

Your Brain on Design Patterns

Head First Design Patterns

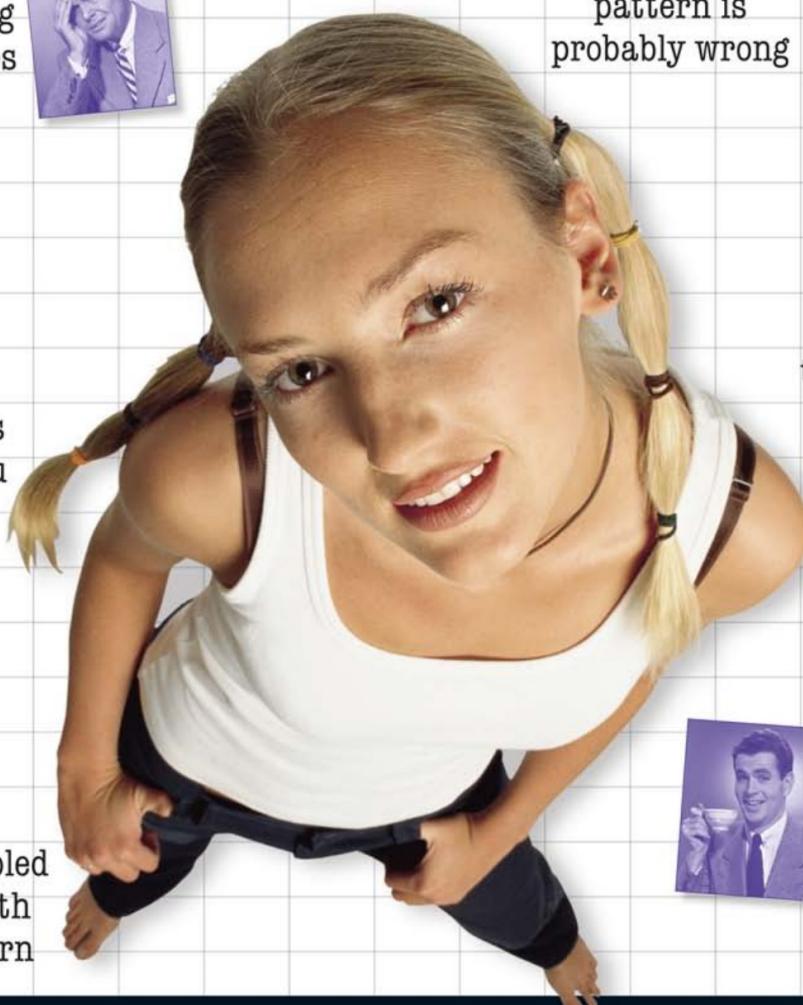
Avoid those embarrassing coupling mistakes



Discover the secrets of the Patterns Guru



Find out how Starbuzz Coffee doubled their stock price with the Decorator pattern



Learn why everything your friends know about Factory pattern is probably wrong



Load the patterns that matter straight into your brain



See why Jim's love life improved when he cut down his inheritance

Head First Design Patterns

"I received the book yesterday and started to read it... and I couldn't stop. This is très "cool." It is fun, but they cover a lot of ground and they are right to the point. I'm really impressed."

—Erich Gamma,
IBM Distinguished Engineer, and
coauthor of *Design Patterns*

"I feel like a thousand pounds of books have just been lifted off of my head."

—Ward Cunningham,
inventor of the Wiki and
founder of the Hillside Group

"This book is close to perfect, because of the way it combines expertise and readability. It speaks with authority and it reads beautifully."

—David Gelernter, Professor of Computer Science, Yale University

"One of the funniest and smartest books on software design I've ever read."

—Aaron LaBerge,
VP Technology, ESPN.com

Software Development/Java

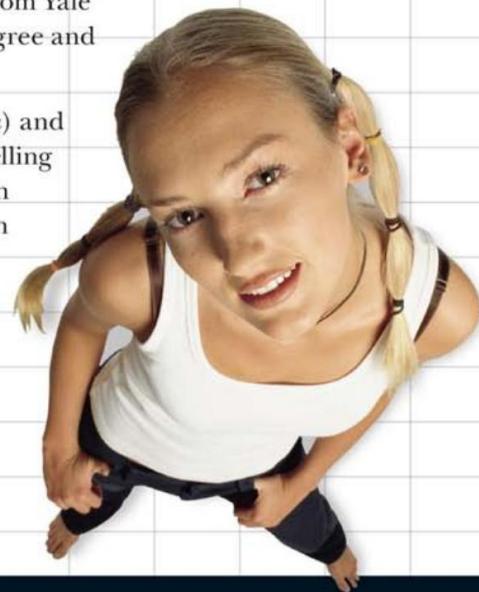
You know you don't want to reinvent the wheel (or worse, a flat tire), so you look to design patterns—the lessons learned by those who've faced the same software design problems. With design patterns, you get to take advantage of the best practices and experience of others, so that you can spend your time on...something else. Something more challenging. Something more complex. Something more *fun*. You want to learn:

- The patterns that *matter*
- *When* to use them, and *why*
- How to *apply* them to your own designs, *right now*
- When *not* to use them (how to avoid pattern fever)
- OO design principles on which patterns are based

Most importantly, you want to learn design patterns in a way that won't put you to sleep. If you've read a Head First book, you know what to expect—a visually rich format designed for the way your brain works. Using the latest research in neurobiology, cognitive science, and learning theory, *Head First Design Patterns* will load patterns into your brain in a way that sticks. In a way that makes you better at solving software design problems, and better at speaking the language of patterns with others on your team.

Eric Freeman and **Elisabeth Freeman** are authors, educators, and technology innovators. After four years leading digital media and Internet efforts at the Walt Disney Company, they're applying some of that pixie dust to their own media, including this book. Eric and Elisabeth both hold computer science degrees from Yale University: Elisabeth holds an M.S. degree and Eric a Ph.D.

Kathy Sierra (founder of *javaranch.com*) and **Bert Bates** are the creators of the best-selling Head First series and developers of Sun Microsystems Java developer certification exams.



www.oreilly.com

US \$44.95 CAN \$65.95

ISBN-10: 0-596-00712-4

ISBN-13: 978-0-596-00712-6



5 4 4 9 5

9 780596 007126

O'REILLY®

Lời khen ngợi cho Cái đầu tiên Thiết kế Các mẫu

"Tôi đã nhận được cuốn sách hôm qua và bắt đầu đọc nó trên đường về nhà... và tôi không thể dừng lại. Tôi đã mang nó đến phòng tập thể dục và tôi nghĩ mọi người đã thấy tôi mỉm cười rất nhiều trong khi tôi đang tập thể dục và đọc sách. Điều này thật 'ngầu'. Nó rất vui nhung chúng bao quát rất nhiều lĩnh vực và chúng đi thẳng vào vấn đề. Tôi thực sự ấn tượng."

– Erich Gamma, Kỹ sư xuất sắc của IBM và đồng tác giả của
Design Patterns

"'Head First Design Patterns' kết hợp giữa sự vui nhộn, tiếng cười sảng khoái, hiểu biết sâu sắc, chiều sâu kỹ thuật và lời khuyên thực tế tuyệt vời trong một bài đọc giải trí và kích thích tư duy. Cho dù bạn là người mới làm quen với các mẫu thiết kế hay đã sử dụng chúng trong nhiều năm, bạn chắc chắn sẽ học được điều gì đó khi ghé thăm Objectville."

– Richard Helm, đồng tác giả của "Design Patterns" cùng với những người khác
Gang of Four - Erich Gamma, Ralph Johnson và John Vlissides

"Tôi cảm thấy như thể hàng ngàn pound sách vừa được nhắc khỏi đầu mình vậy."

– Ward Cunningham, người phát minh ra Wiki và là người sáng lập Hillside Group

"Cuốn sách này gần như hoàn hảo, vì cách nó kết hợp giữa chuyên môn và khả năng đọc. Nó nói lên thẩm quyền và đọc rất hay. Đây là một trong số rất ít sách về phần mềm mà tôi từng đọc mà tôi thấy không thể thiếu. (Tôi sẽ xếp khoảng 10 cuốn vào thể loại này, ở ngoài cùng.)"

– David Gelernter, Giáo sư Khoa học Máy tính,
Đại học Yale và là tác giả của "Mirror Worlds" và "Machine Beauty"

"Một cú lặn sâu vào thế giới của các mẫu hình, một vùng đất nơi những thứ phức tạp trở nên đơn giản, nhưng nơi những thứ đơn giản cũng có thể trở nên phức tạp. Tôi không nghĩ ra được hướng dẫn viên du lịch nào tốt hơn Freemans."

– Miko Matsumura, Nhà phân tích ngành, The Middleware Company
Cựu Trưởng nhóm truyền bá Java, Sun Microsystems

"Tôi đã cười, đã khóc, điều đó làm tôi cảm động."

– Daniel Steinberg, Tổng biên tập, java.net

"Phản ứng đầu tiên của tôi là lật ra cười trên sàn. Sau khi đứng dậy, tôi nhận ra rằng cuốn sách không chỉ chính xác về mặt kỹ thuật mà còn là cuốn sách giới thiệu về các mẫu thiết kế dễ hiểu nhất mà tôi từng thấy."

– Tiến sĩ Timothy A. Budd, Phó Giáo sư Khoa học Máy tính tại Đại học Tiểu bang Oregon và là tác giả của hơn một chục cuốn sách, bao gồm "C++ dành cho Lập trình viên Java"

"Jerry Rice chạy theo mẫu tốt hơn bất kỳ cầu thủ bắt bóng nào trong NFL, nhưng Freemans đã chạy nhanh hơn anh ấy.

Nghiêm túc mà nói... đây là một trong những cuốn sách về thiết kế phần mềm hài hước và thông minh nhất mà tôi từng đọc."

– Aaron LaBerge, Phó chủ tịch Công nghệ, ESPN.com

Thêm lời khen ngợi cho Cái đầu Đầu tiên Thiết kế Các mẫu

"Thiết kế mã tuyệt vời, trước hết và quan trọng nhất, là thiết kế thông tin tuyệt vời. Một nhà thiết kế mã đang dạy máy tính cách làm một việc gì đó, và không có gì ngạc nhiên khi một giáo viên giỏi về máy tính lại trở thành một giáo viên giỏi về lập trình viên. Sự rõ ràng đáng ngưỡng mộ, tính hài hước và liều lượng thông minh đáng kể của cuốn sách này khiến nó trở thành loại sách giúp ngay cả những người không phải lập trình viên cũng suy nghĩ tốt về cách giải quyết vấn đề."

- Cory Doctorow, đồng biên tập của Boing Boing và là tác giả của "Down and Out in the Magic Kingdom" và "Someone Comes to Town, Someone Leaves Town"

"Có một câu nói cũ trong ngành máy tính và trò chơi điện tử - à, nó không thể cũ đến thế được vì ngành này không cũ đến thế - và nó đại loại như thế này: Thiết kế là Cuộc sống. Điều đặc biệt gây tò mò về câu nói này là ngay cả ngày nay, hầu như không ai làm nghề sáng tạo trò chơi điện tử có thể đồng ý về ý nghĩa của việc "thiết kế" một trò chơi. Nhà thiết kế có phải là kỹ sư phần mềm không? Một giám đốc nghệ thuật?

Người kể chuyện? Kiến trúc sư hay thợ xây? Người chào hàng hay người có tầm nhìn xa? Một cá nhân có thể thực sự là một phần của tất cả những điều này không? Và quan trọng nhất, ai quan tâm đến điều đó?

Người ta nói rằng ghi công "thiết kế bởi" trong giải trí tương tác cũng giống như ghi công "đạo diễn bởi" trong làm phim, trên thực tế, điều này cho phép nó chia sẻ DNA với có lẽ là ghi công gây tranh cãi nhất, cường điệu nhất và thường hoàn toàn thiếu khiêm tốn nhất từng được tuyên truyền về nghệ thuật thương mại.

Một công ty tốt, phải không? Nhưng nếu Thiết kế là Cuộc sống, thì có lẽ đã đến lúc chúng ta dành một số thời gian để suy nghĩ về nó.

Eric và Elisabeth Freeman đã dùng cảm tình nguyệt nhin vào bức màn mã cho chúng ta trong "Head First Design Patterns." Tôi không chắc là cả hai người họ có quan tâm nhiều đến PlayStation hay X-Box hay không, và họ cũng không nên quan tâm. Tuy nhiên, họ đề cập đến khái niệm thiết kế ở mức độ trung thực đáng kể, do đó bất kỳ ai đang tìm kiếm sự cung cấp cái tôi cho tác giả tuyệt vời của chính mình thì tốt nhất là không nên đào bới ở đây, nơi sự thật được tiết lộ một cách đáng kinh ngạc. Những kẻ ngụy biện và kẻ rao hàng trong rạp xiếc không cần phải nộp đơn. Thể hệ trí thức tiếp theo hãy đến và mang theo một cây bút chì."

