty ném cho bạn một ánh nhìn dè bỉu.

Tạo Robot với Mẫu thiết kế Template Method

Ngày tiếp theo, “Tin tốt”, giám đốc điều hành của GigundoCorp la lớn, trong khi phóng vào phòng họp. “Chúng ta kí được hợp đồng khác!”

“Hợp đồng khác nào?” Mọi người hỏi

“Hợp đồng cho robot nướng bánh” Vị giám đốc nói “Giờ thì ra khỏi đây và viết phần mềm cho nó”

Các lập trình viên nhìn vào trong ly cà phê của họ “Chúng ta phải viết lại tất cả phần mềm từ đầu”, họ nói

Vị giám đốc liếc mắt nhìn bạn và hỏi “Có tốn nhiều chi phí không?”

“Rất nhiều”, các lập trình viên nói. Và bạn thì đang chống lại sự thúc giục để nói rằng “Tôi đã nói với các anh từ trước”

Đây là thời điểm thích hợp để nói về mẫu thiết kế Template Method. Có một rắc rối mà lập trình viên GigundoCopr đối mặt, họ có một con robot tự động như hình sau:

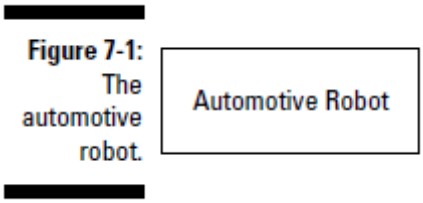


Figure 7-1: The automotive robot.

Nhưng bây giờ họ cần một con robot nướng bánh như hình sau, và thế là phải viết lại mã nguồn từ đầu:

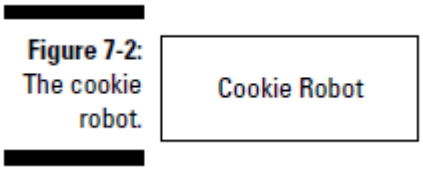


Figure 7-2: The cookie robot.

Con robot nướng bánh có một số chức năng giống như con robot lắp ráp ô tô, như là hàm *start, stop,* tuy nhiên nó có những sự khác biệt như lắp ráp *assemble* sẽ không hiển thị "*Getting a carburetor*" mà thay vào đó là "*Getting flour and sugar…*"

Đó là nơi mà mẫu thiết kế Template Method được áp dụng. Mẫu này nói rằng, bạn có thể viết một phương thức, dùng để xác định một loạt các thuật toán, giống như hàm go mà bạn thấy trước đây, để chạy một loạt các chức năng cho robot như hình:

```
public void go()
{
  start();
  getParts();
  assemble();
  test();
  stop();
}
```

Sau đó bạn đưa hàm này vào một bộ khuôn template bằng cách cho phép các lớp con định nghĩa lại các bước thuật toán theo cách cần thiết. Trong trường hợp này, để làm một con robot nướng bánh, bạn sẽ viết lại các hàm getParts, assemble, và test.

Theo định nghĩa chính thức của sách GoF, mẫu Template Method như sau: "Định nghĩa một bộ khung của một thuật toán trong một chức năng, chuyển giao việc thực hiện nó cho các lớp con. Mẫu Template Method cho phép lớp con định nghĩa lại cách thực hiện của một thuật toán, mà không phải thay đổi cấu trúc thuật toán."

Điều này có nghĩa là bạn nên sử dụng mẫu Template Method khi bạn có một thuận toán được tạo bởi nhiều bước, và bạn muốn thể tùy chỉnh một số bước trong đó. Chú ý rằng

nếu bạn muốn viết lại mọi thứ từ đầu – khi mọi bước đều phải tùy chỉnh lại – thì bạn không cần dùng template.

Tạo robot bằng bộ khuôn Template

Nếu bạn có một bộ khuôn Template dựa trên robot, bạn có thể cho nó kế thừa như hình sau:

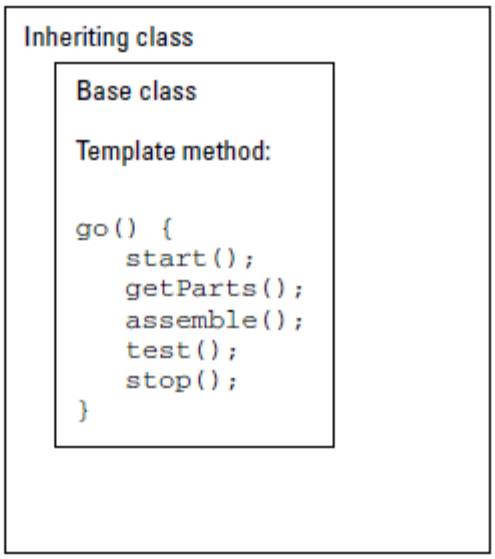


Figure 7-3: Inheriting from the template base class.

Bằng cách gọi hàm go, tập hợp các thuật toán sẽ được thực hiện. Để tùy chỉnh trong lớp kế thừa, bạn chỉ cần viết lại một số bước nào bạn muốn, trong trường hợp robot nướng bánh sẽ như hình sau:

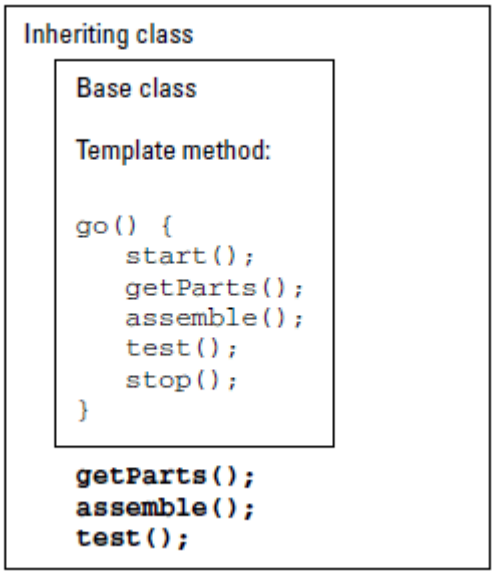


Figure 7-4: Modifying the inherited methods.

Đó là ý tưởng đằng sau mẫu thiết kế Template Method – Một chức năng bao gồm nhiều bước sẽ được tùy chỉnh bởi lớp con. Trong trường hợp bạn cần hai robot, một robot lắp ráp ô tô, một robot nướng bánh, mọi việc sẽ như thế nào?

Bạn bắt đầu bằng cách tạo một bộ khuôn Template trong một lớp trừu tượng abstract (để lớp khác có thể kế thừa nó), gọi là RobotTemplate

```
public abstract class RobotTemplate
{
  public final void go()
  {
    start();
    getParts();
    assemble();
    test();
    stop();
  }
  .
  .
  .
}
```

Và lớp này cũng cài đặt việc thực hiện mặc định cho từng chức năng trong hàm *algorithm, start, getParts, assemble, test* và *stop*.

```
public abstract class RobotTemplate
{
  public final void go()
  {
    start();
    getParts();
    assemble();
    test();
    stop();
  }

  public void start()
  {
    System.out.println("Starting....");
  }

  public void getParts()
  {
    System.out.println("Getting parts....");
  }

  public void assemble()
  {
    System.out.println("Assembling....");
  }

  public void test()
  {
    System.out.println("Testing....");
  }

  public void stop()
  {
    System.out.println("Stopping....");
  }
}
```

Nếu một con robot sử dụng đúng các phương thức này, ví dụ như hàm *start* và *stop*, chúng ta không cần phải viết lại chúng. Ngược lại bạn có thể thay đổi các phương thức này trong các lớp con.

Ví dụ, bạn có thể sử dụng RobotTemplate để tạo một con robot lắp ráp ô tô. Bạn có thừa kế từ lớp trừu tượng RobotTemplate trong một lớp mới, lớp AutomotiveTobot.

```
public class AutomotiveRobot extends RobotTemplate
{
    .
    .
    .
}
```

Robot lắp ráp ô tô này phải viết lại một số hàm của RobotTemplate như hàm *getParts* sẽ thông báo *"Getting a carburetor…"*, hàm *assemble* sẽ thông báo *"Installing the carburetor…"*, và hàm *test* sẽ thông báo *"Revving the engine…"*. Bạn thấy đó, bạn có thể tùy chỉnh các bước trong một thuật toán được cung cấp bởi một bộ khuôn template:

```
public class AutomotiveRobot extends RobotTemplate
{
  public void getParts()
  {
    System.out.println("Getting a carburetor....");
  }

  public void assemble()
  {
    System.out.println("Installing the carburetor....");
  }

  public void test()
  {
    System.out.println("Revving the engine....");
  }
}
```

Bạn cũng có thể tùy chỉnh mã nguồn dựa trên template bằng cách thêm vào một số hàm, ví dụ như hàm khởi tạo sẽ nhận tên của con robot, và hàm *getName* sẽ trả về tên này.

```
public class AutomotiveRobot extends RobotTemplate
{
  private String name;

  public AutomotiveRobot(String n)
  {
    name = n;
  }

  public void getParts()
  {
    System.out.println("Getting a carburetor....");
  }

  public void assemble()
  {
    System.out.println("Installing the carburetor....");
  }

  public void test()
  {
    System.out.println("Revving the engine....");
  }

  public String getName()
  {
    return name;
  }
}
```

Tuyệt vời. Bạn đã kế thừa phương thức *go* từ template, và tùy chỉnh nó cho robot lắp ráp ô tô.

Bạn cũng có thể tùy chỉnh hàm *go* kế thừa từ template, trong trường hợp tạo robot nướng bánh. Bạn tạo lớp mới CookieRobot, kế thừa từ lớp RobotTemplate. Bạn có thể viết lớp CookieRobot bằng cách làm cho hàm *getParts* thông *báo "Getting flour and sugar…"*, hàm *assemble* thông báo *"Baking a cookie…"*, và hàm *test* thông báo *"Crunching a cookie…"*

```
public class CookieRobot extends RobotTemplate
{
  private String name;

  public CookieRobot(String n)
  {
     name = n;
  }

  public void getParts()
  {
    System.out.println("Getting flour and sugar....");
  }

  public void assemble()
  {
    System.out.println("Baking a cookie....");
  }

  public void test()
  {
    System.out.println("Crunching a cookie....");
  }

  public String getName()
  {
    return name;
  }
}
```

Tới giờ, bạn đã sử dụng hàm *go* từ bộ khuôn template để tạo hai lớp mới, AutomotiveRobot và CookieRobot, và bạn đã viết lại một số bước trong thuật toán tùy thuộc vào hai loại robot khác nhau. Bạn đã không phải viết lại hai lớp này từ đầu.