- Ken Goldstein, Phó chủ tịch điều hành & Tổng giám đốc, Disney trực tuyến

"Đúng là giọng điệu phù hợp với những lập trình viên lão luyện, bình thường nhưng tuyệt vời trong tất cả chúng ta. Tài liệu tham khảo phù hợp cho các chiến lược phát triển thực tế - giúp não tôi hoạt động mà không cần phải vật lộn với một loạt ngôn ngữ giáo sư nhảm chán, cũ kỹ."

- Travis Kalanick, Nhà sáng lập Scour và Red Swoosh Thành viên của MIT TR100

"Cuốn sách này kết hợp sự hài hước, những ví dụ tuyệt vời và kiến thức sâu sắc về Design Patterns theo cách khiến việc học trở nên thú vị. Là người làm trong ngành công nghệ giải trí, tôi rất thích Nguyên tắc Hollywood và Home Theater Facade Pattern, để kể tên một vài ví dụ. Hiểu biết về Design Patterns không chỉ giúp chúng ta tạo ra phần mềm chất lượng có thể tái sử dụng và bảo trì được mà còn giúp chúng ta nâng cao kỹ năng giải quyết vấn đề trên mọi lĩnh vực vấn đề. Cuốn sách này là cuốn sách phải đọc đối với tất cả các chuyên gia và sinh viên máy tính."

- Newton Lee, Nhà sáng lập và Tổng biên tập, Máy tính trong Giải trí (acmcie.org) của Hiệp hội Máy tính (ACM)

Thêm lời khen ngợi cho Cái đầu Đầu tiên Thiết kế Các mẫu

"Nếu có một môn học nào đó cần được giảng dạy tốt hơn, cần phải thú vị hơn khi học thì đó chính là các mẫu thiết kế. Cảm ơn Chúa vì đã có Head First Design Patterns.

Từ những người tuyệt vời của Head First Java, cuốn sách này sử dụng mọi thủ thuật có thể để giúp bạn hiểu và nhớ.

Không chỉ là hàng loạt hình ảnh: hình ảnh con người, có xu hướng thu hút sự chú ý của những người khác.

Bất ngờ ở khắp mọi nơi. Những câu chuyện, vì con người thích kể chuyện. (Những câu chuyện về những thứ như pizza và sô cô la. Chúng ta cần nói thêm không?) Thêm vào đó, nó cực kỳ buồn cười.

Nó cũng bao gồm một loạt các khái niệm và kỹ thuật khổng lồ, bao gồm hầu hết các mẫu mà bạn sẽ sử dụng nhiều nhất (observer, decorator, factory, singleton, command, adapter, façade, template method, iterator, composite, state, proxy). Hãy đọc nó, và những thứ đó sẽ không chỉ là 'những từ ngữ': chúng sẽ là những ký ức làm bạn thích thú, và là những công cụ mà bạn sở hữu."

– Bill Camarda, CHỈ ĐỌC

"Sau khi sử dụng Head First Java để dạy sinh viên năm nhất cách bắt đầu lập trình, tôi đã háo hức chờ đợi để xem cuốn sách tiếp theo trong bộ sách. Head First Design Patterns là cuốn sách đó và tôi rất vui mừng. Tôi chắc chắn rằng nó sẽ nhanh chóng trở thành cuốn sách chuẩn mực đầu tiên về các mẫu thiết kế để đọc và là cuốn sách tôi đang giới thiệu cho sinh viên."

– Ben Bederson, Phó Giáo sư Khoa học Máy tính & Giám đốc

Phòng thí nghiệm tương tác giữa người và máy tính, Đại học Maryland

"Thông thường khi đọc một cuốn sách hoặc bài viết về các mẫu thiết kế, thỉnh thoảng tôi phải tự chọn vào mắt mình bằng một vật gì đó chỉ để chắc chắn rằng tôi đang chú ý. Nhưng không phải với cuốn sách này. Nghe có vẻ kỳ lạ, nhưng cuốn sách này khiến việc học về các mẫu thiết kế trở nên thú vị.

Trong khi những cuốn sách khác về mẫu thiết kế đang nói rằng, 'Buehler... Buehler... Buehler...' thì cuốn sách này lại vang lên rằng 'Hãy lắc mạnh lên, em yêu!'

– Eric Wuehler

"Tôi thực sự thích cuốn sách này. Thực tế là tôi đã hôn cuốn sách này trước mặt vợ tôi."

– Satish Kumar

Lời khen ngợi cho Cái đầu Đầu tiên Tiếp cận

"Công nghệ Java có ở khắp mọi nơi-trong điện thoại di động, ô tô, máy ảnh, máy in, trò chơi, PDA, ATM, thẻ thông minh, máy bơm xăng, sân vận động thể thao, thiết bị y tế, Web cam, máy chủ, v.v. Nếu bạn phát triển phần mềm và chưa học Java, thì chắc chắn đã đến lúc phải bắt đầu-Head First."

– Scott McNealy, Chủ tịch, Tổng giám đốc điều hành và Chủ tịch Sun Microsystems

"Nó nhanh, không trang trọng, vui vẻ và hấp dẫn. Hãy cần thận-bạn thực sự có thể học được điều gì đó!"

– Ken Arnold, cựu Kỹ sư cao cấp tại Sun Microsystems

Đồng tác giả (với James Gosling, người sáng tạo ra Java), "Ngôn ngữ lập trình Java"

Những cuốn sách liên quan khác từ O'Reilly

Học Java

Java tóm tắt

Java Enterprise tóm tắt

Ví dụ về Java tóm tắt

Sách dạy nấu ăn Java

Các mẫu thiết kế J2EE

Những cuốn sách khác trong bộ Head First của O'Reilly

Đầu tiên Java

Đầu tiên EJB

Head First Servlets & JSP

Phân tích và thiết kế hướng đối tượng đầu tiên

Head First HTML với CSS & XHTML

Đầu Rush Ajax

Đầu tiên PMP

SQL đầu tiên (2007)

Đầu tiên C# (2007)

Phát triển phần mềm đầu tiên (2007)

JavaScript đầu tiên (2007)

Hãy theo dõi để biết thêm nhiều cuốn sách khác trong bộ Head First nhé!

Các mẫu thiết kế Head First



Eric Freeman
Elisabeth Freeman

với
Kathy Sierra
Bert Bates

O'REILLY®

Bắc Kinh • Cambridge • Köln • Paris • Sebastopol • Đà Nẵng • Tokyo



Mã số ISBN-10: 0-596-00712-4

[Nam]

Mã số ISBN-13: 978-0-596-00712-6

[07/07]

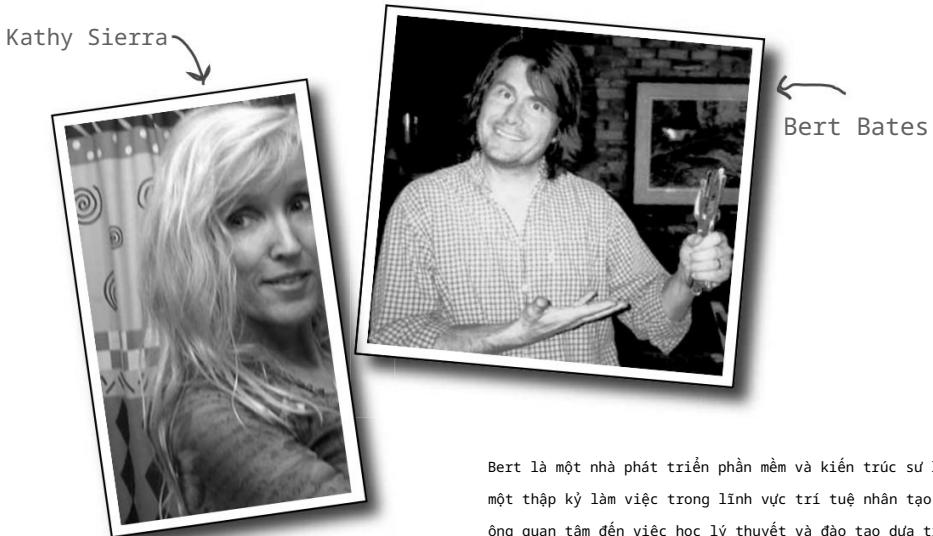
Xin gửi lời cảm ơn đến Gang of Four, những người có hiểu biết sâu sắc và chuyên môn trong việc nắm bắt và truyền đạt các Mẫu thiết kế đã thay đổi bộ mặt của thiết kế phần mềm mãi mãi và cải thiện cuộc sống của các nhà phát triển trên toàn thế giới.

Nhưng nghiêm túc mà nói, khi nào chúng ta mới được thấy phiên bản thứ hai? Dù sao thì cũng chỉ mới mười năm thôi mà!



Những người sáng tạo ra loạt truyện Head First

(và những người đồng mưu trong cuốn sách này)



Kathy đã quan tâm đến việc học lý thuyết kể từ những ngày còn là nhà thiết kế trò chơi (cô đã viết trò chơi cho Virgin, MGM và Amblin'). Cô đã phát triển phần lớn định dạng Head First trong khi giảng dạy về New Media Authoring cho chương trình Nghiên cứu giải trí của UCLA Extension.

Gần đây hơn, cô là một giảng viên chính cho Sun Microsystems, hướng dẫn các giảng viên Java của Sun cách giảng dạy các công nghệ Java mới nhất và phát triển một số kỳ thi chứng chỉ của Sun. Cùng với Bert Bates, cô đã tích cực sử dụng các khái niệm Head First để giảng dạy cho hàng ngàn nhà phát triển. Kathy là người sáng lập javaranch.com, trang web đã giành được Giải thưởng Năng suất Jolt Cola của tạp chí Phát triển phần mềm năm 2003 và 2004.

Bạn có thể thấy cô ấy dạy Java trên Java Jam Geek Cruise (geekcruises.com).

Cô ấy vừa mới chuyển từ California đến Colorado, nơi cô phải học những từ mới như "ice scraper" và "fle eece", nhưng tia sét ở đó thi thật tuyệt vời.

Thích: chạy, trượt tuyết, trượt ván, chơi với chó ngựa Iceland và khoa học kỹ lưỡng. Ghét: entropy.

Bạn có thể tìm thấy cô ấy trên javaranch hoặc thỉnh thoảng viết blog trên java.net. Viết thư cho cô ấy theo địa chỉ kathy@wickedlysmart.com.

Bert là một nhà phát triển phần mềm và kiến trúc sư lâu năm, nhưng một thập kỷ làm việc trong lĩnh vực trí tuệ nhân tạo đã thúc đẩy ông quan tâm đến việc học lý thuyết và đào tạo dựa trên công nghệ. Ông đã giúp khách hàng trở thành những lập trình viên giỏi hơn kể từ đó. Gần đây, ông đã đứng đầu nhóm phát triển cho một số kỳ thi Chứng chỉ Java của Sun.

Ông đã dành thập kỷ đầu tiên trong sự nghiệp phần mềm của mình để đi khắp thế giới để giúp các khách hàng phát sóng như Radio New Zealand, Weather Channel và Arts & Entertainment Network (A & E). Một trong những dự án yêu thích nhất mọi thời đại của ông là xây dựng mô phỏng hệ thống đường sắt dài dọc cho Union Pacific Railroad.

Bert là một người chơi cờ vây lâu năm và nghiên cứu trò này, và đã dành quá nhiều thời gian để lập trình một chương trình cờ vây. Anh ấy là một người chơi guitar khá giỏi và hiện đang thử sức với đàn banjo.

Bạn có thể tìm anh ấy trên javaranch, trên máy chủ IGS go hoặc viết thư cho anh ấy theo địa chỉ terrapin@wickedlysmart.com.

mục lục

Mục lục (tóm tắt)

Giới thiệu

xv

1	Chào mừng đến với Design Patterns: phần giới thiệu	1
2	Giữ cho Đối tượng của bạn được biệt: Mẫu Quan sát	37
3	Đồ vật trang trí: Mẫu trang trí	79
4	Nướng với sự tuyệt vời của OO: Mẫu Factory	109
5	Đối tượng độc đáo: Mẫu Singleton	169
6	Đóng gói lệnh gọi: Mẫu lệnh	191
7	Trở nên thích nghi: Các mẫu Adaptor và Facade	235
8	Thuật toán đóng gói: Mẫu phương pháp Template	275
9	Bộ sưu tập được quản lý tốt: Iterator và các mẫu tổng hợp	315
10	Trạng thái của sự vật: Mô hình trạng thái	385
11	Kiểm soát truy cập đối tượng: Mẫu Proxy	429
12	Mẫu của Mẫu: Mẫu Hợp chất	499
13	Mẫu trong Thế giới Thực: Sống Tốt hơn với Các Mẫu	577
14	Phụ lục: Các mẫu còn sót lại	611

Mục lục (thực tế)

Giới thiệu

Bộ não của bạn về các Mẫu thiết kế. Ở đây bạn đang cố gắng học một cái gì đó, trong khi ở đây não của bạn đang giúp bạn một việc bằng cách đảm bảo rằng việc học không bị đính chặt. Não của bạn nghĩ rằng, "Tốt hơn là nên để chỗ cho những thứ quan trọng hơn, như là tránh những loài động vật hoang dã nào và liều trượt túýt khóa thân có phải là một ý tưởng tồi không." Vậy làm thế nào để bạn đánh lừa bộ não của mình suy nghĩ rằng cuộc sống của bạn phụ thuộc vào việc biết các Mẫu thiết kế?

Cuốn sách này dành cho ai?	xxvi
Chúng tôi biết não bạn đang nghĩ gì	xxvii
Siêu nhận thức	xxix
Hãy uốn cong bộ não của bạn để khuất phục	xxxI
Người đánh giá kỹ thuật	xxxiv
Lời cảm ơn	xxxv

giới thiệu về Mẫu thiết kế

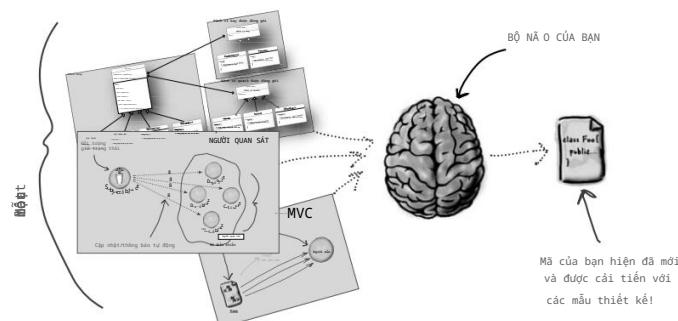
Chào mừng đến với Design Patterns

1

Ai đó đã giải quyết vấn đề của bạn. Trong chương này, bạn sẽ học được lý do tại sao (và làm thế nào) bạn có thể khai thác trí tuệ và bài học kinh nghiệm của những nhà phát triển khác đã từng trải qua cùng một vấn đề thiết kế và đã sống sót chuyền đi. Trước khi kết thúc, chúng ta sẽ xem xét việc sử dụng và lợi ích của các mẫu thiết kế, hãy xem một số nguyên tắc thiết kế chính và hướng dẫn qua một ví dụ về cách một mẫu hoạt động. Cách tốt nhất để sử dụng các mẫu là nạp chúng vào não bạn và sau đó nhận ra những vị trí trong thiết kế và ứng dụng hiện tại của bạn mà bạn có thể áp dụng chúng. Thay vì sử dụng lại mã, với các mẫu, bạn sẽ có được kinh nghiệm sử dụng lại.



Ứng dụng SimUDuck	2
Joe nghĩ về việc thừa kế...	5
Thể côn giao diện thì sao?	6
Một hằng số trong phát triển phần mềm	8
Phân biệt những gì thay đổi với những gì vẫn giữ nguyên	10
Thiết kế hành vi của vịt	11
Kiểm tra mã Duck	18
Thiết lập hành vi một cách năng động	20
Bức tranh toàn cảnh về hành vi được đóng gói	22
HAS-A có thể tốt hơn IS-A	23
Mẫu chiến lược	24
Sức mạnh của một vốn từ vựng mẫu chung	28
Tôi sử dụng Mẫu thiết kế như thế nào?	29
Công cụ cho hộp công cụ thiết kế của bạn	32
Giải pháp bài tập	34



Mẫu quan sát viên

Giữ cho các đối tượng của bạn được biết

2

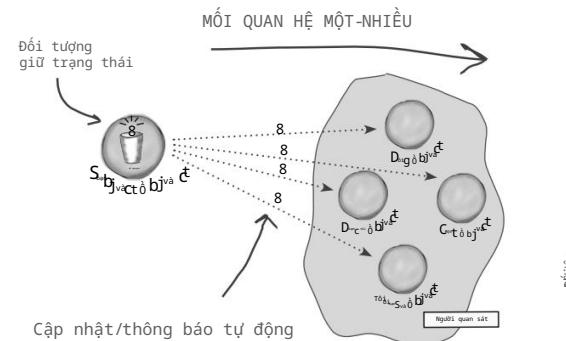
Đừng bỏ lỡ khi có điều gì thú vị xảy ra!

Chúng tôi có một mẫu giúp các đối tượng của bạn biết khi có thứ gì đó có thể quan tâm đến những gì xảy ra. Các đối tượng thậm chí có thể quyết định tại thời gian chạy xem chúng muốn được thông báo. Mẫu quan sát là một trong những mẫu được sử dụng nhiều nhất mẫu trong JDK và nó cực kỳ hữu ích. Trước khi hoàn thành, chúng ta cũng sẽ xem xét ở một đến nhiều mối quan hệ và sự kết hợp lồng lèo (vâng, đúng vậy, chúng tôi đã nói (ghép nối)). Với Observer, bạn sẽ là linh hồn của Patterns Party.

Cơ bản về 00	Ứng dụng theo dõi thời tiết	39
Mô hình	Gặp gỡ mẫu quan sát viên	44
Đóng gói	Nhà xuất bản + Người đăng ký = Mẫu quan sát	45
Đa hình	Kịch nambi phút: chủ đề để quan sát	48
Di truyền	Mẫu quan sát được định nghĩa	51
Bao gồm những gì thay đổi	Sức mạnh của sự kết nối lồng lèo	53
Ưu tiên thành phần hơn là kế thừa	Thiết kế Trạm thời tiết	56
sự thay đổi	Triển khai Trạm thời tiết	57
Chương trình đến Giao diện,	Sử dụng Observer Pattern tích hợp của Java	64
không phải triển khai	Mặt tối của java.util.Observable	71
Có gắng thiết kế các đối	Công cụ cho hộp công cụ thiết kế của bạn	74
tượng có sự kết hợp lồng lèo	Giải pháp bài tập	78
tương tác		

Nguyên tắc 00

Đa hình
Di truyền
Bao gồm những gì thay đổi
Ưu tiên thành phần hơn là kế thừa
sự thay đổi
Chương trình đến Giao diện,
không phải triển khai
Có gắng thiết kế các đối
tượng có sự kết hợp lồng lèo
tương tác



Mẫu trang trí

Đồ vật trang trí

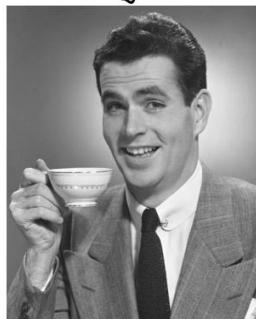
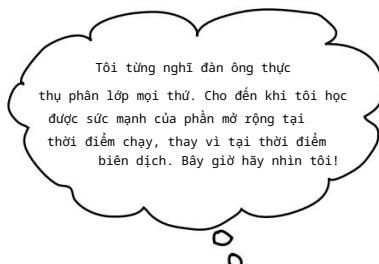
3

Chỉ cần gọi chương này là "Con mắt thiết kế dành cho người thừa kế".

Chúng ta sẽ xem xét lại việc lạm dụng thừa kế thông thường và bạn sẽ học cách để trang trí các lớp của bạn khi chạy bằng cách sử dụng một dạng thành phần đối tượng. Tại sao?

Một khi bạn biết các kỹ thuật trang trí, bạn sẽ có thể cung cấp cho (hoặc

của người khác) đối tượng trách nhiệm mới mà không cần thực hiện bất kỳ thay đổi mã nào đến các lớp cơ bản.



Chào mừng đến với Starbuzz Coffee	80
Nguyên lý Mở-Đóng	86
Gặp gỡ mẫu trang trí	88
Xây dựng đơn đặt hàng đồ uống với người trang trí	89
Mẫu Decorator được định nghĩa	91
Trang trí đồ uống của chúng tôi	92
Viết mã Starbuzz	95
Trình trang trí thế giới thực: Java I/O	100
Viết Java I/O Decorator của riêng bạn	102
Công cụ cho hộp công cụ thiết kế của bạn	105
Giải pháp bài tập	106

Mẫu Nhà Máy

Nướng với OO Goodness

4

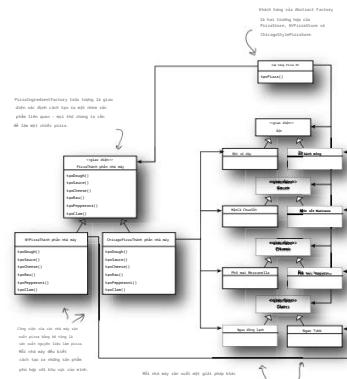
Hãy chuẩn bị để thực hiện một số thiết kế OO lỏng lẻo.

Có nhiều cách để tạo ra các đối tượng hơn là chỉ sử dụng toán tử mới . Bạn sẽ học

rằng việc khởi tạo là một hoạt động không phải lúc nào cũng nên được thực hiện ở nơi công cộng và có thể

thường dẫn đến các vấn đề về khớp nối. Và bạn không muốn điều đó, phải không? Tìm hiểu cách

Factory Patterns có thể giúp bạn tránh khỏi những sự phụ thuộc đáng xấu hổ.



Khi bạn nhìn thấy "mới", hãy nghĩ đến "bê tông"	110
Pizza Objectville	112
Đóng gói việc tạo đối tượng	114
Xây dựng một nhà máy sản xuất pizza đơn giản	115
Nhà máy đơn giản được định nghĩa	117
Một khuôn khổ cho cửa hàng pizza	120
Cho phép các lớp con quyết định	121
Hãy cùng làm một PizzaStore	123
Khai báo phương thức nhà máy	125
Gặp gỡ mô hình phương pháp nhà máy	131
Phân cấp lớp song song	132
Mẫu phương pháp nhà máy được xác định	134
Một PizzaStore rất phụ thuộc	137
Xem xét các phụ thuộc đối tượng	138
Nguyên lý đảo ngược sự phụ thuộc	139
Trong khi đó, quay lại PizzaStore...	144
Các nhóm thành phần...	145
Xây dựng nhà máy sản xuất nguyên liệu của chúng tôi	146
Nhìn vào Nhà máy trữ lượng	153
Hậu trường	154
Mẫu Abstract Factory được định nghĩa	156
So sánh Factory Method và Abstract Factory	160
Công cụ cho hộp công cụ thiết kế của bạn	162
Giải pháp bài tập	164

Mẫu Singleton

5 Một trong những đối tượng loại
Mẫu Singleton: tấm vé giúp bạn tạo ra những vật thể độc nhất vô nhị, chỉ có một trường hợp duy nhất. Bạn có thể vui mừng khi biết rằng trong tất cả các mẫu, Singleton là mẫu đơn giản nhất về mặt của sơ đồ lớp của nó; trên thực tế sơ đồ chỉ chứa một lớp duy nhất! Nhưng đừng hiểu quá thoải mái; mặc dù nó đơn giản từ góc độ thiết kế lớp, chúng tôi sẽ gặp khá nhiều trở ngại và khó khăn trong quá trình thực hiện. Vì vậy, hãy thắt chặt lênh-việc này không đơn giản như bạn nghĩ đâu...



Một và chỉ một đối tượng	170
Cô gái độc thân nhỏ bé	171
Phân tích mô hình Singleton cổ điển	173
Lời thú tội của một người độc thân	174
Nhà máy Sôcôla	175
Mẫu Singleton được định nghĩa	177
Hershey, PA	
Houston, chúng ta có vấn đề rồi...	178
TRỞ THÀNH JVM	179
Xử lý đa luồng	180
Hỏi & Đáp về Singleton	184
Công cụ cho hộp công cụ thiết kế của bạn	186
Giải pháp bài tập	188

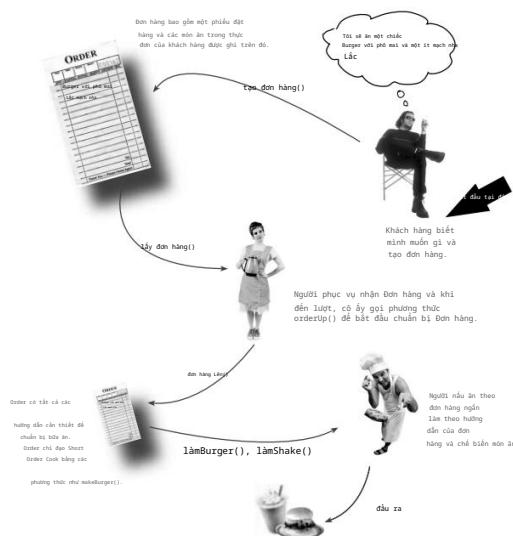


Mẫu lệnh

Đóng gói lời gọi

Trong chương này, chúng ta sẽ đưa việc đóng gói lên một cấp độ hoàn toàn mới: chúng ta sẽ đóng gói phương pháp gọi

Đúng vậy, bằng cách đóng gói lệnh gọi, chúng ta có thể kết tinh các phần tính toán để đối tượng gọi phép tính không cần phải lo lắng về cách thực hiện mọi thứ; nó chỉ sử dụng phương pháp kết tinh của chúng tôi để thực hiện nó. Chúng tôi cũng có thể làm một số những thử thông minh một cách độc ác với những lời gọi phương thức được đóng gói này, như lưu xóa chúng để ghi nhớ ký hiệu tái sử dụng chúng để thực hiện hoàn tác trong mã của chúng ta.