Kiểm tra việc tạo Robot

Bạn hãy tạo hai đối tượng của hai lớp AutomotiveRobot và CookieRobot, và gọi hàm *go* như sau:

```
public class TestTemplate
{
  public static void main(String args[])
  {
    AutomotiveRobot automotiveRobot =
      new AutomotiveRobot("Automotive Robot");

    CookieRobot cookieRobot = new CookieRobot("Cookie Robot");

    System.out.println(automotiveRobot.getName() + ":");
    automotiveRobot.go();
    System.out.println();
    System.out.println(cookieRobot.getName() + ":");
    cookieRobot.go();
  }
}
```

Khi bạn chạy thử chương trình, bạn có thể thấy rằng bạn thật sự có thể tùy chỉnh một số bước trong thuật toán của hai loại robot khác nhau.

```
Automotive Robot:
Starting....
Getting a carburetor....
Installing the carburetor....
Revving the engine....
Stopping....

Cookie Robot:
Starting....
Getting flour and sugar....
Baking a cookie....
Crunching a cookie....
Stopping....
```

Thêm vào một "hook" (ND: Hook – móc câu – một kỹ thuật chặn bắt thông điệp chương trình)

Bạn cũng có thể thêm vào một hook trong thuật toán. Một hook là phương pháp kiểm soát một số khía cạnh của thuật toán. Ví dụ, nếu bạn muốn phần kiểm tra testing trong thuật toán Robot có trả về kết quả đúng không, bạn có thể thêm vào một điều kiện, một hàm hook có tên *testOK* như sau:

```
public abstract class RobotHookTemplate
{
  public final void go()
  {
    start();
    getParts();
    assemble();
    if (testOK()){
      test();
    }
    stop();
  }

  public void start()
  {
    System.out.println("Starting....");
  }

  public void getParts()
  {
    System.out.println("Getting parts....");
  }

  public void assemble()
  {
    System.out.println("Assembling....");
  }

  public void test()
  {
    System.out.println("Testing....");
  }

  public void stop()
  {
    System.out.println("Stopping....");
  }

  public boolean testOK()
  {
    return true;
  }
}
```

Mặc định, bạn có thể bỏ qua hàm hook *testOK* – nếu không làm gì khác, thuật toán Robot sẽ gọi đầy đủ các bước, bao gồm cả hàm test. Tuy nhiên bạn có thể "câu móc" vào thuật toán bằng cách viết lại hàm testOK trong một lớp con, lớp CookieHookRobot, nơi mà hàm *testOK* sẽ trả về giá trị false, không phải là true.

```
public class CookieHookRobot extends RobotHookTemplate
{
  private String name;

  public CookieHookRobot(String n)
  {
     name = n;
  }

  public void getParts()
  {
    System.out.println("Getting flour and sugar....");
  }

  public void assemble()
  {
    System.out.println("Baking a cookie....");
  }

  public String getName()
  {
    return name;
  }

  public boolean testOK()
  {
    return false;
  }

}
```

Bởi vì hàm hook *testOK* trả về giá trị false, thuật toán Robot sẽ không gọi hàm test từ hàm *go*, bạn có thể xem mã sau:

```
  public final void go()
  {
    start();
    getParts();
    assemble();
    if (testOK()){
      test();
    }
    stop();
  }
```

Kiểm tra hàm hook:

Bây giờ tạo chương trình, và gọi hàm *cookieHookRobot.go:*

```
public class TestHookTemplate
{
  public static void main(String args[])
  {
    CookieHookRobot cookieHookRobot =
      new CookieHookRobot("Cookie Robot");

    System.out.println(cookieHookRobot.getName() + ":");
    cookieHookRobot.go();
  }
}
```

Bạn sẽ thấy thuật toán Robot thực hiện, trừ bước test:

```
Cookie Robot:
Starting....
Getting flour and sugar....
Baking a cookie....
Stopping....
```

Bạn thấy đó, bạn đã không phải làm bất cứ thứ gì với hàm hook, tuy nhiên nếu bạn muốn, bạn có thể tác động lên việc thực hiện của thuật toán. Nếu bạn xây dựng một thuật toán sử dụng nhiều hàm trừu tượng, từng hàm này sẽ được viết lại ở lớp con, mặc khác, hàm hook sẽ không phải viết lại, trừ khi bạn muốn thay đổi việc thực thi mặc định của thuật toán.

Bạn sử dụng mẫu thiết kế Template Method khi bạn có một thuật toán với nhiều bước và bạn muốn cho phép tùy chỉnh chúng trong lớp con. Thật dễ dàng. Bằng cách viết lại các hàm đã được khai báo trong lớp trừu tượng, bạn sẽ thay đổi được theo cách bạn muốn.

Mẫu thiết kế Template Method là một ý tưởng tuyệt vời khi bạn có một thuật toán nhiều bước mà chính bạn có thể tùy chỉnh nó. Có một mẫu thiết kế khác cũng làm việc giống vậy, mà tôi sẽ thảo luận trong phần tới của chương, nó là mẫu Builder.

Xây dựng Robots với mẫu Builder

“Tin tốt!” Giám đốc điều hành của GigundoCorp reo lên, trong khi phóng như bay vào phòng họp. “Khách hàng của chúng ta nói rằng họ muốn kiểm soát nhiều hơn tính năng của Robot, vì vậy chúng ta không thể sử dụng những bộ khuôn Template đã viết sẵn được nữa. Bây giờ họ muốn họ có thể chọn hành động mà robot sẽ thực hiện”

“Để tôi làm rõ chỗ này”, bạn nói “Đầu tiên, chúng ta thiết lập mọi thứ , robot khởi động, nhận nguyên liệu, lắp ráp, kiểm tra và dừng. Nhưng bây giờ khách hàng lại muốn kiểm soát trình tự này và chọn lựa những chức năng họ muốn? Có thể là robot khởi động, rồi kiểm tra, rồi lắp ráp, rồi dừng?”

“Đúng vậy”, Giám đốc nói

“Đây là thời điểm để sử dụng một mẫu thiết kế mới”, bạn nói

“Tôi e rằng phải làm như vậy”, Giám đốc nói.

Những quy định của khách hàng

Trong mẫu thiết kế Template Method, vấn đề chính là những thuật toán nhiều bước – bạn có thể cài đặt nó theo cách bạn muốn, và những lớp con sử dụng theo cách bạn đã thiết lập (Mặc dù bạn có thể viết lại một số bước, nhưng quy trình vẫn không thay đổi). Nhưng bây giờ tình hình đã khác – khách hàng muốn họ thiết lập trình tự hoạt động và số lượng các bước của thuật toán. Vì vậy mã nguồn mà bạn đã phát triển không còn là trung tâm chính nữa, bạn phải đóng gói nó trong một lớp mới, lớp builder.

Mẫu Template Method mà ta đã được làm quen trong phần trước cho phép bạn tùy chỉnh các bước của một thuật toán bằng cách viết lại các bước trong thuật toán như hình sau:

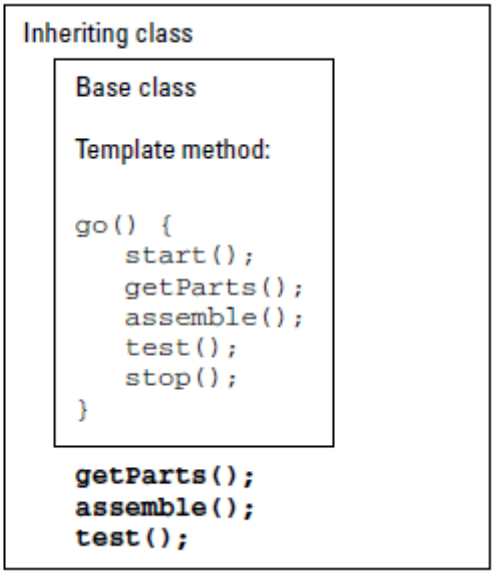


Figure 7-5: The Template Method pattern allows you to customize the multi-step algorithm.

Mọi chức năng đều dựa trên khuôn mẫu Template trong mẫu thiết kế này, và bạn có thể tùy chỉnh template theo cách bạn muốn. Nhưng bây giờ bạn không còn điều khiển thuật toán nữa, thay vào đó chính khách hàng thực hiện. Họ tạo robot với những chức năng và trình tự họ muốn. Ví dụ để thêm hành động khởi động, khách hàng có thể gọi hàm *addStart*. Để thêm hành động kiểm tra, họ gọi hàm *addTest* và vân vân. Hình minh họa như sau:

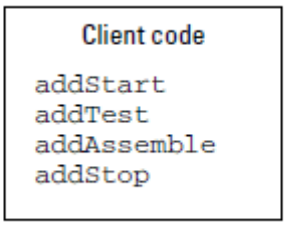


Figure 7-6: Customizing client code with an addStart method.

Để có thể đáp ứng yêu cầu kiểm soát hành động robot của khách hàng GigundoCorp, bạn phải chuyển mã nguồn cũ qua một lớp mới, lớp CookieRobotBuilder, lớp này hỗ trợ các hàm addStart, addTest, addAssemble và hàm addStop, như hình sau:

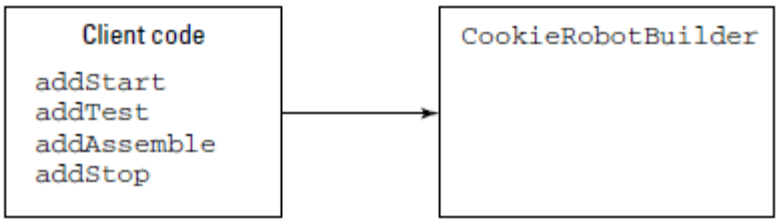


Figure 7-7: Separating out the construction code into the Cookie Robot Builder class.

Nhờ đó, khách hàng có thể sử dụng CookieRobotBuilder để tạo robot nướng bánh. Khi khách hàng tạo xong robot, mã nguồn sẽ gọi hàm getRobot của đối tượng CookieRobotBuilder để nhận về một robot mới, như hình vẽ sau:

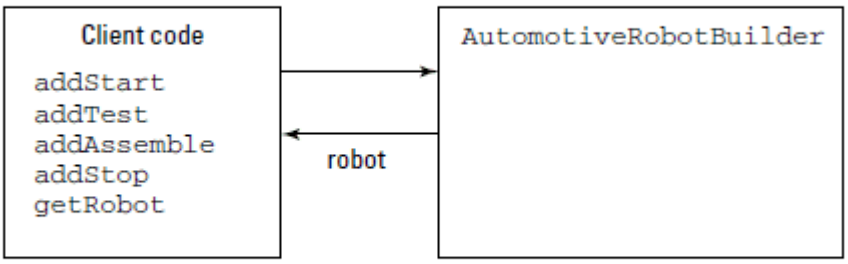


Figure 7-9: Using the Automotive Robot Builder class for construction instead of the Cookie Robot Builder class.

Và bây giờ khách hàng đã nắm quyền kiểm soát các thuật toán, bạn không phải kế thừa một template mẫu nữa.Thay vì vậy, để tạo một loại khác của robot, bạn cho phép khách hàng sử dụng những đối tượng builder khác nhau.

Ý tưởng chính như sau: bây giờ khách hàng có thể thiết lập trình tự và số lượng các bước trong thuật toán, và chọn lựa đúng đối tượng builder để tạo ra robot mà họ muốn.

Sách của GoF nói rằng, mẫu thiết kế Builder "Tách rời việc tạo dựng một đối tượng phức tạp ra khỏi bản thân đối tượng vì vậy cho phép cùng một quá trình tạo dựng có thể tạo ra nhiều loại đối tượng khác nhau"

Khác biệt lớn nhất giữa mẫu Template Method và mẫu Builder là ai sẽ tạo ra trình tự các bước trong thuật toán. Trong mẫu Template, bạn là người tạo ra trình tự, và các lớp con sẽ hiện thực chúng. Trong mẫu Builder, khách hàng sẽ thiết lập trình tự và số lượng các bước trong thuật toán, và hoán đổi giữa các builder mà phải cung cấp để tạo ra các đối tượng khác nhau thể hiện thuật toán đó.