Tự động hóa nhà ở hoặc phá sản	192
Điều khiển từ xa	193
Xem xét các lớp nhà cung cấp	194
Trong khi đó, quay lại quán Diner...	197
Chúng ta hãy cùng nghiên cứu tương tác Diner	198
Vai trò và trách nhiệm của Objectville Diner	199
Từ Diner đến Command Pattern	201
Đổi tương lệnh đầu tiên của chúng tôi	203
Mẫu lệnh được xác định	206
Mẫu lệnh và điều khiển từ xa	208
Thực hiện điều khiển từ xa	210
Đưa điều khiển từ xa vào thử nghiệm	212
Đã đến lúc viết tài liệu đó	215
Sử dụng trạng thái để thực hiện Hoàn tác	220
Mọi điều khiển từ xa đều cần có Chế độ tiệc tùng!	224
Sử dụng lệnh Macro	225
Thêm các cách sử dụng của Command Pattern: Xếp hàng yêu cầu	228
Thêm các cách sử dụng của Command Pattern: Ghi nhật ký yêu cầu	229
Công cụ cho hộp công cụ thiết kế của bạn	230
Giải pháp bài tập	232

Các mẫu Adapter và Facade

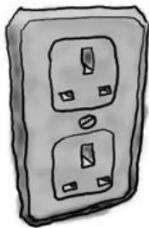
Có khả năng thích nghi

7

Trong chương này chúng ta sẽ thử những việc không thể như nhét một cái chốt vuông vào một lỗ tròn. Nghe có vẻ không thể?

Không phải khi chúng ta có Design Patterns. Bạn còn nhớ Decorator Pattern không? Chúng ta đã bao bọc các đối tượng để cung cấp cho chúng những trách nhiệm mới. Vậy giờ chúng ta sẽ bao bọc một số đối tượng với mục đích khác: để làm cho giao diện của chúng trông giống như thứ mà chúng không phải. Tại sao chúng ta lại làm như vậy? Vì vậy, chúng ta có thể điều chỉnh một thiết kế mong đợi một giao diện cho một lớp triển khai một giao diện khác. Không chỉ vậy, trong khi chúng ta đang thực hiện, chúng ta sẽ xem xét một pattern khác bao bọc các đối tượng để đơn giản hóa giao diện của chúng.

Ở cắm tường Châu Âu



Phích cắm AC tiêu chuẩn



Bộ chuyển đổi xung quanh chúng ta

236

Bộ điều hợp hướng đối tượng

237

Giải thích về mẫu Adapter

241

Mẫu bộ điều hợp được xác định

243

Bộ điều hợp đối tượng và lớp

244

Bài nói chuyện tối nay: Bộ điều hợp đối tượng và Bộ điều hợp lớp

247

Bộ chuyển đổi thế giới thực

248

Điều chỉnh một Enumeration thành một Iterator

249

Bài nói chuyện tối nay: Mẫu trang trí và mẫu bộ điều hợp

252

Rạp hát tại nhà ngọt ngào

255

Đèn, Camera, Mật tiền!

258

Xây dựng mặt tiền rạp hát tại nhà của bạn

261

Mẫu mặt tiền được xác định

264

Nguyên tắc ít kiến thức nhất

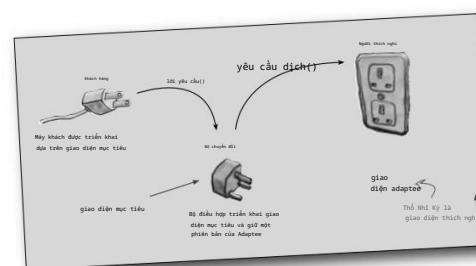
265

Công cụ cho hộp công cụ thiết kế của bạn

270

Giải pháp bài tập

272



Mẫu Phương pháp Mẫu

8

Thuật toán đóng gói

Chúng tôi đã đóng gói việc tạo đối tượng, gọi phương thức, giao diện phức tạp, vịt, pizza... điều gì có thể xảy ra tiếp theo? Chúng ta sẽ bắt đầu đóng gói các phần của thuật toán để các lớp con có thể tự mọc mình vào một phép tính bất cứ lúc nào họ muốn. Chúng ta thậm chí sẽ tìm hiểu về nguyên lý thiết kế lấy cảm hứng từ Hollywood.



Chuẩn bị một số lớp học pha cà phê và trà	277
Tóm tắt về cà phê và trà	280
Đưa thiết kế đi xa hơn	281
Tóm tắt prepareRecipe()	282
Chúng ta đã làm gì?	285
Gặp gỡ Phương pháp Mẫu	286
Chúng ta hãy pha một ít trà nhé	287
Phương pháp Mẫu mang lại cho chúng ta điều gì?	288
Mẫu Phương pháp Mẫu được xác định	289
Mã gần	290
Mê mẩn phương pháp mẫu...	292
Sử dụng mọc	293
Cà phê? Trà? Thôi, chạy TestDrive thôi	294
Nguyên tắc Hollywood	296
Nguyên tắc Hollywood và Phương pháp Mẫu	297
Phương pháp mẫu trong tự nhiên	299
Sắp xếp theo phương pháp mẫu	300
Chúng tôi có một số con vịt để phân loại	301
So sánh vịt và vịt	302
Quá trình chế tạo máy phân loại vịt	304
Đu đưa với Khung	306
Các ứng dụng nhỏ	307
Bài nói chuyện tối nay: Phương pháp và Chiến lược Mẫu	308
Công cụ cho hộp công cụ thiết kế của bạn	311
Giải pháp bài tập	312

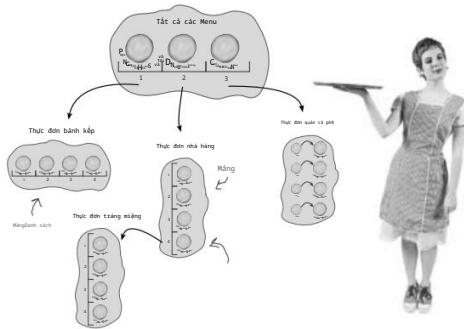
Iterator và các mẫu tổng hợp

Bộ sưu tập được quản lý tốt

9

Có rất nhiều cách để nhồi nhét các đối tượng vào một bộ sưu tập.

Đặt chúng vào một Mảng, một Ngăn xếp, một Danh sách, một Bản đồ, tùy bạn chọn. Mỗi cái có một lợi thế và sự đánh đổi. Nhưng khi khách hàng của bạn muốn lặp lại các đối tượng của bạn, bạn có định cho anh ấy xem cách thực hiện của bạn không? Chúng tôi chắc chắn hy vọng là không! Điều đó chỉ sẽ không chuyên nghiệp. Đừng lo lắng-trong chương này bạn sẽ thấy cách bạn có thể để khách hàng của bạn lặp lại các đối tượng của bạn mà không bao giờ thấy cách bạn lưu trữ đối tượng. Bạn cũng sẽ học cách tạo một số bộ sưu tập siêu đối tượng có thể vượt qua một số cấu trúc dữ liệu ánh tượng trong một lần giới hạn. Bạn cũng sẽ học một hoặc hai điều về trách nhiệm của đối tượng.



Objectville Diner và Pancake House hợp nhất

316

So sánh các triển khai Menu

318

Chúng ta có thể đóng gói quá trình lặp lại không?

323

Gặp gỡ mẫu Iterator

325

Thêm Iterator vào DinerMenu

326

Nhìn vào thiết kế

331

Dọn dẹp mọi thứ bằng java.util.Iterator

333

Điều này mang lại cho chúng ta điều gì?

335

Mẫu Iterator được định nghĩa

336

Trách nhiệm duy nhất

339

Trình lặp và Bộ sưu tập

348

Iterator và Collections trong Java 5

349

Đúng lúc chúng ta nghĩ rằng đã an toàn...

353

Mẫu tổng hợp được định nghĩa

356

Thiết kế Menu với Composite

359

Triển khai Menu tổng hợp

362

Quay lại Iterator

368

Trình lặp Null

372

Sự kỳ diệu của Iterator và Composite khi kết hợp với nhau...

374

Công cụ cho hộp công cụ thiết kế của bạn

380

Giải pháp bài tập

381

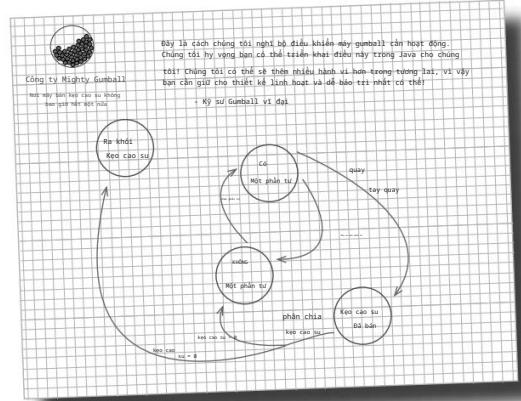
Mẫu Nhà Nước

Tình hình của sự vật

10

Một sự thật ít người biết: Mẫu Chiến lược và Mẫu Nhà nước là cặp song sinh tách ra khi mới sinh. Như bạn đã biết, Mẫu Chiến lược tiếp tục để tạo ra một doanh nghiệp cực kỳ thành công xung quanh các thuật toán có thể hoán đổi cho nhau.

Tuy nhiên, đã thực hiện con đường có lẽ cao quý hơn là giúp các đối tượng học cách kiểm soát chúng hành vi bằng cách thay đổi trạng thái bên trong của chúng. Người ta thường nghe thấy anh ta nói với đối tượng của mình khách hàng, "cứ lặp lại theo tôi, tôi đủ giỏi, tôi đủ thông minh, và tôi khôn nạn..."



Chúng ta triển khai trạng thái như thế nào?	387
Máy nhà nước 101	388
Một nỗ lực đầu tiên trong một máy trạng thái	390
Bạn biết điều đó sắp xảy ra... một yêu cầu thay đổi!	394
Tình trạng hỗn loạn của mọi thứ...	396
Xác định các giao diện và lớp trạng thái	399
Thực hiện các lớp học của tiểu bang chúng ta	401
Làm lại máy Gumball	402
Mẫu trạng thái được xác định	410
Nhà nước so với Chiến lược	411
Kiểm tra trạng thái tính táo	417
Chúng tôi gần như quên mất!	420
Công cụ cho hộp công cụ thiết kế của bạn	423
Giải pháp bài tập	424



Mẫu Proxy

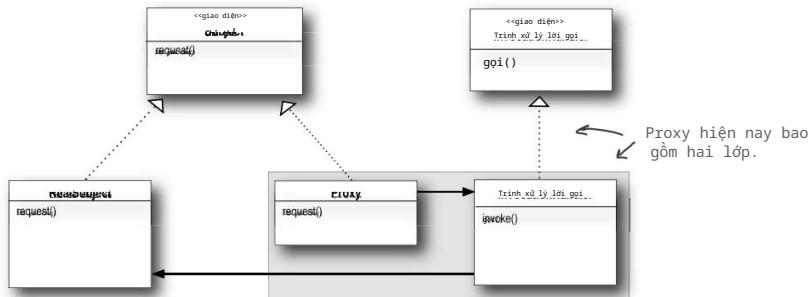
Kiểm soát truy cập đối tượng

11

Bạn đã bao giờ đóng vai cảnh sát tốt, cảnh sát xấu chưa? Bạn là cảnh sát tốt và bạn cung cấp tất cả các dịch vụ của bạn một cách tốt đẹp và thân thiện, nhưng bạn không muốn tất cả mọi người yêu cầu bạn cung cấp dịch vụ, vì vậy bạn có thể để cảnh sát xấu kiểm soát quyền truy cập vào bạn. Đó là proxy làm gì: kiểm soát và quản lý quyền truy cập. Như bạn sẽ thấy có nhiều cách mà proxy thay thế cho các đối tượng mà chúng proxy. Proxy có được biết đến là có thể vận chuyển toàn bộ các cuộc gọi phương thức qua Internet cho các đối tượng được ủy quyền của chúng; họ cũng được biết đến là kiên nhẫn đứng ở đó trong một thời gian khá lâu biếng dò vật.



Theo dõi máy bán kẹo cao su	430
Vai trò của 'proxy từ xa'	434
Đường vòng RMI	437
Proxy từ xa GumballMachine	450
Proxy từ xa ẩn sau hậu trường	458
Mẫu Proxy được định nghĩa	460
Chuẩn bị cho proxy ảo	462
Thiết kế proxy ảo cho bìa CD	464
Proxy ảo đằng sau hậu trường	470
Sử dụng proxy của Java API	474
Kịch nambi phút: bảo vệ chủ thẻ	478
Tạo proxy động	479
Vườn thú Proxy	488
Công cụ cho hộp công cụ thiết kế của bạn	491
Giải pháp bài tập	492

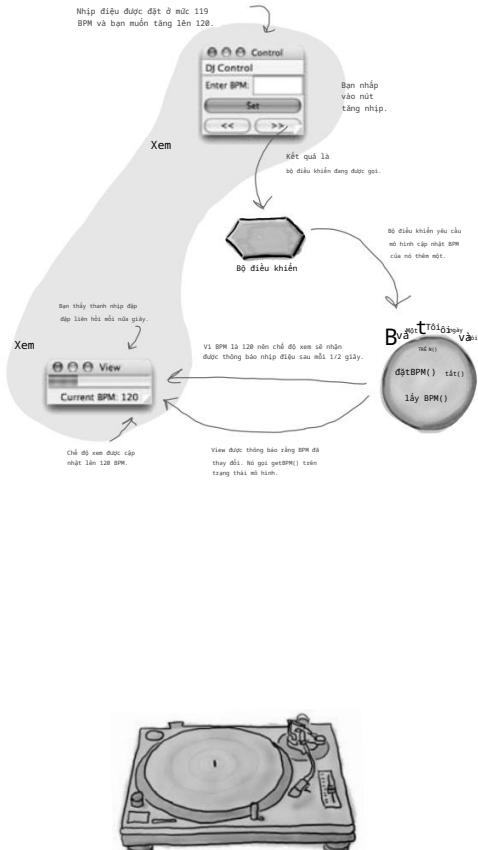


Mẫu hợp chất

Các mẫu của các mẫu

12

Ai có thể ngờ rằng Patterns có thể làm việc cùng nhau? Bạn đã chứng kiến những cuộc trò chuyện bên lò sưởi cay đắng (và biết ơn vì bạn không phải nhìn thấy các trang Pattern Death Match mà nhà xuất bản buộc chúng tôi phải xóa khỏi cuốn sách để chúng tôi có thể tránh phải sử dụng Parent's Nhẫn cảnh báo tư vấn), vì vậy ai có thể nghĩ rằng các mẫu thực sự có thể hòa hợp tốt với nhau? Tin hay không thì tùy, một số thiết kế OOP mạnh mẽ nhất sử dụng một số các mẫu với nhau. Hãy sẵn sàng để đưa kỹ năng mẫu của bạn lên một tầm cao mới; đã đến lúc Mẫu hợp chất. Chỉ cần cần thận-dòng nghiệp của bạn có thể giết bạn nếu bạn bị tấn công với Pattern Fever.



Mẫu hợp chất	500
Đoàn tụ vịt	501
Thêm bộ điều hợp	504
Thêm một trình trang trí	506
Thêm một nhà máy	508
Thêm một hợp chất và một trình lập	513
Thêm người quan sát	516
Tóm tắt các mẫu	523
Góc nhìn của một con vịt: sơ đồ lớp	524
Model-View-Controller, bài hát	526
Các mẫu thiết kế là chìa khóa của bạn để đến với MVC	528
Nhìn MVC qua lăng kính màu mẫu	532
Sử dụng MVC để điều khiển nhịp điệu...	534
Mô hình	537
Quan điểm	539
Bộ điều khiển	542
Khám phá chiến lược	545
Điều chỉnh mô hình	546
Bây giờ chúng ta đã sẵn sàng cho HeartController	547
MVC và Web	549
Mẫu thiết kế và Mô hình 2	557
Công cụ cho hộp công cụ thiết kế của bạn	560
Giải pháp bài tập	561

Cuộc sống tốt hơn với các mẫu

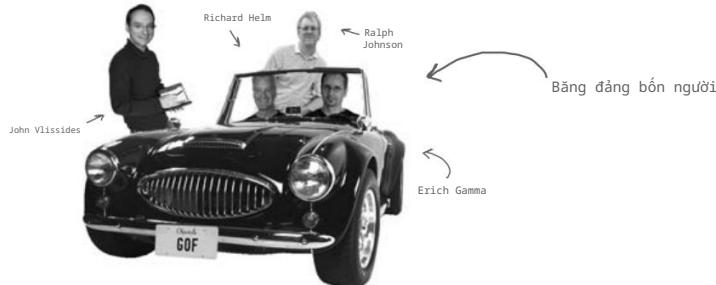
Các mẫu trong thế giới thực

13

Ahhh, giờ bạn đã sẵn sàng cho một thế giới mới tươi sáng tràn ngập các Mẫu thiết kế. Nhưng trước khi bạn mở tất cả những cánh cửa cơ hội mới đó chúng ta cần đè cập đến một số chi tiết mà bạn sẽ gặp phải ngoài thế giới thực-mọi thứ trở nên phức tạp hơn một chút ở ngoài kia so với ở Objectville. Hãy đến đây, chúng tôi có một hướng dẫn hữu ích giúp bạn vượt qua giai đoạn chuyển đổi...



Hướng dẫn Objectville của bạn	578
Mẫu thiết kế được xác định	579
Xem xét kỹ hơn định nghĩa Mẫu thiết kế	581
Mong rằng sức mạnh sẽ ở bên bạn	582
Danh mục mẫu	583
Làm thế nào để tạo ra các mẫu	586
Vậy bạn có muốn trở thành người viết Mẫu thiết kế không?	587
Tổ chức các mẫu thiết kế	589
Suy nghĩ theo khuôn mẫu	594
Tâm trí của bạn về các mẫu	597
Đừng quên sức mạnh của vốn từ vựng chung	599
Năm cách hàng đầu để chia sẻ vốn từ vựng của bạn	600
Du ngoạn Objectville cùng Gang of Four	601
Hành trình của bạn vừa mới bắt đầu...	602
Các nguồn tài nguyên Mẫu thiết kế khác	603
Vườn thú Patterns	604
Tiêu diệt cái ác bằng Anti-Patterns	606
Công cụ cho hộp công cụ thiết kế của bạn	608
Rời khỏi Objectville...	609



14 Phụ lục: Các mẫu còn sót lại

Không phải ai cũng có thể là người nổi tiếng nhất. Rất nhiều thứ đã thay đổi

10 năm qua. Kể từ khi Design Patterns: Các yếu tố của Reusable Object-Oriented

Khi phần mềm đầu tiên ra đời, các nhà phát triển đã áp dụng những mô hình này hàng nghìn lần.

Các mẫu mà chúng tôi tóm tắt trong phần phụ lục này là các mẫu có viền đầy đủ, mang thẻ, chính thức

Các mẫu GoF, nhưng không phải lúc nào cũng được sử dụng thường xuyên như các mẫu chúng ta đã khám phá

xã. Nhưng những mẫu này thật tuyệt vời theo cách riêng của chúng, và nếu tình huống của bạn đòi hỏi

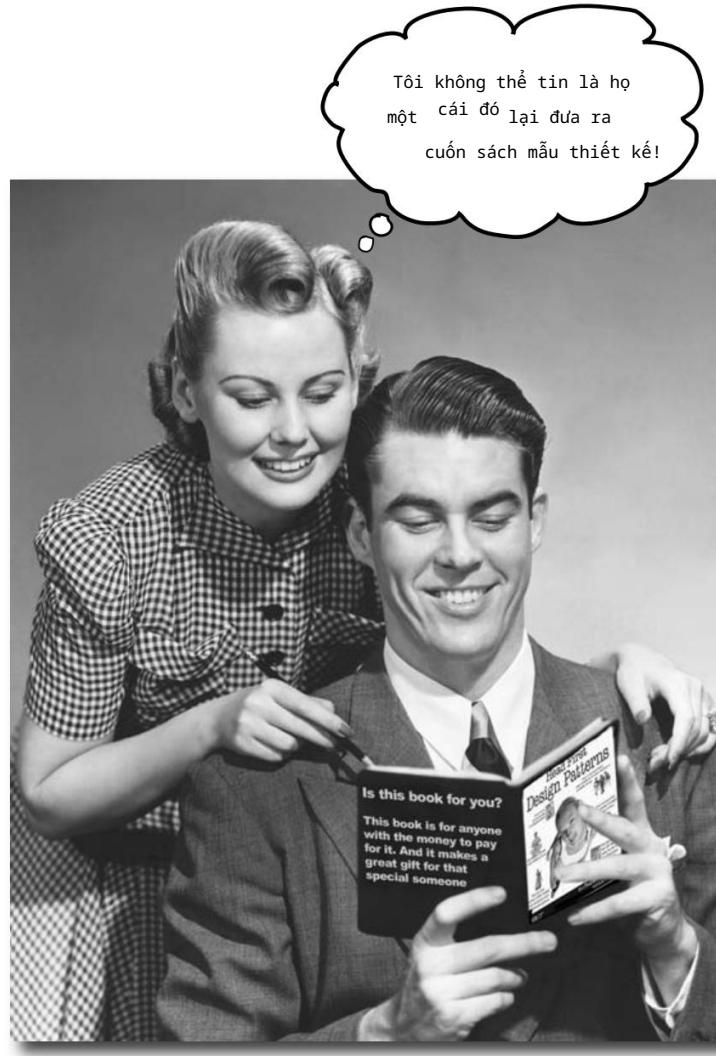
chúng, bạn nên áp dụng chúng với cái đầu ngẩng cao. Mục tiêu của chúng tôi trong phần phụ lục này là

để cung cấp cho bạn ý tưởng tổng quan về những mô hình này.

<p>Khách hàng yêu cầu Khách truy cập lấy thông tin từ Câu trúc tổng hợp... Có thể thực hiện các phương pháp mới vào Visitor mà không ảnh hưởng đến Composite.</p> <p>Người truy cập cần có khả năng gọi getState() trên các lớp và đây là nơi bạn có thể thêm các phương thức mới cho khách hàng sử dụng.</p> <p>Tất cả những lớp tổng hợp này phải làm là thêm phương thức getState() (và không phải lo lắng về việc tự tiết lộ :).</p>	Câu Câu Người xây dựng Chuỗi trách nhiệm Cân ruồi Người phiên dịch Người hòa giải Ký vật Nguyên mẫu Khách thăm tôi Mục lục 631	612 614 616 618 620 622 624 626 628
---	---	---

cách sử dụng cuốn sách này

Giới thiệu



Trong phần này, chúng tôi sẽ trả lời câu hỏi cấp thiết:
"Vậy tại sao họ LẠI đưa điều đó vào một cuốn sách về mẫu thiết kế?"

cách sử dụng cuốn sách này

Cuốn sách này dành cho ai?

Nếu bạn có thể trả lời "có" cho tất cả những câu hỏi này:

1 Bạn có biết Java không? (Bạn không cần phải là một chuyên gia.)

Có lẽ sẽ ổn hơn
nếu bạn biết C#.

2 Bạn có muốn học, hiểu, ghi nhớ và thiết kế các mẫu, bao gồm
áp dụng các nguyên tắc thiết kế hướng đối tượng mà
các mẫu thiết kế dựa trên không?

3 Bạn thích những cuộc trò chuyện sôi nổi trong bữa tiệc tối hơn là những bài
giảng khô khan, buồn tẻ và mang tính học thuật?

Cuốn sách này dành cho bạn.

Có lẽ ai nên tránh xa cuốn sách này?

Nếu bạn có thể trả lời "có" cho bất kỳ câu hỏi nào sau đây:

1 Bạn hoàn toàn mới với Java phải không?

(Bạn không cần phải quá giỏi, và ngay cả khi bạn
không biết Java, nhưng bạn biết C#, bạn có thể hiểu
ít nhất 80% các ví dụ mã. Bạn cũng có thể ổn với chỉ
kiến thức nền về C++)

2 Bạn có phải là một nhà thiết kế/lập trình viên OO tài năng không?
cho một thẩm quyền giải quyết

3 Bạn có phải là kiến trúc sư đang tìm kiếm mẫu thiết kế doanh nghiệp không?

4 Bạn có sợ thử điều gì đó khác biệt không?

Bạn có muốn điều trị tủy răng hơn là kết hợp soc với caro không?
Bạn có tin rằng một cuốn sách kỹ thuật không thể nghiêm túc nếu các thành
phần Java được nhân cách hóa không?



Cuốn sách này không dành cho bạn.

[lưu ý từ bộ phận tiếp thị: cuốn sách
này dành cho bất kỳ ai có thẻ tín dụng.]

Chúng tôi biết bạn đang nghĩ gì.

"Làm sao đây có thể là một cuốn sách lập trình nghiêm túc được?"