Sử dụng mẫu thiết kế Builder khi bạn muốn khách hàng kiểm soát được quá trình tạo dựng. Ví dụ, đây là mẫu thiết kế mà bạn muốn khi bạn xây dựng robot sử dụng cùng một quá trình khởi tạo nhưng muốn có thể tạo ra những con robot khác nhau. Tất cả những gì khách hàng cần là gọi những builder khác nhau – quá trình xây dựng vẫn như cũ. Đây là một ví dụ khác, bạn muốn đọc một đoạn văn bản và xây dựng một tài liệu, nhưng bạn lại không biết định dạng chính xác của nó là RTF, Microsoft Word, hay văn bản đơn giản… Mặc dù quá trình tạo dựng là giống nhau cho từng tài liệu, bạn có thể sử dụng những builder khác nhau để tạo dựa vào kiểu của loại tài liệu.

Nói cách khác, khi khách hàng muốn kiểm soát quá trình tạo dựng, nhưng bạn vẫn muốn có thể tạo ra nhiều đối tượng khác nhau, mẫu Builder sẽ giúp bạn thực hiện điều đó.

Ghi nhớ: Mẫu thiết kế này tương tự với mẫu Factory, nhưng mẫu Factory là trung tâm trong quá trình khởi tạo một bước, chứ không cài đặt nhiều bước như ở đây.

Cho phép khách hàng tạo Robot

Khi bạn sử dụng mẫu Builder, khách hàng sẽ phụ trách quá trình tạo dựng. Khách hàng sử dụng đối tượng xây dựng builder của bạn để làm những gì họ muốn. Để cho phép khách hàng tạo robot thể hiện một loạt các hành động – khởi động, lắp ráp, ngừng… – Tôi tạo ra một giao diện interface Robot Builder hỗ trợ các hàm như sau: addStart, addGetParts, addAssemble, addTest và addStop:

Ví dụ để tạo một robot với các hành động start, test, assemble và sau đó là stop, khách hàng chỉ cần gọi hàm addStart, addTest, addAssemble và addStop của đối tượng xây dựng builder theo đúng trình tự đó. Khi robot đã được tạo xong, khách hàng chỉ cần gọi hàm getRobot của Builder để nhận về một robot mới. Và đối tượng robot mới này có hỗ trợ hàm go, hàm này sẽ thực hiện hàng loạt hành động mà bạn đã tạo dựng trước đó.

Bởi vì bạn có nhiều loại đối tượng builder để tạo nhiều loại robot khác nhau – ví dụ như builder xây dựng robot làm bánh, builder xây dựng robot lắp ráp ô tô – Tôi sẽ tạo một giao diện interface RobotBuilder mà tất cả các builder sẽ hiện thực giao diện này. Đây là những hàm mà các robot builder phải hiện thực, từ hàm addStart tới hàm addStop, kể cả hàm getRobot. Xem mã sau:

```
public interface RobotBuilder
{
  public void addStart();
  public void addGetParts();
  public void addAssemble();
  public void addTest();
  public void addStop();
  public RobotBuildable getRobot();
}
```

Tôi bắt đầu tạo builder cho robot làm bánh, CookieRobotBuilder, cũng giống như tất cả các builder khác, builder này cần phải hiện thực giao diện RobotBuilder

```
public class CookieRobotBuilder implements RobotBuilder
{
    .
    .
    .
}
```

Đối tượng robot trong mã nguồn trên, dựa trên lớp CookieRobotBuildable, sẽ được nói tới trong phần "Tạo một vài robot tương thích". Đối tượng robot được tạo sẽ là một đối tượng CookieRobotBuildalbe. Vì thế chúng ta cần một biến để lưu trữ đối tượng này, mã như sau:

```
public class CookieRobotBuilder implements RobotBuilder
{
  CookieRobotBuildable robot;

  public CookieRobotBuilder()
  {
    robot = new CookieRobotBuildable();
  }
    .
    .
    .
}
```

Khách hàng có thể cài đặt các hành động cho robot như start, stop, test, assemble, getParts … theo trình tự bất kì. Để lưu lại trình tự này, tôi sử dụng kiểu ArrayList, với đối tượng actions như sau:

```
import java.util.*;

public class CookieRobotBuilder implements RobotBuilder
{
  CookieRobotBuildable robot;
  ArrayList<Integer> actions;

  public CookieRobotBuilder()
  {
    robot = new CookieRobotBuildable();
    actions = new ArrayList<Integer>();
  }
    .
    .
    .
}
```

Cách dễ dàng nhất để lưu trữ trình tự các hành động của robot trong mảng danh sách actions là gán từng giá trị số nguyên cho từng hành động, như hình sau:

- start = 1
- getParts = 2
- assemble = 3, and so on

Tôi lưu đối tượng số nguyên trong một mảng danh sách ArrayList. Ví dụ, khi khách hàng muốn thêm hành động start, chương trình gọi hàm addStart, và robot builder sẽ thêm một đối tượng số nguyên có giá trị 1 vào mảng danh sách actions, và cứ thế tiếp tục… Đây là tất cả các hàm để thêm chức năng cho robot trong builder:

```
import java.util.*;

public class CookieRobotBuilder implements RobotBuilder
{
  CookieRobotBuildable robot;
  ArrayList<Integer> actions;

  public CookieRobotBuilder()
  {
    robot = new CookieRobotBuildable();
    actions = new ArrayList<Integer>();
  }

  public void addStart()
  {
    actions.add(new Integer(1));
  }

  public void addGetParts()
  {
    actions.add(new Integer(2));
  }

  public void addAssemble()
  {
    actions.add(new Integer(3));
  }

  public void addTest()
  {
    actions.add(new Integer(4));
  }

  public void addStop()
  {
    actions.add(new Integer(5));
  }
    .
    .
    .
}
```

Khi khách hàng muốn tạo một đối tượng robot, họ sẽ gọi hàm getRobot của builder này. Khi hàm này được gọi, bạn biết rằng quá trình khởi tạo đã hoàn tất, vì vậy bạn có thể cài đặt robot bằng cách chuyển giao cho nó tham số mảng danh sách actions mà nó sẽ thực thi. Trong ví dụ này, từng robot sẽ được cài đặt bằng cách chuyển tham số actions thông qua hàm loadActions. Mã như sau:

```
import java.util.*;

public class CookieRobotBuilder implements RobotBuilder
{
  CookieRobotBuildable robot;
  ArrayList<Integer> actions;

  public CookieRobotBuilder()
  {
    robot = new CookieRobotBuildable();
    actions = new ArrayList<Integer>();
  }

  public void addStart()
  {
    actions.add(new Integer(1));
  }
     .
     .
     .
  public void addStop()
  {
    actions.add(new Integer(5));
  }

  public RobotBuildable getRobot()
  {
    robot.loadActions(actions);
    return robot;
  }

}
```

Vậy là hoàn thành phần đối tượng xây dựng builder, nó cho phép khách hàng tự cài đặt robot theo trình tự họ muốn. Vậy làm sao để tạo lớp Robot mà ta đã sử dụng ở trên?

Tạo một số robot thích hợp:

Từng loại thợ xây builder sẽ tạo ra một loại robot khác nhau, từng robot lại được tạo từ lớp cơ sở của nó, như lớp CookieRobotBuildable hay AutomotiveRobotBuildalble. Tất cả các robot phải có cùng một hàm go để thực hiện các chức năng. Vì vậy bạn có tạo một giao diện interface với tên RobotBuildable để chắc chắn rằng mọi robot đều phải hiện thực giao diện này. Mã như sau:

```
public interface RobotBuildable
{
  public void go();
}
```

Bây giờ tất cả Robot đều phải hiện thực giao diện này. Đây là cách lớp RobotBuildable hoạt động. Bạn có thể nạp robot với mảng danh sách actions, thông qua hàm loadActions, với tham số actions đã được đối tượng builder tạo trước. Xem mã sau:

```
import java.util.*;

public class CookieRobotBuildable implements RobotBuildable
{
  ArrayList<Integer> actions;

  public CookieRobotBuildable()
  {
  }

  public void loadActions(ArrayList a)
  {
    actions = a;
  }
    .
    .
    .
}
```

Khi khách hàng muốn robot thực hiện các hành động được cài đặt sẵn, họ gọi hàm go. Trong hàm go, bạn có thể duyệt qua mảng actions và gọi từng hàm tương ứng với chức năng đó. Ví dụ bạn duyệt qua đối tượng số nguyên “1” trong actions, bạn sẽ gọi hàm start, duyệt tới số “2” bạn gọi hàm getParts và vân vân..Bạn có thể sử dụng một đối tượng Iterator và phát biểu switch trong hàm go như sau:

```
import java.util.*;

public class CookieRobotBuildable implements RobotBuildable
{
  ArrayList<Integer> actions;

  public CookieRobotBuildable()
  {
  }

  public final void go()
  {
    Iterator itr = actions.iterator();
    while(itr.hasNext()) {
      switch ((Integer)itr.next()){
        case 1:
          start();
          break;
        case 2:
          getParts();
          break;
        case 3:
          assemble();
          break;
        case 4:
          test();
          break;
        case 5:
          stop();
          break;
      }
    }
  }
    .
    .
    .
  public void loadActions(ArrayList a)
  {
    actions = a;
  }
}
```

Ghi chú: Bạn cũng cần phải thêm các hàm cho từng hành động như : hàm start (hiển thị chữ “Starting…”), hàm getParts (hiển thị chữ Getting flour and sugar…) vân vân.

```
import java.util.*;

public class CookieRobotBuildable implements RobotBuildable
{
  ArrayList<Integer> actions;

  public CookieRobotBuildable()
  {
  }

  public final void go()
  {
    Iterator itr = actions.iterator();

    while(itr.hasNext()) {
      switch ((Integer)itr.next()){
        case 1:
          start();
          break;
        case 2:
          getParts();
          break;
        case 3:
          assemble();
          break;
        case 4:
          test();
          break;
        case 5:
          stop();
          break;
      }
    }
  }

  public void start()
  {
    System.out.println("Starting....");
  }

  public void getParts()
  {
    System.out.println("Getting flour and sugar....");
  }

  public void assemble()
  {
    System.out.println("Baking a cookie....");
  }

  public void test()
  {
    System.out.println("Crunching a cookie....");
  }

  public void stop()
  {
    System.out.println("Stopping....");
  }

  public void loadActions(ArrayList a)
  {
    actions = a;
  }
```

Vậy là hoàn tất lớp CookieRobotBuilable. Giờ bạn đã có đối tượng xây dựng Builder và robot. Giờ là lúc để thử nghiệm chúng

```
import java.io.*;

public class TestRobotBuilder
{
  public static void main(String args[])
  {
    String response = "a";

    System.out.print(
      "Do you want a cookie robot [c] or an automotive one [a]? ");
    BufferedReader reader = new
      BufferedReader(new InputStreamReader(System.in));

    try{
      response = reader.readLine();
    } catch (IOException e){
      System.err.println("Error");
    }
      .
      .
      .
}
```

Tùy thuộc vào loại robot mà user chọn lựa, đối tượng builder dành cho robot làm bánh hay builder dành cho robot lắp ráp ô tô sẽ được tạo ra. Xem mã sau:

```
import java.io.*;

public class TestRobotBuilder
{
  public static void main(String args[])
  {
    RobotBuilder builder;
    String response = "a";

    System.out.print(
      "Do you want a cookie robot [c] or an automotive one [a]? ");
    BufferedReader reader = new
      BufferedReader(new InputStreamReader(System.in));

    try{
      response = reader.readLine();
    } catch (IOException e){
      System.err.println("Error");
    }

    if (response.equals("c")){
      builder = new CookieRobotBuilder();
    } else {
      builder = new AutomotiveRobotBuilder();
    }
      .
      .
      .
  }
}
```