"Sao lại có nhiều đồ họa thế này?"

"Tôi thực sự có thể học theo cách này không?"

Và chúng tôi biết những gì của bạn não đang suy nghĩ.

Bộ não của bạn thèm khát sự mới lạ. Nó luôn tìm kiếm, quét, chờ đợi điều gì đó bất thường. Nó được xây dựng theo cách đó và giúp bạn sống sót.

Ngày nay, bạn ít có khả năng trở thành bữa ăn nhẹ của hổ. Nhưng não bạn vẫn đang tìm kiếm. Bạn không bao giờ biết được.

Vậy não bạn làm gì với tất cả những thứ thường ngày, bình thường, bình thường mà bạn gặp phải? Mọi thứ có thể để ngăn chúng can thiệp vào công việc thực sự của não - ghi lại những thứ quan trọng. Nó không bận tâm đến việc lưu những thứ nhảm chán; chúng không bao giờ vượt qua được bộ lọc "rõ ràng là điều này không quan trọng".

Bộ não của bạn biết điều gì là quan trọng như thế nào? Giả sử bạn đi bộ đường dài một ngày và một con hổ nhảy ra trước mặt bạn, điều gì xảy ra bên trong đầu và cơ thể bạn?

Nhịn hoạt động. Cảm xúc dâng trào. Hóa chất tăng vọt.

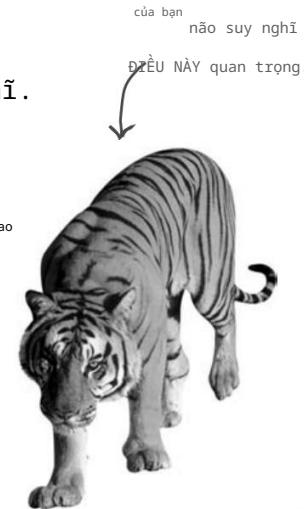
Và đó là cách bộ não của bạn biết...

Điều này chắc hẳn rất quan trọng! Đừng quên nhé!

Nhưng hãy tưởng tượng bạn đang ở nhà hoặc trong thư viện. Đó là một khu vực an toàn, ấm áp và không có hổ. Bạn đang học. Chuẩn bị cho kỳ thi. Hoặc cố gắng học một chủ đề kỹ thuật khó mà sép bạn nghĩ sẽ mất một tuần, nhiều nhất là mười ngày.

Chỉ có một vấn đề. Bộ não của bạn đang cố gắng giúp bạn một việc lớn. Nó đang cố gắng đảm bảo rằng nội dung rõ ràng là không quan trọng này không làm lộn xộn các nguồn tài nguyên khan hiếm. Các nguồn tài nguyên được sử dụng tốt hơn để lưu trữ những thứ thực sự lớn. Giống như hổ. Giống như nguy cơ hỏa hoạn. Giống như cách bạn không bao giờ nên trượt tuyệt trong quần short nữa.

Và không có cách đơn giản nào để nói với bộ não của bạn rằng, "Này bộ não, cảm ơn bạn rất nhiều, nhưng dù cuốn sách này có nhảm chán đến đâu, và dù tôi không cảm nhận được nhiều cảm xúc như thế nào trên thang Richter lúc này, thì tôi thực sự muốn bạn tiếp tục đọc nó."



của bạn
não suy nghĩ

ĐIỀU NÀY quan trọng.



Tuyệt. Chỉ còn
637 trang buồn tẻ, khô khan
và nhảm chán nữa thôi.

bộ não của bạn nghĩ
ĐIỀU NÀY không đáng
tiết kiệm.

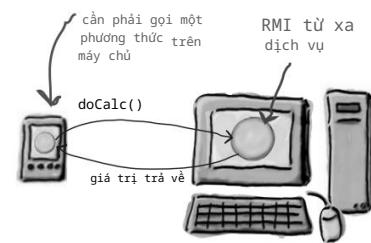
cách sử dụng cuốn sách này

Chúng ta coi người đọc theo kiểu "Đầu tiên" là người học.

Vậy thì cần phải làm gì để học hỏi cái gì đó? Đầu tiên, bạn phải lấy nó, sau đó hãy chắc chắn bạn không quên nó. Nó không phải là về việc đầy sự thật vào đầu bạn. Dựa trên nghiên cứu mới nhất về khoa học nhận thức, thần kinh học và tâm lý giáo dục, chúng ta cần nhiều hơn là văn bản trên một trang giấy. Chúng tôi biết điều gì làm bạn phản khích.

Một số nguyên tắc học tập Head First:

Làm cho nó trực quan. Hình ảnh dễ nhớ hơn nhiều so với chỉ từ ngữ và làm cho việc học hiệu quả hơn nhiều (cải thiện tới 89% trong các nghiên cứu về khả năng nhớ lại và chuyển giao). Nó cũng làm cho mọi thứ dễ hiểu hơn. Đặt các từ trong hoặc gần đồ họa mà chúng liên quan, thay vì ở dưới cùng hoặc trên một trang khác, và người học sẽ có khả năng tăng gấp đôi để giải quyết các vấn đề liên quan đến nội dung.



Sử dụng phong cách đàm thoại và cá nhân hóa. Trong các nghiên cứu gần đây, học sinh đạt kết quả tốt hơn tới 40% trong các bài kiểm tra sau khi học nội dung nói trực tiếp với người đọc, sử dụng phong cách đàm thoại ngôi thứ nhất thay vì sử dụng giọng điệu trang trọng. Kể chuyện thay vì thuyết giảng. Sử dụng ngôn ngữ thông thường. Đừng quá nghiêm túc.

Thật tệ khi trở thành một phương pháp trừu tượng.
Bạn không có cơ thể.

Bạn sẽ chú ý đến điều gì hơn: một bữa tiệc tối kích thích người bạn đồng hành hay một bài giảng?



trừu tượng void roam();

Không có phương pháp cơ thể
xết thúc bằng dấu chấm phẩy.

Khiến người học suy nghĩ sâu sắc hơn. Nói cách khác, trừ khi bạn chủ động uốn cong các tế bào thần kinh của mình, sẽ không có gì xảy ra nhiều trong đầu bạn. Người đọc phải có động lực, sự tham gia, tò mò và cảm hứng để giải quyết vấn đề, rút ra kết luận và tạo ra kiến thức mới.

Và để làm được điều đó, bạn cần những thử thách, bài tập, câu hỏi kích thích tư duy và các hoạt động liên quan đến cả hai bán cầu não.

và nhiều giác quan.

Có hợp lý không khi
nói Tub IS-A Bathroom?
Phòng tắm Là bồn tắm?
Hay là mối quan hệ Có?



↑ Thu hút và giữ sự chú ý của người đọc. Chúng ta đều đã từng trải qua cảm giác "Tôi thực sự muốn học điều này nhưng tôi không thể tỉnh táo sau trang đầu". Bộ não của bạn chú ý đến những thứ khác thường, thú vị, lạ lùng, bất mắt, bất ngờ.

Học một chủ đề mới, khó, kỹ thuật không nhất thiết phải nhảm chán. Bộ não của bạn sẽ học nhanh hơn nhiều nếu không.



Chạm vào cảm xúc của họ. Böyle giờ chúng ta biết rằng khả năng ghi nhớ một điều gì đó của bạn phụ thuộc phần lớn vào nội dung cảm xúc của nó. Bạn nhớ những gì bạn quan tâm. Bạn nhớ khi bạn cảm thấy điều gì đó. Không, chúng ta không nói đến những câu chuyện đau lòng về một cậu bé và ~~chú chó~~ Tôi là ~~chú chó~~ "Tôi là ~~chú chó~~ thông minh!" Chúng ta đang nói về những cảm xúc như ngạc nhiên, tò mò, vui vẻ, "cái gì thế...?", xuất hiện khi bạn giải được một câu đố, học được điều gì đó mà mọi người khác cho là khó, hoặc nhận ra bạn biết điều gì đó mà "tôi rành về kỹ thuật hơn anh" Bob làm nghề kỹ thuật thì không

Siêu nhận thức: suy nghĩ về suy nghĩ

Nếu bạn thực sự muốn học, và bạn muốn học nhanh hơn và sâu hơn, hãy chú ý đến cách bạn chú ý.
Nghĩ về cách bạn nghĩ. Học cách bạn học.

Hầu hết chúng ta không học các khóa học về siêu nhận thức hoặc lý thuyết học tập khi chúng ta lớn lên. Chúng ta được kỳ vọng là sẽ học, nhưng hiếm khi được dạy cách học.

Nhưng chúng tôi cho rằng nếu bạn đang cầm cuốn sách này, bạn thực sự muốn học các mẫu thiết kế. Và bạn có thể không muốn dành nhiều thời gian. Và bạn muốn nhớ những gì bạn đã đọc và có thể áp dụng nó. Và để làm được điều đó, bạn phải hiểu nó. Để tận dụng tối đa cuốn sách này, hoặc bất kỳ cuốn sách hay trải nghiệm học tập nào, hãy chịu trách nhiệm cho bộ não của bạn. Bộ não của bạn về nội dung này.

Bí quyết là khiến não bạn thấy tài liệu mới mà bạn đang học là Thực sự quan trọng.

Có ý nghĩa then chốt đối với sức khỏe của bạn. Quan trọng như một con hổ.

Nếu không, bạn sẽ phải liên tục đấu tranh với bộ não để ngăn không cho nội dung mới được ghi nhớ.

Vậy làm thế ~~hỗ~~^{để} não bạn nghĩ rằng Mẫu thiết kế quan trọng như một con hổ?

Có cách chậm rãi, tẻ nhạt, hoặc cách nhanh hơn, hiệu quả hơn. Cách chậm rãi là về sự lặp lại tuyệt đối. Rõ ràng là bạn biết rằng bạn có thể học và nhớ ngay cả những chủ đề buồn tẻ nhất, nếu bạn cứ liên tục nhấn mạnh vào cùng một thứ. Khi lặp lại đủ nhiều, não của bạn sẽ nói, "Điều này không có vẻ quan trọng với anh ấy, nhưng anh ấy cứ nhìn đi nhìn lại cùng một thứ, vì vậy tôi cho rằng nó phải như vậy."

Cách nhanh hơn là làm bất cứ điều gì giúp tăng hoạt động của não, đặc biệt là các loại hoạt động não khác nhau. Những điều ở trang trước là một phần lớn của giải pháp và tất cả đều là những điều đã được chứng minh là giúp não bạn hoạt động có lợi cho bạn. Ví dụ, các nghiên cứu cho thấy việc đặt các từ trong hình ảnh mà chúng mô tả (trái ngược với một nơi nào đó khác trên trang, như chú thích hoặc trong phần thân văn bản) khiến não bạn cố gắng hiểu cách các từ và hình ảnh liên quan, và điều này khiến nhiều tế bào thần kinh hoạt động hơn.

Nhiều tế bào thần kinh hoạt động hơn = nhiều cơ hội hơn để não bạn nhận ra rằng đây là điều đáng chú ý và có thể ghi lại.

Phong cách đàm thoại hữu ích vì mọi người có xu hướng chú ý nhiều hơn khi họ nhận thấy mình đang trong một cuộc trò chuyện, vì họ được kỳ vọng sẽ theo dõi và giữ lời. Điều tuyệt vời là, não của bạn không nhất thiết phải quan tâm đến việc "cuộc trò chuyện" là giữa bạn và một cuốn sách! Một khác, nếu phong cách viết trang trọng và khô khan, não của bạn sẽ cảm nhận nó theo cùng cách bạn trải nghiệm khi bị thuyết giảng trong khi ngồi trong một căn phòng đầy những người tham dự thụ động. Không cần phải thức.

Nhưng hình ảnh và phong cách trò chuyện chỉ là khởi đầu.

Tôi tự hỏi làm sao
tôi có thể đánh lừa
bộ não mình để nhớ
những điều này...



cách sử dụng cuốn sách này

Đây là những gì CHÚNG TÔI đã làm:

Chúng tôi sử dụng hình ảnh, vì não của bạn được điều chỉnh cho hình ảnh, không phải văn bản. Đó với não của bạn, một hình ảnh thực sự có giá trị bằng 1024 từ. Và khi văn bản và hình ảnh hoạt động cùng nhau, chúng tôi nhúng văn bản vào hình ảnh vì não của bạn hoạt động hiệu quả hơn khi văn bản nằm trong thứ mà văn bản đề cập đến, trái ngược với chủ thích hoặc được chôn trong văn bản ở đâu đó.

Chúng tôi sử dụng phương pháp trùng lặp, nói cùng một điều theo nhiều cách khác nhau , bằng nhiều loại phương tiện truyền thông khác nhau và nhiều giác quan khác nhau để tăng khả năng nội dung được mã hóa vào nhiều vùng não của bạn.