Sau đó khách hàng tạo loại robot mà họ muốn, và sử dụng các hàm addStart, addGetParts, addAssemble, addTest và addStop theo trình tự họ muốn

```
import java.io.*;

public class TestRobotBuilder
{
  public static void main(String args[])
  {
    RobotBuilder builder;
    String response = "a";

    System.out.print(
      "Do you want a cookie robot [c] or an automotive one [a]? ");
    BufferedReader reader = new
      BufferedReader(new InputStreamReader(System.in));
    try{
      response = reader.readLine();
    } catch (IOException e){
      System.err.println("Error");
    }

    if (response.equals("c")){
      builder = new CookieRobotBuilder();
    } else {
      builder = new AutomotiveRobotBuilder();
    }

    //Start the construction process.

    builder.addStart();
    builder.addTest();
    builder.addAssemble();
    builder.addStop();
        .
        .
        .
  }
}
```

Sau khi robot được tạo, khách hàng gọi hàm getRobot, đối tượng robot trả về được lưu trong biến RobotBuildable. Và bạn có thể gọi hàm go của robot. Mã như sau:

```
import java.io.*;

public class TestRobotBuilder
{
  public static void main(String args[])
  {
    RobotBuilder builder;
    RobotBuildable robot;
    String response = "a";

    System.out.print(
      "Do you want a cookie robot [c] or an automotive one [a]? ");
    BufferedReader reader = new
      BufferedReader(new InputStreamReader(System.in));

    try{
      response = reader.readLine();
    } catch (IOException e){
      System.err.println("Error");
    }
    if (response.equals("c")){
      builder = new CookieRobotBuilder();
    } else {
      builder = new AutomotiveRobotBuilder();
    }

    //Start the construction process.

    builder.addStart();
    builder.addTest();
    builder.addAssemble();
    builder.addStop();

    robot = builder.getRobot();

    robot.go();
  }
}
```

Khách hàng có thể tạo robot làm bánh hay robot lắp ráp ô tô một cách đơn giản thông qua việc chọn đúng builder. Đây là kết quả:

```
Do you want a cookie robot [c] or an automotive one [a]? c
Starting....
Crunching a cookie....
Baking a cookie....
Stopping....
```

Và đây là kết qủa việc tạo robot lắp ráp ôtô, sử dụng cùng một quy trình khởi tạo:

```
Do you want a cookie robot [c] or an automotive one [a]? a
Starting....
Revving the engine....
Installing the carburetor....
Stopping....
```

Tuyệt vời. Bạn có thể đưa builder cho khách hàng, giúp khách hàng có thể kiểm soát quá trình tạo dựng đối tượng.

DP4Dummies – Chương 8: Iterator, Composite

CHƯƠNG VIII:XỬ LÝ TẬP HỢP VỚI MẪU ITERATOR VÀ MẪU COMPOSITE

Trong chương này:

- Sử dụng mẫu Iterator (đối tượng lặp lại)
– Tạo một đối tượng Iterator
– Duyệt qua danh sách các phó giám đốc bằng một Iterator nội tại
– Hiểu được mẫu Composite (tập hợp)
– Sử dụng một Iterator bên trong một Composite

Giám đốc điều hành của công ty GianDataPool, một công ty mà bạn mới chuyển đến với vị trí tư vấn, vừa đi khẽ vào phòng làm việc của bạn và nói lầm bầm gì đó.

“Gì vậy?” bạn hỏi.

Vị giám đốc nhìn quanh với vẻ mặt bí mật, và nói “Tôi có một dự án tuyệt mật dành cho bạn”

“Tuyệt mật?” bạn nói “Nó nói về cái gì?”

“Đừng to tiếng!” vị giám đốc nói khẽ. “Chúng ta cần một người khách quan cho chuyện này, Vì vậy tôi mới gặp anh. Chúng ta dường như đang gặp phải một số vấn đề với việc quản trị và chúng ta cần phải theo dõi các phó giám đốc – Không ai được biết việc này. Bây giờ, có thể có hai hay vài vị phó giám đốc làm việc như một lập trình viên vậy”

“Thừa thầy thiếu thợ”, bạn thở dài “Chuyện dài tập của các công ty”

“Chúng ta bắt đầu với khu vực bán hàng,” vị giám đốc nói khẽ “Anh có thể viết một chương trình duyệt qua hết hồ sơ và in chúng ra chứ?”

“Còn hơn thế nữa”, bạn nói. ”Tôi sẽ sử dụng mẫu Iterator”

Chương này nói về hai mẫu có quan hệ mật thiết với nhau: mẫu Iterator và mẫu Composite. Mẫu Iterator cung cấp cho bạn cách thức truy cập một bộ phận bên trong một đối tượng mà không cần phải hiểu rõ cấu trúc nội tại của đối tượng đó. Ví dụ, hãng Sun đã giới thiệu một kiểu tập hợp trong việc biểu diễn các mối quan hệ trong ngôn ngữ Java, những tập hợp này cho phép bạn tạo iterator – một đối tượng đặc biệt được thiết kế cho phép bạn truy cập một phần tử của tập hợp – để cung cấp một cách thức truy cập dễ dàng.

Mẫu Composite cũng nói về tập hợp. Với mẫu Composite, ý tưởng là bạn có thể một cấu trúc hình cây nơi mà từng đối tượng sẽ thuộc về một cái cây -là một nút lá không có nút con, hoặc là một nhánh cây với nhiều nút lá con – để có thể xử lý trong cùng một cách. Mẫu Composite được thiết kế cho phép bạn xử lý nhiều đối tượng khác chủng loại trong cùng một tập hợp theo cùng một cách, và một đối tượng lặp iterator lại vô tình phù hợp tại đây – dùng để xử lý từng phần tử của một nhánh cây – ví dụ, bạn có thể duyệt qua hết cây. Chúng ta sẽ thảo luận về hai mẫu trong chương này.

Truy cập đối tượng với mẫu Iterator

Khi bạn làm việc với một tập hợp nhiều đối tượng, mẫu Iterator là một giải pháp tốt. Hàng ngày, bạn phải làm việc với nhiều loại tập hợp như cấu trúc cây, cây nhị phân, mảng, vòng đệm, bảng băm, danh sách mảng và vân vân… Cách thức mà tập hợp này lưu trữ đối tượng của nó rất khác nhau, và nếu bạn muốn truy cập dữ liệu của những đối tượng này, bạn phải học những kỹ thuật khác nhau cho từng loại tập hợp.

Và đó là nơi mẫu Iterator xuất hiện. Bạn có thể sử dụng một giao diện interface được xác định rõ ràng để truy cập tới từng phần tử của tập hợp. Trong những năm qua, các phương pháp cơ bản đã dần trở nên thích hợp hơn, và chúng cũng xuất hiện xuyên suốt chương này. Sử dụng những phương pháp này, bạn có thể truy xuất tới các phần tử trong tập hợp theo cách cơ bản nhất.

Ghi nhớ: Theo sách của Gang of Four (Gof), bạn có thể sử dụng mẫu thiết kế Iterator để “Cung cấp một cách thức truy cập tuần tự tới các phần tử của một đối tượng tổng

hợp, mà không cần phải tạo dựng riêng các phương pháp truy cập cho đối tượng tổng hợp này"

Nói cách khác, một Iterator được thiết kế cho phép bạn xử lý nhiều loại tập hợp khác nhau bằng cách truy cập những phần tử của tập hợp với cùng một phương pháp, cùng một cách thức định sẵn, mà không cần phải hiểu rõ về những chi tiết bên trong của những tập hợp này.

Gợi ý: Mẫu thiết kế Iterator đặc biệt quan trọng khi tập hợp bạn đang xây dựng được tạo thành từ những tập hợp con riêng rẽ, ví dụ khi bạn chỉnh sửa bảng băm với danh sách mảng, chẳng hạn.

Thông tin: Iterator thường được viết trong Java như là những lớp độc lập. Tại sao những Iterator có thể làm việc được trong các tập hợp khác nhau? Chúng có thể, nhưng trong Java, còn ngôn ngữ khác, chúng không thể. Ý tưởng thiết kế này là một trong những kỹ thuật được gọi là "đơn trách nhiệm" – một lớp chỉ có duy nhất một công việc để làm. Hãy suy nghĩ rằng tập hợp duy trì các phần tử, một iterator cung cấp cách thức làm việc với các phần tử đó. Tách biệt trách nhiệm giữa các lớp rất hữu dụng khi một lớp bị thay đổi – Nếu có quá nhiều thứ bên trong một lớp đơn lẻ, sẽ rất khó khăn để viết lại mã nguồn. Khi diễn ra sự thay đổi, một lớp "đơn trách nhiệm" sẽ chỉ có một lý do duy nhất để thay đổi.

Truy cập đối tượng của bạn với một Iterator

Bạn bắt đầu làm việc với rắc rối giám đốc, đó là phải theo dõi các phó giám đốc. Trong trường hợp này, bạn quyết định lưu các phó giám đốc vào trong một tập hợp, với một tập hợp các chức năng cho phép truy xuất các vị này. Trong phiên bản đầu tiên này, các chức năng cơ bản mà một Iterator phải có như sau:

- was
- first
- next
- isDone
- currentItem

Ngày này Java đã hỗ trợ một giao diện iterator trong java.util.Iterator, được định nghĩa với ba phương pháp sau:

- next
- hasNext
- remove

Hàm next trả về phần tử kế tiếp trong tập hợp, hàm hasNext trả về giá trị True nếu vẫn còn phần tử trong tập hợp và trả về false trong trường hợp ngược lại, hàm remove cho phép bạn gỡ bỏ một phần tử trong tập hợp.

Đó là cách Iterator làm việc – Nó cung cấp một giao diện đơn giản, nhất quán để làm việc với các tập hợp khác nhau. Giả sử rằng khách hàng phải làm việc với một tập hợp phức tạp và rắc rối (như hình sau) và không biết cách thức làm việc với nó như thế nào.

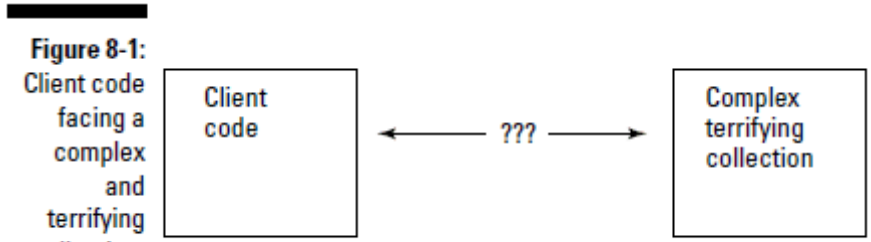


Figure 8-1: Client code facing a complex and terrifying collection.

Khách hàng có thể sử dụng iterator để làm cầu nối với tập hợp, và khách hàng có thể sử dụng các phương thức cơ bản của Iterator để giao tiếp với tập hợp. Như hình sau:

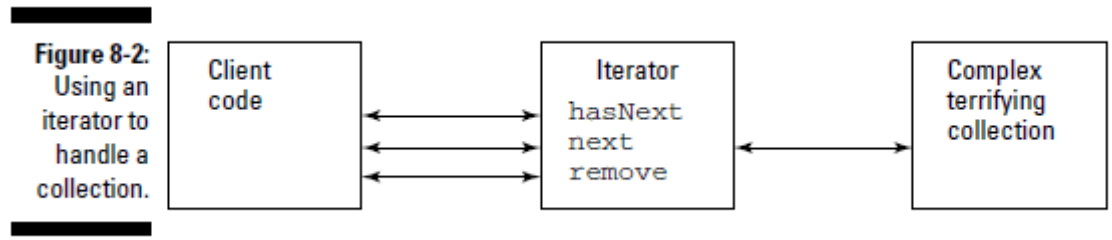


Figure 8-2: Using an iterator to handle a collection.

Công việc đầu tiên khi lưu trữ dữ liệu các phó giám đốc cũng giống cách trên. Bạn quyết định, đầu tiên là tạo một lớp lưu trữ thông tin cho từng phó giám đốc, với tên lớp VP (Vice President – phó giám đốc, phó chủ tịch..)

Bạn phải tạo bốn thành phần quan trọng trong lớp này, bao gồm:

- Một hàm khởi dựng cho phép truyền giá trị tên của vị phó này
– Tên khu vựa làm việc của vị phó
– Hàm getName trả về tên của người này
– Hàm print cho phép in ra thông tin của vị phó này, bao gồm tên và khu vực làm việc

```
public class VP
{
  private String name;
  private String division;

  public VP(String n, String d)
  {
    name = n;
    division = d;
  }

  public String getName()
  {
    return name;
  }

  public void print()
  {
    System.out.println("Name: " + name + " Division: " + division);
  }

}
```

Lớp này đã đóng gói thông tin một phó giám đốc. Bây giờ ta phải lưu trữ tất cả giám đốc trong một lớp.

Thu thập các phó giám đốc vào một tập hợp:

Trong ví dụ này, bạn tạo tập hợp các phó giám đốc dựa trên mảng căn bản của Java. Lí do dùng kiểu căn bản này, thay vì dùng các chức năng có sẵn trong Java như vector, danh sách mảng, bản đồ băm … với phần tử Iterator có sẵn, đó là việc tạo Iterator từ đầu để làm việc với tập hợp thì hơi ngớ ngẩn, nhưng rất tốt để hiểu về mẫu này.

Bạn quyết định lưu thông tin các phó giám đốc trong từng khu vực, ví dụ khu vực bán hàng Sales, trong lớp tên là Division

```
public class Division
{
    .
    .
    .
}
```

Hàm khởi dựng của lớp Division sẽ lưu trữ tên của khu vực này, ví dụ Sales, và hàm getNames sẽ trả về tên đó

```
public class Division
{
  private String name;

  public Division(String n)
  {
    name = n;
  }

  public String getName()
  {
    return name;
  }
    .
    .
    .
}
```

Các phó giám đốc sẽ được lưu trong một mảng, tên là vPs, và bạn có thể thêm một phó giám đốc bằng hàm add như sau:

```
public class Division
{
  private VP[] VPs = new VP[100];
  private int number = 0;
  private String name;

  public Division(String n)
  {
    name = n;
  }

  public String getName()
  {
    return name;
  }

  public void add(String n)
  {
    VP vp = new VP(n, name);
    VPs[number++] = vp;
  }
    .
    .
    .
}
```

Nói cách khác, đối tượng Division là một tập hợp, và đối tượng phó giám đốc VP là một phần tử của tập hợp này. Để thêm một iterator, tập hợp cần phải có một hàm – tên bạn có thể đặt tùy ý – ví dụ như iterator chẳng hạn (có thể tên bao gồm việc tạo createIterator và việc nhận getIterator). Hàm này sẽ chuyển mảng các phó giám đốc vào hàm khởi dựng của lớp iterator, ta gọi tên lớp này là lớp DivisionIterator. Mã như sau:

```
public class Division
{
  private VP[] VPs = new VP[100];
  private int number = 0;
  private String name;

  public Division(String n)
  {
    name = n;
  }

  public String getName()
  {
    return name;
  }

  public void add(String n)
  {
    VP vp = new VP(n, name);
    VPs[number++] = vp;
  }

  public DivisionIterator iterator()
  {
    return new DivisionIterator(VPs);
  }

}
```

Bước tiếp theo là tạo iterator, lớp DivisionIterator, cho phép bạn lặp xuyên qua tập hợp các phó giám đốc trong tập hợp.

Tạo lớp Iterator

Lớp iterator, DivisionIterator, hiện thực ba hàm trong giao diện java.util.Iterator : hàm nex, hàm hasNext, và hàm remove. Mã như sau:

```
import java.util.Iterator;

public class DivisionIterator implements Iterator
{
    .
    .
    .
}
```

Hàm khởi dựng chấp nhận một mảng các phần tử VP và lưu trữ lại như sau:

```
import java.util.Iterator;

public class DivisionIterator implements Iterator
{
  private VP[] VPs;

  public DivisionIterator(VP[] v)
  {
    VPs = v;
  }
    .
    .
    .
}
```

Bây giờ bạn phải hiện thực các giao diện của Iterator. Hàm next trả về phần tử kế tiếp trong mảng. Mã như sau:

```
import java.util.Iterator;

public class DivisionIterator implements Iterator
{
  private VP[] VPs;
  private int location = 0;

  public DivisionIterator(VP[] v)
  {
    VPs = v;
  }

  public VP next()
  {
    return VPs[location++];
  }
    .
    .
    .
}
```

Hàm hasNext trả về true nếu có phần tử kế tiếp trong tập hợp, ngược lại trả về false. Trong trường hợp này, bạn phải kiểm trả đã ở cuối của dãy chưa? Bởi vì bạn đang làm việc với một mảng cố định, bạn cũng phải kiểm tra nếu phần tử kế tiếp là phần tử trống (null) – và bạn cũng phải kiểm tra xem mảng có phải là rỗng hay không. Hàm hasNext như sau:

```
import java.util.Iterator;

public class DivisionIterator implements Iterator
{
  private VP[] VPs;
  private int location = 0;

  public DivisionIterator(VP[] v)
  {
    VPs = v;
  }

  public VP next()
  {
    return VPs[location++];
  }

  public boolean hasNext()
  {
    if(location < VPs.length && VPs[location] != null){
      return true;
    } else {
      return false;
    }
  }
    .
    .
    .
}
```

Hiện tại bạn muốn mảng phó giám đốc này chỉ độc, bạn tiếp tục hiện thực hàm remove với nội dung rỗng như sau:

```
import java.util.Iterator;

public class DivisionIterator implements Iterator
{
  private VP[] VPs;
  private int location = 0;

  public DivisionIterator(VP[] v)
  {
    VPs = v;
  }

  public VP next()
  {
    return VPs[location++];
  }

  public boolean hasNext()
  {
    if(location < VPs.length && VPs[location] != null){
      return true;
    } else {
      return false;
    }
  }

  public void remove()
  {
  }
}
```

Tuyệt vời. Bạn đã có đối tượng phó giám đốc, một khu vực thể hiện như một tập hợp các phó giám đốc, và một đối tượng lặp Iterator. Việc cuối cùng là đưa tất cả chúng vào một chương trình và bắt đầu lặp qua các phó giám dốc

Lặp qua các phó giám đốc

Xem mã sau:

```
public class TestDivision
{
  public static void main(String args[])
  {
    TestDivision d = new TestDivision();
  }

  public TestDivision()
  {
    .
    .
    .
}
```

Mã nguồn bắt đầu từ việc tạo khu vực bán hàng Sales và thêm vào một vài vị giám đốc:

```
public class TestDivision
{
  Division division;

  public static void main(String args[])
  {
    TestDivision d = new TestDivision();
  }

  public TestDivision()
  {
    division = new Division("Sales");

    division.add("Ted");
    division.add("Bob");
    division.add("Carol");
    division.add("Alice");
      .
      .
      .
  }
}
```

Sau đó ta tạo một iterator bằng cách gọi hàm iterator và sử dụng các hàm hasNext, next để duyệt qua từng phó giám đốc trong tập hợp và hiển thị thông tin từng người một.

```
public class TestDivision
{
  Division division;
  DivisionIterator iterator;

  public static void main(String args[])
  {
    TestDivision d = new TestDivision();
  }

  public TestDivision()
  {
    division = new Division("Sales");

    division.add("Ted");
    division.add("Bob");
    division.add("Carol");
    division.add("Alice");
    iterator = division.iterator();

    while (iterator.hasNext()){
      VP vp = iterator.next();
      vp.print();
    }
  }
}
```

Kết quả là, chương trình in ra toàn bộ thông tin các phó giám đốc:

```
Name: Ted Division: Sales
Name: Bob Division: Sales
Name: Carol Division: Sales
Name: Alice Division: Sales
```

Đặt mọi thứ vào trong tập hợp composites

Giám đốc của GianDataPool Inc, chạy ào vào văn phòng bạn với vẻ đắc thắng và nói lớn: “Tôi muốn sa thải một vài phó giám đốc!”

“Tốt,” bạn nói.

“Tôi muốn làm thêm nữa. Bây giờ tôi cần in ra tất cả thông tin phó giám đốc của toàn bộ công ty – không chỉ khu vực bán hàng, mà là toàn bộ các khu vực.”

“Tất cả các khu vực?” bạn hỏi.

“Vâng. Và cả các phó giám đốc hoạt động độc lập, không trực thuộc vào một khu vực nào ”.

“Hmm”, bạn nói ,”Đã đến lúc sử dụng một mẫu thiết kế mới”.

“Đợi đã”, giám đốc nói “Nhớ kỹ rằng đây là một vụ cắt giảm chi phí đó”

“Tôi sẽ sử dụng mẫu tổng hợp composites”, bạn nói.

“Có tốn nhiều chi phí không?”

“Không” bạn nói. “nhưng tôi phải làm nhiều thôi”

Bạn đã hiểu rõ rắc rối. bây giờ bạn phải xử lý toàn bộ công ty, không chỉ là một phân khu. Toàn bộ công ty có nhiều khu vực với các phó giám đốc, và khu vực này có thể bao gồm cả khu vực khác – và bao gồm cả các phó giám đốc tự do nữa. Hình sau chỉ ra mô hình công ty:

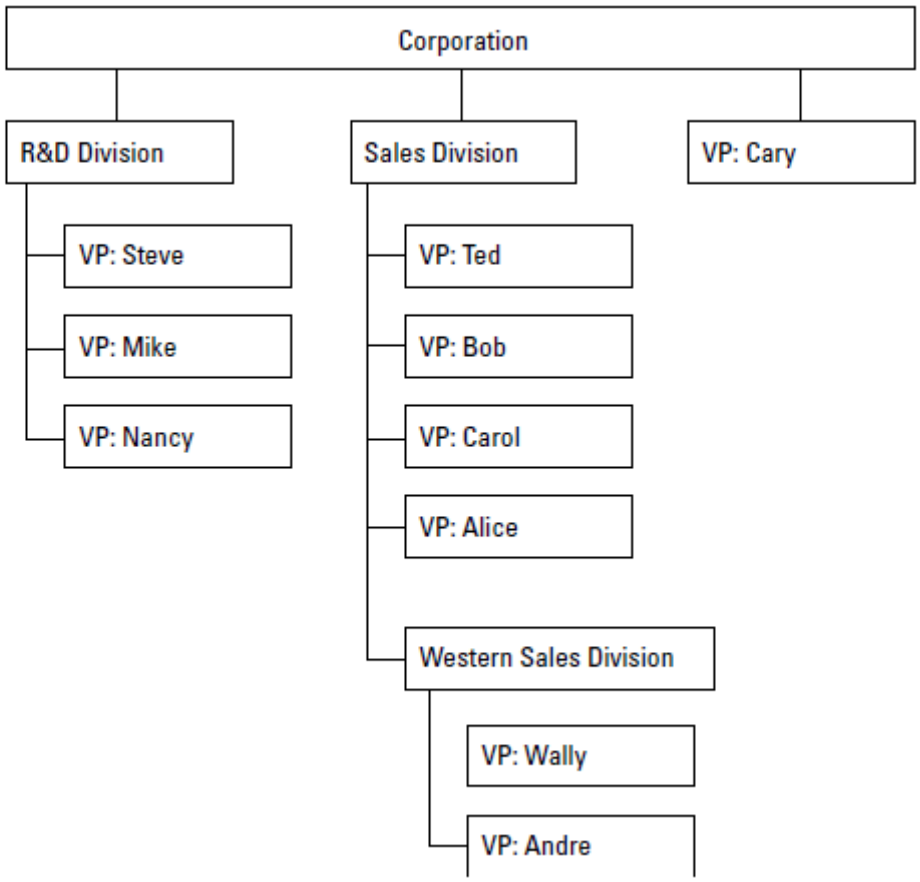


Figure 8-3: All the parts of the corporation.