Chúng tôi sử dụng các khái niệm và hình ảnh theo những cách không ngờ tới vì não của bạn được điều chỉnh cho sự mới lạ, và chúng tôi sử dụng các hình ảnh và ý tưởng có ít nhất một số nội dung cảm xúc , vì não của bạn được điều chỉnh để chú ý đến sinh hóa của cảm xúc. Những gì khiến bạn cảm thấy một điều gì đó có nhiều khả năng được ghi nhớ hơn, ngay cả khi cảm giác đó không gì hơn là một chút hài hước, ngạc nhiên hoặc hứng thú.

Chúng tôi sử dụng phong cách đàm thoại cá nhân , vì não của bạn được điều chỉnh để chú ý nhiều hơn khi nó tin rằng bạn đang trong một cuộc trò chuyện hơn là khi nó nghĩ rằng bạn đang thụ động lắng nghe một bài thuyết trình. Não của bạn làm điều này ngay cả khi bạn đang đọc.

Chúng tôi đã đưa vào hơn 40 hoạt động, vì não của bạn được điều chỉnh để học và nhớ nhiều hơn khi bạn làm việc gì đó hơn là khi bạn đọc về những thứ đó. Và chúng tôi đã làm cho các bài tập trở nên đầy thử thách nhưng vẫn có thể thực hiện được, vì đó là điều mà hầu hết mọi người thích.

Chúng tôi sử dụng nhiều phong cách học tập khác nhau, vì bạn có thể thích các quy trình từng bước, trong khi người khác muốn hiểu bức tranh toàn cảnh trước, trong khi người khác chỉ muốn xem ví dụ về mã. Nhưng bất kể sở thích học tập của bạn là gì, mọi người đều được hưởng lợi khi thấy cùng một nội dung được thể hiện theo nhiều cách.

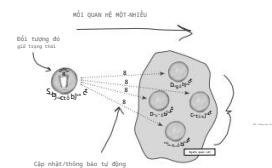
Chúng tôi bao gồm nội dung cho cả hai bán cầu não của bạn, vì bạn càng sử dụng nhiều bán cầu não thì khả năng học và ghi nhớ của bạn càng cao, đồng thời bạn có thể tập trung lâu hơn. Vì khi một bên não hoạt động thường có nghĩa là cho bên não kia cơ hội nghỉ ngơi, bạn có thể học tập hiệu quả hơn trong thời gian dài hơn.

Và chúng tôi đã đưa vào những câu chuyện và bài tập trình bày nhiều quan điểm hơn, vì não của bạn sẽ được điều chỉnh để học sâu hơn khi buộc phải đưa ra đánh giá và phán đoán.

Chúng tôi đưa vào các thử thách, với các bài tập và bằng cách đặt ra những câu hỏi không phải lúc nào cũng có câu trả lời trực tiếp, vì não của bạn được điều chỉnh để học và nhớ khi nó phải làm việc gì đó. Hãy nghĩ về điều này—you không thể có được vóc dáng chỉ bằng cách xem mọi người ở phòng tập thể dục. Nhưng chúng tôi đã cố gắng hết sức để đảm bảo rằng khi bạn làm việc chăm chỉ, đó là việc đúng đắn . Rằng bạn không tồn tại một nhánh cây để xử lý một ví dụ khó hiểu hoặc phân tích cú pháp văn bản khó, nhiều thuật ngữ chuyên ngành hoặc quá ngắn gọn.

Chúng tôi sử dụng con người. Trong các câu chuyện, ví dụ, hình ảnh, v.v., bởi vì, ừm, bởi vì bạn là một con người. Và não của bạn chú ý đến con người nhiều hơn là đến đồ vật.

Chúng tôi sử dụng phương pháp tiếp cận 80/20 . Chúng tôi cho rằng nếu bạn đang học tiến sĩ về thiết kế phần mềm, thì đây sẽ không phải là cuốn sách duy nhất của bạn. Vì vậy, chúng tôi không nói về mọi thứ. Chỉ những thứ bạn thực sự cần.



Chuyên gia về các mẫu

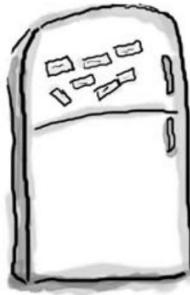


ĐIỂM ĐẦU TIÊN



Câu đố





Đây là những gì BẠN có thể làm để khuất phục bộ não của mình

Vậy là chúng tôi đã hoàn thành phần việc của mình. Phần còn lại tùy thuộc vào bạn. Những mẹo này là điểm khởi đầu; hãy lắng nghe não bộ của bạn và tìm ra điều gì hiệu quả với bạn và điều gì không. Hãy thử những điều mới.

cắt hình này ra và
dán lên tủ lạnh.

- 1.** Chậm lại. Bạn càng hiểu nhiều, bạn càng phải ghi nhớ ít hơn.

Đừng chỉ đọc. Hãy dừng lại và suy nghĩ. Khi cuốn sách hỏi bạn một câu hỏi, đừng chỉ lướt qua câu trả lời. Hãy tưởng tượng rằng có người thực sự đang hỏi câu hỏi đó. Bạn càng ép não mình suy nghĩ sâu sắc, bạn càng có nhiều cơ hội học hỏi và ghi nhớ.

- 2.** LÀM BÀI TẬP. VIẾT GHI CHÚ CỦA RIÊNG BẠN.

Chúng tôi đưa chúng vào, nhưng nếu chúng tôi làm thay bạn, thì cũng giống như có người khác làm bài tập thay bạn vậy. Và đừng chỉ nhìn vào các bài tập. Hãy dùng bút chì. Có rất nhiều bằng chứng cho thấy việc học hoạt động thể chất có thể tăng khả năng học tập.

trong khi

- 3.** ĐỌC "KHÔNG CÓ CÂU HỎI NÀO LÀ NGỜ NGẦN"

Nghĩa là tất cả chúng. Chúng không phải là thanh bên tuy chọn-chúng là một phần của nội dung cốt lõi! Đừng bỏ qua chúng.

- 5.** HÃY COI ĐÂY LÀ ĐIỀU CUỐI CÙNG BẠN ĐỌC TRƯỚC KHI ĐI NGỦ.
HOẶC ÍT NHẤT LÀ ĐIỀU CUỐI CÙNG. THÁCH THỨC

Một phần của quá trình học (đặc biệt là quá trình chuyển sang trí nhớ dài hạn) diễn ra sau khi bạn đặt cuốn sách xuống. Bộ não của bạn cần thời gian riêng để xử lý thêm. Nếu bạn đưa vào một cái gì đó mới trong thời gian xử lý đó, một số thông tin bạn vừa học sẽ bị mất.

- 6.** UỐNG NƯỚC. UỐNG THẬT NHIỀU NƯỚC.

Bộ não của bạn hoạt động tốt nhất khi được cung cấp đủ nước. Mất nước (có thể xảy ra trước khi bạn cảm thấy khát) làm giảm chức năng nhận thức.

- 7.** HÃY NÓI VỀ ĐIỀU ĐÓ. NÓI TO LÊN.

Việc nói chuyện kích hoạt một phần khác của não. Nếu bạn đang cố gắng hiểu điều gì đó, hoặc tăng khả năng nhớ lại sau này, hãy nói to. Tốt hơn nữa, hãy cố gắng giải thích điều đó cho người khác. Bạn sẽ học nhanh hơn và có thể khám phá ra những ý tưởng mà bạn không biết là có khi bạn đọc về nó.

- 8.** HÃY LẮNG NGHE BỘ Não CỦA BẠN.

Hãy chú ý xem não bạn có bị quá tải không. Nếu bạn thấy mình bắt đầu lướt qua bề mặt hoặc quên những gì vừa đọc, thì đã đến lúc nghỉ ngơi. Khi bạn vượt qua một điểm nhất định, bạn sẽ không học nhanh hơn bằng cách cố nhồi nhét thêm, và bạn thậm chí có thể làm hỏng quá trình.

- 9.** CẢM THẤY THỨ GIÀ ĐÓ!

Bộ não của bạn cần biết rằng điều này quan trọng. Hãy tham gia vào các câu chuyện. Tự nghĩ ra chủ thích cho các bức ảnh. Tham thờ về một trò đùa tệ hại vẫn tốt hơn là không cảm thấy gì cả.

- 10.** THIẾT KẾ THỨ GIÀ ĐÓ!

Áp dụng điều này vào một thứ mới mà bạn đang thiết kế hoặc tái cấu trúc một dự án cũ. Chỉ cần làm một điều gì đó để có thêm kinh nghiệm ngoài các bài tập và hoạt động trong cuốn sách này. Tất cả những gì bạn cần là một cây bút chì và một ván để để giải quyết... một ván đẽ có thể được hưởng lợi từ một hoặc nhiều mẫu thiết kế.

cách sử dụng cuốn sách này

Đọc Tôi

Đây là một trải nghiệm học tập, không phải là một cuốn sách tham khảo. Chúng tôi có tinh loại bỏ mọi thứ có thể cản trở việc học bất cứ điều gì chúng tôi đang làm tại thời điểm đó trong cuốn sách. Và lần đầu tiên, bạn cần bắt đầu từ đầu, vì cuốn sách đưa ra các giả định về những gì bạn đã thấy và đã học.

Chúng tôi sử dụng một UML
giả đơn giản hơn, đã được sửa đổi

Chúng tôi sử dụng các sơ đồ đơn giản giống UML.

Mặc dù có khả năng cao là bạn đã từng gặp UML, nhưng nó không được đề cập trong sách và nó không phải là điều kiện tiên quyết để đọc sách. Nếu bạn chưa từng thấy UML trước đây, đừng lo lắng, chúng tôi sẽ cung cấp cho bạn một vài mẹo trong quá trình học. Nói cách khác, bạn sẽ không phải lo lắng về Design Patterns và UML cùng một lúc. Các sơ đồ của chúng tôi "giống UML" -- mặc dù chúng tôi cố gắng trung thành với UML nhưng đôi khi chúng tôi phá vỡ các quy tắc một chút, thường là vì lý do nghệ thuật ích kỷ của riêng chúng tôi.

Giám đốc

lấyPhim
lấy giải Oscar()
lấyKevinBaconDegrees()

Chúng tôi không đề cập đến mọi Mẫu thiết kế đã từng được tạo ra.

Có rất nhiều Mẫu thiết kế: Các mẫu nền tảng ban đầu (được gọi là các mẫu GoF), các mẫu J2EE của Sun, các mẫu JSP, các mẫu kiến trúc, các mẫu thiết kế trò chơi và nhiều hơn nữa. Nhưng mục tiêu của chúng tôi là đảm bảo rằng cuốn sách có trọng lượng nhẹ hơn người đọc nó, vì vậy chúng tôi không đề cập đến tất cả chúng ở đây. Trọng tâm của chúng tôi là các mẫu cốt lõi quan trọng từ các mẫu GoF gốc và đảm bảo rằng bạn thực sự, thực sự, hiểu sâu sắc cách thức và thời điểm sử dụng chúng. Bạn sẽ tìm thấy một cái nhìn tổng quan về một số mẫu khác (những mẫu mà bạn ít có khả năng sử dụng) trong phần phụ lục. Trong mọi trường hợp, sau khi hoàn thành Head First Design Patterns, bạn sẽ có thể chọn bất kỳ danh mục nào và nhanh chóng bắt kịp.

Các hoạt động KHÔNG phải là tùy chọn.

Các bài tập và hoạt động này không phải là phần bổ sung; chúng là một phần nội dung cốt lõi của cuốn sách. Một số trong số chúng giúp tăng cường trí nhớ, một số để hiểu và một số để giúp bạn áp dụng những gì đã học. Đừng bỏ qua các bài tập. Trò chơi ô chữ là những thứ duy nhất bạn không phải làm, nhưng chúng rất tốt để giúp não bạn có cơ hội suy nghĩ về các từ trong một ngữ cảnh khác.

Chúng tôi sử dụng từ "composition" theo nghĩa chung của OO, linh hoạt hơn so với cách sử dụng "composition" nghiêm ngặt của UML.

Khi chúng tôi nói "một đối tượng được tạo thành từ một đối tượng khác" thì chúng tôi muốn nói rằng chúng có liên quan với nhau theo mối quan hệ CÓ-MỘT. Cách sử dụng của chúng tôi phản ánh cách sử dụng thuật ngữ truyền thống và là cách sử dụng trong văn bản GoF (Bạn sẽ tìm hiểu thuật ngữ đó là gì sau). Giản dị hơn, UML đã tinh chỉnh thuật ngữ này thành một số loại thành phần. Nếu bạn là chuyên gia về UML, bạn vẫn có thể đọc cuốn sách và bạn sẽ có thể dễ dàng ánh xạ việc sử dụng thành phần thành các thuật ngữ tinh chỉnh hơn khi bạn đọc.

Sự duy thừa này là có chủ ý và quan trọng.