Vì vậy, giờ đây bạn đang làm việc với một tổ chức phức tạp, không chỉ là một khu vực bán hàng Sales nữa. Và giám đốc điều hành muốn bạn in ra toàn bộ công ty, vì vậy bạn

không chỉ cần hàm print của đối tượng VP, mà từng khu vực phải có một hàm print riêng. OK, đã đến lúc sử dụng mẫu tổng hợp Composite.

Bạn muốn có một hàm print, mà khi được gọi, nó sẽ in ra thông tin của một phó giám đốc, một phòng ban, hoặc cả tổ chức. Mẫu Composites là mẫu nói về việc tạo ra một cấu trúc hình cây nơi mà từng lá trong cây, có thể được sử dụng trong cùng một cách với nhánh của nó (nhánh là cấu trúc chứa nhiều lá, và giống các nhánh khác). Ý tưởng chính ở đây là, để làm mọi chuyện dễ dàng, bạn có thể xử lý các nút lá và tập hợp các nút lá trong một cái cây theo cùng một cách.

Ghi nhớ: Sách của GoF nói rằng, bạn sử dụng mẫu Composites để "Tạo ra các đối tượng trong một cấu trúc hình cây để biểu diễn cho một cấu trúc phân cấp. Mẫu Composites cho phép khách hàng xử lý một đối tượng riêng hoặc toàn bộ đối tượng theo cùng một cách"

Đó là những gì bạn cần – một mẫu thiết kế cho phép bạn xử lý các nút lá hoặc các nhánh của cấu trúc cây theo cách giống nhau bởi vì bạn muốn có thể in ra thông tất cả các phó giám đốc riêng lẻ, trong một khu vực, hoặc cả công ty, chỉ bằng cách gọi hàm print.

Mẫu thiết kế Composites rất phù hợp với mẫu Iterator bởi vì khi bạn gọi từng khu vực để in chính nó, nó có thể dễ dàng duyệt qua từng phó giám đốc một. Đó là đặc điểm điển hình của mẫu Composite – khi bạn yêu cầu một nhánh thực hiện một hành động gì đó , nó sẽ lặp qua tất cả các lá con và nhánh con của nó.

Ý tưởng đằng sau của mẫu Composite là việc xử lý các nút lá và nhánh trong một cấu trúc hình cây sẽ giống nhau. Điều này giúp cho việc xử lý các cấu trúc phức tạp theo dạng hình cây sẽ dễ dàng hơn bởi vì bạn không cần phải thiết lập các hàm khác nhau cho từng phần của cấu trúc.

Để thực hiện mẫu Composite, sách của GoF khuyên rằng bạn nên sử một lớp trừu tượng như là một lớp cơ sở cho cả nút lá và các nhánh trong cấu trúc cây. Việc làm này giúp cho các nút lá và các nhánh sẽ có chung một tập hợp các hàm, đó là tất cả những gì mẫu Composite muốn nói tới. Sách của GoF đề nghị bạn sử dụng một lớp trừu tượng, tuy nhiên bạn cũng có thể sử dụng một giao diện interface để làm việc này trong Java.

Tất cả bắt đầu với một lớp trừu tượng

Tôi sẽ theo chỉ dẫn của sách GoF và tạo một lớp trừu tượng cho cả phó giám đốc cũng như khu vực, lớp này tên Corporate. Bên dưới là mã nguồn của lớp này. Chú ý nó cũng có hàm add, và hàm iterator để trả về một iterator, và một hàm print:

```
import java.util.*;

public abstract class Corporate
{
  public String getName()
  {
    return "";
  }

  public void add(Corporate c)
  {
  }
  public Iterator iterator()
  {
    return null;
  }

  public void print()
  {
  }
}
```

Đây là lớp dùng để kế thừa cho cả các nút lá phó giám đốc và các nhánh cây khu vực.

Tạo nút lá phó giám đốc

Lớp VP bạn tạo trước đây phải chỉnh sửa một chút, để bạn có thể thống nhất cách làm việc với cả phó giám đốc và khu vực trong cùng một cây tổ chức, theo cách mẫu Composite đã nói. Đặc biệt, bạn phải kế thừa lớp VP từ lớp trừu tượng Corporate mà bạn đã tạo trong phần trên

```
import java.util.*;

public class VP extends Corporate
{
     .
     .
     .
}
```

Lớp VP trước đây chỉ chứa tên và khu vực làm việc của phó giám đốc và hàm print để in ra thông tin này. Nhưng để khách hàng có thể xử lý lớp VP cùng cách với các khu

vực division, bạn cần thêm một hàm tạo iterator cho nó. Bởi vì một phó giám đốc không chứa bất cứ phó giám đốc nào, nên iterator được tạo ra chỉ tạo trả về một đối tượng phó giám đốc duy nhất khi bạn gọi hàm next và hàm hasNext luôn trả về giá trị sai false. Mã như sau:

```
import java.util.*;

public class VP extends Corporate
{
  private String name;
  private String division;

  public VP(String n, String d)
  {
    name = n;
    division = d;
  }

  public String getName()
  {
    return name;
  }

  public void print()
  {
    System.out.println("Name: " + name + " Division: " + division);
  }

  public Iterator iterator()
  {
    return new VPIterator(this);
  }

}
```

Lớp VPIterator sẽ như thế nào? Rất dễ dàng, bạn chỉ cần hiện thực giao diện Iterator của Java, đưa vào đối tượng VP thông qua hàm khởi dựng, tạo hàm next trả về đối tượng đó và hàm hasNext trả về giá trị false, như mã sau:

```
import java.util.Iterator;

public class VPIterator implements Iterator
{
  private VP vp;

  public VPIterator(VP v)
  {
    vp = v;
  }

  public VP next()
  {
    return vp;
  }

  public boolean hasNext()
  {
    return false;
  }

  public void remove()
  {
  }
}
```

Bây giờ khách hàng có thể xử lý nút lá phó giám đốc giống như một nhánh cây khu vực. Thực tế là iterator nút lá phó giám đốc chỉ trả về duy nhất một phó giám đốc, nhưng bây giờ bạn đã có một iterator cho từng nút lá, bạn không phải chỉnh sửa mã nguồn để có thể vừa làm việc với nút lá vừa làm việc với các phân khu.

Tạo một nhánh cây các khu vực

Từng nhánh cây trong cấu trúc cây công ty là một khu vực trong công ty, mà có thể bao gồm nhiều phó giám đốc hoặc khu vực con. Để xử lý khu vực, bạn quyết định chỉnh sửa lớp Division (đã tạo trước đây) theo cách mở rộng từ lớp Corporate, như cách đã làm với lớp VP, mã nguồn như sau:

```
import java.util.*;

public class Division extends Corporate
{
    .
    .
    .
}
```

Phần còn lại của lớp Division sẽ giống như trước, ngoài trừ bạn phải chuyển đổi một chút để phù với đối tượng Corporate. Trước đây, lớp Division lưu trữ một mảng VPs các phó giám đốc bởi vì bạn chỉ làm việc với một khu vực của công ty. Bây giờ bạn phải làm việc với cả công ty, một khu vực có thể chứa một khu vực con như là những mảng các phó giám đốc VPs. Từ khi cả hai lớp divison và mảng VPs kế thừa từ lớp Corparate, bạn có thể dễ dàng hóan chuyển để lưu trữ và làm việc với đối tượng Corparate trong lớp Division – chú ý rằng hàm print sẽ duyệt qua tất cả các đối tượng trong một khu vực, cho dù chúng là mảng VPs hay khu vực.

```
import java.util.*;

public class Division extends Corporate
{
  private Corporate[] corporate = new Corporate[100];
  private int number = 0;
  private String name;

  public Division(String n)
  {
    name = n;
  }
  public String getName()
  {
    return name;
  }

  public void add(Corporate c)
  {
    corporate[number++] = c;
  }

  public Iterator iterator()
  {
    return new DivisionIterator(corporate);
  }

  public void print()
  {
    Iterator iterator = iterator();

    while (iterator.hasNext()){
      Corporate c = (Corporate) iterator.next();
      c.print();
    }
  }
}
```

Bằng cách chuyển đổi từ việc xử lý lớp VP bên trong một khu vực division sang việc xử lý một lớp Corporate, giờ đây bạn có thể lưu trữ mảng VPs và khác khu vực khác – và

vì vậy bạn đã hiện thực được mẫu Composite, cho phép bạn có thể xử lý các nút lá hay nhánh cây theo cùng một cách.

Iterator của lớp division, được hiện thực từ lớp DivisionIterator, mã như sau:

```
import java.util.Iterator;

public class DivisionIterator implements Iterator
{
  private Corporate[] corporate;
  private int location = 0;

  public DivisionIterator(Corporate[] c)
  {
    corporate = c;
  }
  public Corporate next()
  {
    return corporate[location++];
  }

  public boolean hasNext()
  {
    if(location < corporate.length && corporate[location] != null){
      return true;
    } else {
      return false;
    }
  }

  public void remove()
  {
  }
}
```

Bạn ngả người ra trong ghế với nụ cười hài lòng trên môi, thầm cảm ơn mẫu thiết kế Composite. Để làm một sự chuyển đổi từ ví dụ một khu vực trong phần đầu của chương này, sang một một cấu trúc hình cây cho toàn bộ công ty, tất cả những gì bạn phải làm là bảo đảm rằng tất cả các đối tượng trong cùng một cây phải dựa trên cùng một lớp, và hiện thực cùng một tập hợp các hàm, cho phép chúng được sử dụng theo cùng một cách.

Xây dựng công ty của bạn

Bạn đã có phó giám đốc; bạn đã có các khu vực. Bây giờ là lúc bạn xây dựng một công ty để chứa chúng. Để giữa cho mọi việc đơn giản, bạn có thể sử dụng một ArrayList để lưu trữ các khu vực divisions và các phó giám đốc trong công ty. Tất cả các đối tượng

trong công ty đều là đối tượng Corporate, vì vậy ArrayList sẽ lưu trữ các đối tượng Corporate.

```
import java.util.*;

public class Corporation extends Corporate
{
  private ArrayList<Corporate> corporate = new ArrayList<Corporate>();

  public Corporation()
  {
  }
    .
    .
    .
}
```

Khi bạn muốn thêm một đối tượng Corporate vào cây, chỉ cần sử dụng hàm add của corporate, hàm sẽ thêm một đối tượng mới vào ArrayList.

```
import java.util.*;

public class Corporation extends Corporate
{
  private ArrayList<Corporate> corporate = new ArrayList<Corporate>();

  public Corporation()
  {
  }

  public void add(Corporate c)
  {
    corporate.add(c);
  }
    .
    .
    .
}
```

Muốn in ra thông tin tất cả các đối tượng trong công ty? Chỉ cần gọi hàm print của đối tượng Corporate, khi đó các iterator trong ArrayList sẽ in ra thông tin trong các khu vực và các phó giám đốc của toàn công ty – chú ý rằng khi bạn gọi hàm print của khu vực division, nó sẽ duyệt qua toàn bộ các đối tượng bên trong, và gọi từng hàm print của chúng. Vì vậy khi gọi hàm print từ cấp cao nhất của Corporate, nó sẽ in ra toàn bộ thông tin của công ty.

```
import java.util.*;

public class Corporation extends Corporate
{
  private ArrayList<Corporate> corporate = new ArrayList<Corporate>();

  public Corporation()
  {
  }

  public void add(Corporate c)
  {
    corporate.add(c);
  }
  public void print()
  {
    Iterator iterator = corporate.iterator();

    while (iterator.hasNext()){
      Corporate c = (Corporate) iterator.next();
      c.print();
    }
  }
}
```

OK. Đã đến lúc cho chương trình chạy thử. Đầu tiên bạn tạo một đối tượng Corporation.

```
import java.util.*;

public class TestCorporation
{
  Corporation corporation;

  public static void main(String args[])
  {
    TestCorporation t = new TestCorporation();
  }

  public TestCorporation()
  {
    corporation = new Corporation();
    .
    .
    .
}
```

Sau đó bạn tạo khu vực R&D và tạo ra một vài phó giám đốc

```
import java.util.*;

public class TestCorporation
{
    .
    .
    .
  public TestCorporation()
  {
    corporation = new Corporation();

    Division rnd = new Division("R&D");
    rnd.add(new VP("Steve", "R&D"));
    rnd.add(new VP("Mike", "R&D"));
    rnd.add(new VP("Nancy", "R&D"));
        .
        .
        .
  }
}
```

Tiếp theo, bạn tạo khu vực Sales. Bạn sử dụng hàm add để thêm không chỉ các phó giám đốc mà còn có thể thêm cả các khu vực con, ví dụ khu vực Western Sales, với một số phó giám đốc

```
import java.util.*;

public class TestCorporation
{
    .
    .
    .
  public TestCorporation()
  {
    corporation = new Corporation();

    Division rnd = new Division("R&D");
    rnd.add(new VP("Steve", "R&D"));
    rnd.add(new VP("Mike", "R&D"));
    rnd.add(new VP("Nancy", "R&D"));

    Division sales = new Division("Sales");

    sales.add(new VP("Ted", "Sales"));
    sales.add(new VP("Bob", "Sales"));
    sales.add(new VP("Carol", "Sales"));
    sales.add(new VP("Alice", "Sales"));

    Division western = new Division("Western Sales");
    western.add(new VP("Wally", "Western Sales"));
    western.add(new VP("Andre", "Western Sales"));

    sales.add(western);
        .
        .
        .
  }
}
```

Và bạn có thể thêm các phó giám đốc vào công ty một cách trực tiếp, cũng giống như cách thêm vào một khu vực, bởi vì bạn có thể xử lý các nút lá và các nhánh con theo cùng một cách. Sau khi tạo một phó giám đốc, bạn có thêm phó giám đốc – và khu vực bạn đã tạo trước – vào công ty và in tất cả thông tin chúng ra với một hàm duy nhất là hàm print của đối tượng corporation, hàm này sẽ gọi hàm print của từng phần tử bên trong nó.

```
import java.util.*;

public class TestCorporation
{
    .
    .
    .
  public TestCorporation()
  {
    corporation = new Corporation();

    Division rnd = new Division("R&D");
    rnd.add(new VP("Steve", "R&D"));
    rnd.add(new VP("Mike", "R&D"));
    rnd.add(new VP("Nancy", "R&D"));

    Division sales = new Division("Sales");

    sales.add(new VP("Ted", "Sales"));
    sales.add(new VP("Bob", "Sales"));
    sales.add(new VP("Carol", "Sales"));
    sales.add(new VP("Alice", "Sales"));

    Division western = new Division("Western Sales");
    western.add(new VP("Wally", "Western Sales"));
    western.add(new VP("Andre", "Western Sales"));

    sales.add(western);

    VP vp = new VP("Cary", "At Large");

    corporation.add(rnd);
    corporation.add(sales);
    corporation.add(vp);

    corporation.print();
  }
}
```

Chạy chương trình, và bạn nhận được kết quả.

```
Name: Steve Division: R&D
Name: Mike Division: R&D
Name: Nancy Division: R&D
Name: Ted Division: Sales
Name: Bob Division: Sales
Name: Carol Division: Sales
Name: Alice Division: Sales
Name: Wally Division: Western Sales
Name: Andre Division: Western Sales
Name: Cary Division: At Large
```

Bạn đưa danh sách thắng lợi này tới cho vị Giám đốc điều hành. “Đã tới lúc cắt tỉa bớt cái cây này”, bạn nói.

“Hả?” Giám đốc hỏi

“Loại bỏ những cành cây đã chết”, bạn nói. Giám đốc cười hạnh phúc.

DP4Dummies – Chương 9: State, Proxy

CHƯƠNG 9: KIỂM SOÁT ĐỐI TƯỢNG CỦA BẠN VỚI MẪU TRẠNG THÁI STATE VÀ MẪU ĐẠI DIỆN PROXY

Trong chương này:

- Sử dụng mẫu State
– Cho phép trạng thái xác định kết quả
– Hiểu về mẫu Proxy
– Sử dụng proxy để đại diện cho đối tượng của bạn
– Kết nối với các proxy thông qua internet

Giám đốc điều hành của công ty Apartments-N-Stuff Inc đã đăng kí dịch vụ tư vấn của bạn và nói “Chúng tôi đang hoạt động cho thuê căn hộ phức hợp trải rộng khắp đất nước”. Vấn đề lớn nhất của chúng tôi là quản lý tài sản, chi phí cho việc này rất lớn. Vì thế chúng tôi đang chuyển đổi tất cả căn hộ phức hợp của chúng tôi sang việc sử dụng các robot phân phối tự động, nó sẽ chấp nhận các đơn thuê nhà và phân phối chìa khóa cho khách hàng. Chúng tôi muốn những người thuê nhà mới sẽ có thể nộp đơn cho hệ thống tự động này, và khi hệ thống chấp nhận, họ sẽ nhận được chìa khóa nhà từ hệ thống.

“Nghe thật tuyệt”, bạn nói

“Ý tưởng chính là,” vị giám đốc nói tiếp “Bình thường, hệ thống tự động này được đặt xung quanh các tòa nhà và chờ đợi các khách hàng tiềm năng. Khi một người thuê nhà nộp đơn, máy tự động này sẽ kiểm tra đơn. Nếu đơn được chấp thuận, máy tự động sẽ phân phối chìa khóa cho người thuê nhà, ngược lại hệ thống sẽ thông báo từ chối người

thuê nhà và trở về trạng thái chờ đợi. Và nếu một hệ thống tự động cho thuê một căn hộ, nó cũng phải kiểm tra để biết chắc rằng còn trống căn hộ nào trong dãy nhà phức hợp không, và sẽ không cho thuê nữa nếu hết căn hộ.

Vị giám đốc điều hành nhìn bạn đang vẽ nguệch ngoạc và cuối cùng hỏi: "Anh đang làm gì đó?"

"Tôi đang vẽ một biểu đồ trạng thái," bạn nói.

"Tôi có thể xem nó không?", Vị giám đốc hỏi

"Không," bạn nói "Ông cần biết thêm nhiều kiến thức để hiểu biểu đồ này"

"Oh", Vị giám đốc nói

Chương này nói về hai mẫu thiết kế: mẫu trạng thái State, nơi mà một đối tượng lưu trữ thông tin về các trạng thái nội tại của nó, và có thể thay đổi hành vi cho thích hợp với trạng thái đó, và mẫu đại diện Proxy, nơi mà một đối tượng có thể hành động như một sự thay thế cho đối tượng khác. Bạn sẽ thấy chi tiết cách hai mẫu làm việc trong chương này.

Ghi nhận tình trạng của bạn với mẫu State

Bạn hiểu những gì vị giám đốc đã nói. Hệ thống tự động cho thuê sẽ có 4 trạng thái sau:

- Waiting for a new tenant
- Receiving an application
- Renting an apartment
- Fully rented

Đó là: chờ một người thuê nhà mới, nhận một đơn thuê nhà, cho thuê một căn hộ, đã hết căn hộ cho thuê.

Hình sau cho bạn thấy biểu đồ trạng thái, nơi mà từng hộp chủ nhật đại diện cho một trạng thái, và bạn có thể kết nối các trạng thái lại với nhau để biết cách làm việc của một máy tự động cho thuê nhà

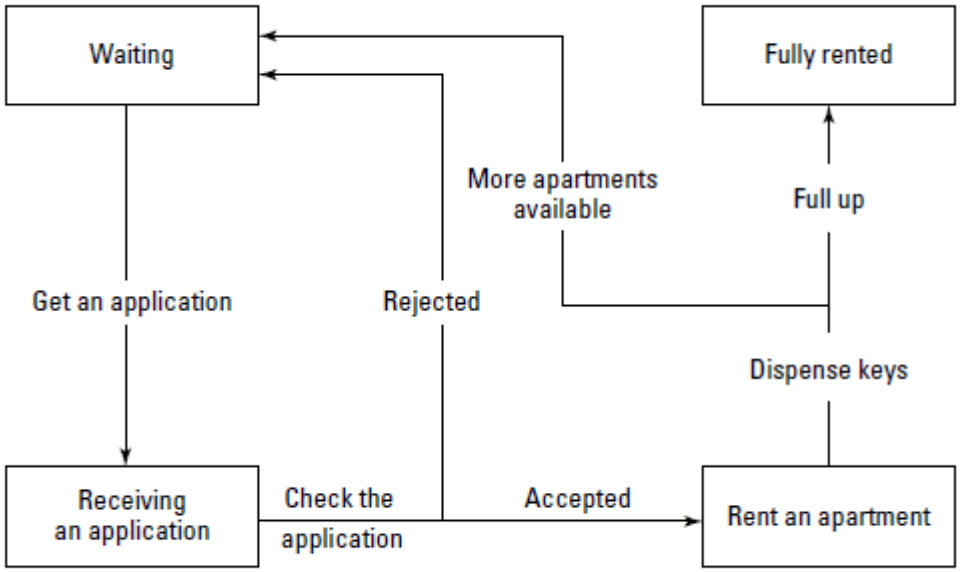


Figure 9-1: The rental automat's states.

Bạn tự nghĩ trong lòng, đây đúng là chỗ để áp dụng mẫu trạng thái State.

Thỉnh thoảng, khi bạn tham gia vào một dự án lớn, mã nguồn thường bắt đầu rất tối tăm. Có quá nhiều điều kiện có thể xảy ra mà bạn phải nắm bắt và rất khó để biết đâu là các ranh giới cũng như làm sao để chia nhỏ mã nguồn ra.

Gợi ý: Khi bạn gặp phải một ứng dụng lớn và mã nguồn không kiểm soát nổi, nó thương giúp cho bạn bắt đầu suy nghĩ tới khái niệm về các trạng thái khác nhau. Đây là công cụ giúp bạn chia nhỏ mã nguồn ra thành từng đơn vị nhỏ hơn, lý tưởng nhất là từng trạng thái này phải độc lập với nhau, và tự động chia nhỏ mã nguồn bạn ra thành từng phần rời rạc.

Ghi nhớ: Sách của Gang o Four (GoF), nói rằng mẫu thiết kế State sẽ "Cho phép một đối tượng thay đổi hành vi khi trạng thái của chính nó thay đổi. Đối tượng sẽ xuất hiện để thay đổi lớp của nó."

Nói cách khác, mã nguồn của bạn theo dõi các trạng thái nội tại, ví dụ như trạng thái chờ một người thuê nhà mới "Waiting for a new tenant". Phần khác của mã nguồn sẽ kiểm tra trạng thái hiện tại là gì và sẽ phản ứng phù hợp. Ví dụ, nếu trạng thái của hệ

thống là hết căn hộ cho thuê Fully rented, và một khách hàng tiềm năng mới tới thuê nhà, khi đưa vào đơn thuê nhà, lập tức hệ thống sẽ từ chối đơn đó.

Khi bạn sử dụng mẫu thiết kế State, một phần mã nguồn có thể kiểm tra trạng thái hiện tại là gì. Điều này giúp bạn làm việc với mã nguồn lớn một cách rõ ràng và tập trung hơn bởi vì bạn phải kiểm soát hoạt động của từng đoạn mã rời rạc, chỉ bằng các cách thay đổi trạng thái hiện tại.

Nói chung, mẫu thiết kế State rất hữu ích khi bạn có nhiều mã nguồn mà ngày càng phức tạp và rối rắm. Nếu bạn có thể chia nhỏ công việc bằng cách đưa chúng vào các trạng thái độc lập với nhau, bạn đã làm công việc của mình dễ dàng hơn rất nhiều.

Sử dụng hàm để lưu trữ trạng thái

Ví dụ đầu tiên, bạn cố gắng tạo một máy tự động đơn giản, sử dụng trạng thái. Bạn quyết định rằng bạn có thể viết ứng dụng tập trung vào một tập hợp các hàm, các hàm này có thể hoạt động dựa trên trạng thái hiện tại của hệ thống. Bạn bắt đầu bằng cách tạo từng hằng số trạng thái cho bốn trạng thái của hệ thống. Mã nguồn như sau:

```
import java.util.*;
import java.lang.Math;

public class RentalMethods
{
  final static int FULLY_RENTED = 0;
  final static int WAITING = 1;
  final static int GOT_APPLICATION = 2;
  final static int APARTMENT_RENTED = 3;
  int state = WAITING;
        .
        .
        .
```

Bây giờ bất cứ mã nguồn nào trong ứng dụng cũng có thể kiểm tra tình trạng hiện tại và đáp ứng thích hợp. Ví dụ mã nguồn sau chỉ ra cách thức hàm getApplication làm việc, hàm được gọi khi hệ thống nhận một đơn thuê nhà của khách hàng – chú ý rằng việc hệ thống xử lý sẽ phụ thuộc vào trạng thái nội tại:

```
import java.util.*;
import java.lang.Math;

public class RentalMethods
{
  final static int FULLY_RENTED = 0;
  final static int WAITING = 1;
  final static int GOT_APPLICATION = 2;
  final static int APARTMENT_RENTED = 3;
  Random random;
  int numberApartments;
  int state = WAITING;

  public RentalMethods(int n)
  {
    numberApartments = n;
    random = new Random(System.currentTimeMillis());
  }

  public void getApplication()
  {
    switch (state)
    {
      case FULLY_RENTED:
        System.out.println("Sorry, we're fully rented.");
        break;
      case WAITING:
        state = GOT_APPLICATION;
        System.out.println("Thanks for the application.");
        break;
      case GOT_APPLICATION:
        System.out.println("We already got your application.");
        break;
      case APARTMENT_RENTED:
        System.out.println("Hang on, we're renting you an apartment.");
        break;
    }
  }
  .
  .
  .
```

Vậy nếu hệ thống nhận được đơn thuê nhà và nó đang ở trạng thái FULLY_RENTED, hệ thống sẽ in ra thông báo xin lỗi, chúng tôi đã cho thuê hết căn hộ. “Sorry, we’re fully rented”. Nếu hệ thống nhận một đơn thuê nhà và trạng thái của nó là WAITTING, nó sẽ thông báo “Thanks for the application”. Và thay đổi hệ thống tự thay đổi trạng thái sang GOT_APPLICATION.

Tương tự vậy, nếu bạn gọi hàm checkApplication của hệ thống, việc đáp trả sẽ tùy thuộc vào tình trạng hiện tại: Ví dụ, nếu hệ thống được yêu cầu kiểm tra một đơn thuê nhà, và nó đang ở tình trạng WAITING, nó sẽ báo với khách hàng là họ cần phải nộp đơn thuê nhà trước. Nếu hệ thống đang ở tình trạng GOT_APPLICATION khi bạn gọi hàm checkApplication, nó sẽ kiểm tra đơn thuê nhà vạ quyết định chấp nhận hay từ chối khách hàng này. Quy trình đó được mô phỏng trong ví dụ này với một số ngẫu nhiên – nếu đơn được chấp nhận, chương trình sẽ đưa hệ thống về trạng thái căn hộ đã cho thuê APARMENT_RENTED và gọi hàm tên rentApartment; nếu đơn bị từ chối, chương trình sẽ đưa hệ thống về trạng thái chờ WAITING.

```
public void checkApplication()
{
  int yesno = random.nextInt() % 10;

  switch (state)
  {
    case FULLY_RENTED:
      System.out.println("Sorry, we're fully rented.");
      break;
    case WAITING:
      System.out.println("You have to submit an application.");
      break;
    case GOT_APPLICATION:
      if (yesno > 4 && numberApartments > 0) {
        System.out.println("Congratulations, you were approved.");
        state = APARTMENT_RENTED;
        rentApartment();
      } else {
        System.out.println("Sorry, you were not approved.");
        state = WAITING;
      }
      break;
    case APARTMENT_RENTED:
      System.out.println("Hang on, we're renting you an apartment.");
      break;
  }
}
```

Hàm rentApartment cũng kiểm tra trạng thái nội tại của hệ thống – nếu trạng thái là APARTMENT_RENTED, nó sẽ giảm số lượng căn hộ xuống 1 đơn vị, và gọi hàm phân phối chìa khóa nhà, hàm dispenseKeys.

```
public void rentApartment()
{
  switch (state)
  {
    case FULLY_RENTED:
      System.out.println("Sorry, we're fully rented.");
      break;
    case WAITING:
      System.out.println("You have to submit an application.");
      break;
    case GOT_APPLICATION:
      System.out.println("You must have your application checked.");
      break;
    case APARTMENT_RENTED:
      System.out.println("Renting you an apartment....");
      numberApartments--;
      dispenseKeys();
      break;
  }
}
```

Cuối cùng, hàm dispenseKeys cũng kiểm tra trạng thái hiện tại và nếu trạng thái là APARTMENT_RENTED, hàm sẽ gởi chìa khóa nhà tới người thuê nhà và đưa hệ thống về trạng thái chờ WAITING để tiếp tục chờ người thuê nhà kế tiếp.

```
public void dispenseKeys()
{
  switch (state)
  {
    case FULLY_RENTED:
      System.out.println("Sorry, we're fully rented.");
      break;
    case WAITING:
      System.out.println("You have to submit an application.");
      break;
    case GOT_APPLICATION:
      System.out.println("You must have your application checked.");
      break;
    case APARTMENT_RENTED:
      System.out.println("Here are your keys!");
      state = WAITING;
      break;
  }
}
```

Ý tưởng chính là như vậy – bạn lưu trữ trạng thái nội tại và mỗi khi mã nguồn bên ngoài gọi các hàm trong chương trình, bạn có thể kiểm tra trạng thái hiện tại để thực hiện công việc thích hợp. Đây là chương trình thử nghiệm. Trong chương trình này bạn tạo mới một đối tượng RentalMethods , đưa vào hàm khởi tạo một số 9, để nói rằng có 9 căn hộ có thể cho thuê. Sau đó chương trình gọi hàm getApplication và checkApplication để mô phỏng có một người thuê nhà mới.

```
public class TestRentalMethods
{
  RentalMethods rentalMethods;

  public static void main(String args[])
  {
    TestRentalMethods t = new TestRentalMethods();
  }

  public TestRentalMethods()
  {
    rentalMethods = new RentalMethods(9);

    rentalMethods.getApplication();
    rentalMethods.checkApplication();

  }
}
```

Và đây là kết quả

```
Thanks for the application.
Congratulations, you were approved.
Renting you an apartment....
Here are your keys!
```

Không tệ, bạn đã tạo ra một hệ thống tự động tiếp nhận và giải quyết các yêu cầu thuê nhà cho khách hàng.

Nhưng có một rắc rối với phương pháp dựa trên các hàm. Đó là khi bạn thêm nhiều trạng thái nữa, từng hàm sẽ trở nên dài và dài hơn, và mỗi hàm phải viết lại cho tất cả trạng thái mới. Bạn sẽ làm gì? Hãy đóng gói chúng.

Sử dụng đối tượng để đóng gói trạng thái

Thay vì lưu trữ từng trạng thái trong các hằng số, có một ý tưởng tốt hơn là đưa chúng vào trong các lớp. Với cách này, bạn có thể gọi hàm dispensKeys hoặc checkApplication trên đối tượng trạng thái state bất cứ đâu trong mã nguồn. Tất cả những gì bạn phải làm là nạp đúng đối tượng trạng thái state vào một biến, và gọi các hàm khác nhau trên biến đó. Ví dụ nếu đối tượng state hiện tại tương ứng với trạng thái chờ đợi, bạn sẽ nhận được một trả lời khác khi bạn gọi hàm gotApplication hơn là nếu đối tượng hiện tại tương ứng với trạng thái đã cho thuê hết, khi mà không còn căn hộ nào nữa.

Bạn quyết định lưu giữ trạng thái hiện tại trong một đối tượng để làm mã nguồn rõ ràng hơn. Làm sao để thiết kế một hệ thống tự động sử dụng một đối tượng trạng thái state? Hệ thống này lưu trữ trạng thái hiện tại trong một đối tượng state, đối tượng này có thể lưu trữ một trong bốn đối tượng trạng thái có thể sau : WaitingState , GotApplicationState , ApartmentRentedState và FullyRentedState.

Xem hình sau:

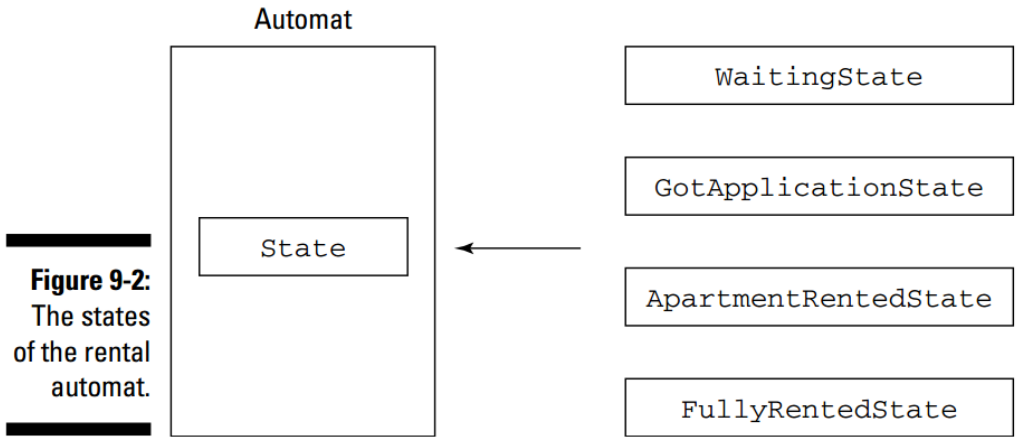


Figure 9-2: The states of the rental automat.