Một điểm khác biệt rõ rệt trong sách Head First là chúng tôi muốn bạn thực sự hiểu được nó. Và chúng tôi muốn bạn đọc hết cuốn sách và nhớ lại những gì bạn đã học. Hầu hết các sách tham khảo không có mục tiêu là ghi nhớ và nhớ lại, nhưng cuốn sách này là về việc học, vì vậy bạn sẽ thấy một số khái niệm giống nhau xuất hiện nhiều lần.

Các ví dụ về mã được trình bày càng ngắn gọn càng tốt.

Độc giả của chúng tôi cho biết rằng thật khó chịu khi phải lội qua 200 dòng mã để tìm hai dòng họ cần hiểu. Hầu hết các ví dụ trong cuốn sách này được trình bày trong ngữ cảnh nhỏ nhất có thể, do đó phần bạn đang cố gắng học sẽ rõ ràng và đơn giản. Đừng mong đợi tất cả các mã đều mạnh mẽ hoặc thậm chí là hoàn chỉnh—các ví dụ được viết riêng cho mục đích học và không phải lúc nào cũng có đầy đủ chức năng.

Trong một số trường hợp, chúng tôi không đưa vào tất cả các câu lệnh import cần thiết, nhưng chúng tôi cho rằng nếu bạn là một lập trình viên Java, bạn biết rằng ArrayList nằm trong java.util, chẳng hạn. Nếu các lệnh import không phải là một phần của API J2SE lỗi thông thường, chúng tôi sẽ đề cập đến nó. Chúng tôi cũng đã đưa tất cả mã nguồn lên web để bạn có thể tải xuống. Bạn sẽ tìm thấy nó tại <http://www.headfirstlabs.com/books/hfdp/>

Ngoài ra, vì mục đích tập trung vào khía cạnh học tập của mã, chúng tôi đã không đưa các lớp của mình vào các gói (nói cách khác, tất cả chúng đều nằm trong gói mặc định của Java). Chúng tôi không khuyến nghị điều này trong thế giới thực và khi bạn tải xuống các ví dụ mã từ cuốn sách này, bạn sẽ thấy rằng tất cả các lớp đều nằm trong các gói.

Các bài tập 'Sức mạnh não bộ' không có đáp án.

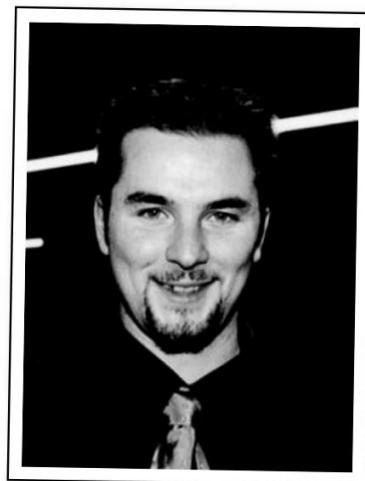
Đối với một số bài tập, không có câu trả lời đúng, và đối với những bài tập khác, một phần của trải nghiệm học tập các hoạt động Brain Power là bạn phải quyết định xem câu trả lời của mình có đúng hay không và khi nào thì đúng. Trong một số bài tập Brain Power, bạn sẽ tìm thấy những gợi ý chỉ cho bạn hướng đi đúng.

nhóm đánh giá sớm

Người đánh giá công nghệ

Valentin Crettaz

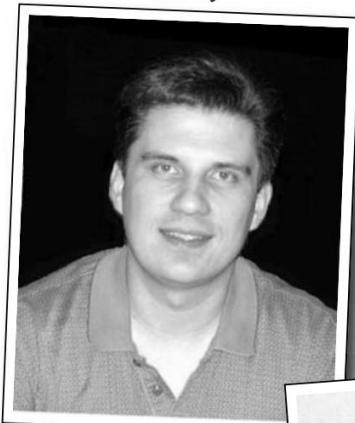
Jef Cumps



Barney Marispini



Ike Van Atta ↘

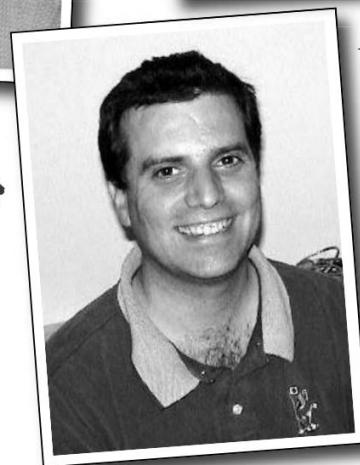


Người lãnh đạo dùng
tầm của Đội đánh giá
Phuyên sâu HFDP.

Jason Menard



Đánh dấu Spritzler



Johannes deJong ↗



Đirk Schreckmann



Philippe Maquet

Tưởng nhớ Philippe Maquet

1960-2004

Trình độ chuyên môn tuyệt vời, lòng nhiệt huyết không ngừng nghỉ và sự quan tâm sâu sắc đến người học của bạn sẽ luôn truyền cảm hứng cho chúng tôi.

Chúng tôi sẽ không bao giờ quên bạn.

Lời cảm ơn

Tại O'Reilly:

Xin gửi lời cảm ơn chân thành nhất đến Mike Loukides tại O'Reilly, người đã khởi xướng tất cả và giúp định hình khái niệm Head First thành một series. Và xin gửi lời cảm ơn chân thành đến động lực thúc đẩy đằng sau Head First, Tim O'Reilly.

Cảm ơn Kyle Hart, "mẹ của series" Head First thông minh, ngôi sao nhạc rock and roll Ellie Volkhausen vì thiết kế bìa đầy cảm hứng của cô ấy và cũng cảm ơn Colleen Gorman vì bản thảo khó khăn của cô ấy. Cuối cùng, cảm ơn Mike Hendrickson vì đã ủng hộ cuốn sách Design Patterns này và xây dựng nhóm.

Những người đánh giá cao chúng tôi:

Chúng tôi vô cùng biết ơn giám đốc đánh giá kỹ thuật Johannes deJong. Anh là anh hùng của chúng tôi, Johannes. Và chúng tôi vô cùng trân trọng những đóng góp của đồng quản lý nhóm đánh giá Javaranch, cố Philippe Maquet. Anh đã một mình làm bừng sáng cuộc sống của hàng nghìn nhà phát triển, và tác động của anh đối với cuộc sống của họ (và của chúng tôi) là mãi mãi.

Jef Cumps rất giỏi trong việc tìm ra vấn đề trong các chương dự thảo của chúng tôi, và một lần nữa tạo nên sự khác biệt lớn cho cuốn sách. Cảm ơn Jef! Valentin Cretaz (anh chàng AOP), người đã cùng chúng tôi ngay từ cuốn sách Head First đầu tiên, đã chứng minh (như mọi khi) chúng tôi thực sự cần đến chuyên môn kỹ thuật và hiểu biết sâu sắc của anh ấy như thế nào. Bạn làm Valentin tuyệt vời (nhưng lại thua cuộc).

Hai người mới tham gia nhóm đánh giá HF, Barney Marispini và Ike Van Atta đã làm rất tốt cuốn sách này—các bạn đã cho chúng tôi một số phản hồi thực sự quan trọng. Cảm ơn vì đã tham gia nhóm.

Chúng tôi cũng nhận được một số trợ giúp kỹ thuật tuyệt vời từ các điều phối viên/chuyên gia Javaranch Mark Spritzler, Jason Menard, Dirk Schreckmann, Thomas Paul và Margarita Isaeva. Và như thường lệ, đặc biệt cảm ơn Trưởng nhóm Javaranch.com, Paul Wheaton.

Cảm ơn những người vào chung kết cuộc thi “Chọn bìa mẫu thiết kế đầu tiên” của Javaranch. Người chiến thắng, Si Brewster, đã gửi bài luận chiến thắng thuyết phục chúng tôi chọn người phụ nữ mà bạn thấy trên bìa của chúng tôi. Những người vào chung kết khác bao gồm Andrew Esse, Gian Franco Casula, Helen Crosbie, Pho Tek, Helen Thomas, Sateesh Kommineni và Jeff Fisher.

vẫn còn nhiều lời cảm ơn hơn nữa

Thậm chí còn nhiều người hơn*

Từ Eric và Elisabeth

Viết một cuốn sách Head First là một chuyến đi thú vị với hai hướng dẫn viên du lịch tuyệt vời: Kathy Sierra và Bert Bates. Với Kathy và Bert, bạn vứt bỏ mọi quy ước viết sách và bước vào một thế giới đầy những câu chuyện kể, lý thuyết học tập, khoa học nhận thức và văn hóa đại chúng, nơi người đọc luôn thống trị. Cảm ơn cả hai bạn đã cho chúng tôi bước vào thế giới tuyệt vời của các bạn; chúng tôi hy vọng chúng tôi đã thực hiện công lý cho Head First. Nghiêm túc mà nói, điều này thật tuyệt vời. Cảm ơn vì tất cả những chỉ dẫn cẩn thận của các bạn, vì đã thúc đẩy chúng tôi tiến về phía trước và hơn hết là vì đã tin tưởng chúng tôi (với đứa con của các bạn). Cả hai bạn chắc chắn đều "thông minh một cách đặc ác" và cũng là những người 29 tuổi sành điệu nhất mà chúng tôi biết. Vậy thì... tiếp theo là gì?

Xin chân thành cảm ơn Mike Loukides và Mike Hendrickson. Mike L. đã đồng hành cùng chúng tôi trong suốt chàng đường. Mike, phản hồi sâu sắc của bạn đã giúp định hình cuốn sách và sự động viên của bạn đã giúp chúng tôi tiến lên phía trước. Mike H., cảm ơn vì sự kiên trì của bạn trong suốt năm năm qua trong nỗ lực giúp chúng tôi viết một cuốn sách mẫu; cuối cùng chúng tôi đã làm được và chúng tôi rất vui vì đã chờ đợi Head First.

Xin gửi lời cảm ơn đặc biệt đến Erich Gamma, người đã vượt xa nhiệm vụ khi đánh giá cuốn sách này (anh ấy thậm chí còn mang theo bản thảo khi đi nghỉ). Erich, sự quan tâm của bạn đối với cuốn sách này đã truyền cảm hứng cho chúng tôi và bài đánh giá kỹ thuật toàn diện của bạn đã cải thiện nó một cách đáng kể. Cũng xin cảm ơn toàn bộ Gang of Four vì sự ủng hộ và quan tâm của họ, và vì đã xuất hiện đặc biệt trong Objectville.
Chúng tôi cũng biết ơn Ward Cunningham và cộng đồng mẫu đã tạo ra Portland Pattern Repository - một nguồn tài nguyên không thể thiếu để chúng tôi viết cuốn sách này.

Cảm ơn các ngôi làng để viết một cuốn sách kỹ thuật: Bill Pugh và Ken Arnold đã cho chúng tôi lời khuyên chuyên môn về Singleton. Joshua Marinacci đã cung cấp những mẹo và lời khuyên về Swing tuyệt vời. John Brewer "Tại sao lại là Việt?" bài báo đã truyền cảm hứng cho SimUDuck (và chúng tôi rất vui vì anh ấy cũng thích viet). Dan Friedman đã truyền cảm hứng cho ví dụ về Little Singleton. Daniel Steinberg đóng vai trò là "người liên lạc kỹ thuật" và mang lối hỗ trợ cảm xúc của chúng tôi. Và cảm ơn James Dempsey của Apple đã cho phép chúng tôi sử dụng bài hát MVC của anh ấy.

Cuối cùng, xin gửi lời cảm ơn cá nhân đến nhóm đánh giá Javaranch vì những đánh giá tuyệt vời và sự hỗ trợ nồng nhiệt của họ. Có nhiều bạn trong cuốn sách này hơn bạn biết.

Từ Kathy và Bert

Chúng tôi muốn cảm ơn Mike Hendrickson vì đã tìm thấy Eric và Elisabeth... nhưng chúng tôi không thể. Nhờ hai người này, chúng tôi phát hiện ra (khiến chúng tôi kinh hoàng) rằng chúng tôi không phải là những người duy nhất có thể làm một cuốn sách Head First. ;) Tuy nhiên, nếu độc giả muốn tin rằng Kathy và Bert thực sự là những người đã làm những điều tuyệt vời trong cuốn sách, thì chúng tôi là ai để chỉnh sửa chúng?

*Số lượng lớn lời cảm ơn là vì chúng tôi đang kiểm tra lý thuyết rằng mọi người được đề cập trong lời cảm ơn của một cuốn sách sẽ mua ít nhất một bản, có thể nhiều hơn, với người thân và mọi thứ. Nếu bạn muốn được ghi nhận trong lời cảm ơn của cuốn sách tiếp theo của chúng tôi và bạn có một gia đình lớn, hãy viết thư cho chúng tôi.

1 Giới thiệu về Mẫu thiết kế

Chào mừng đến với
Thiết kế Các mẫu

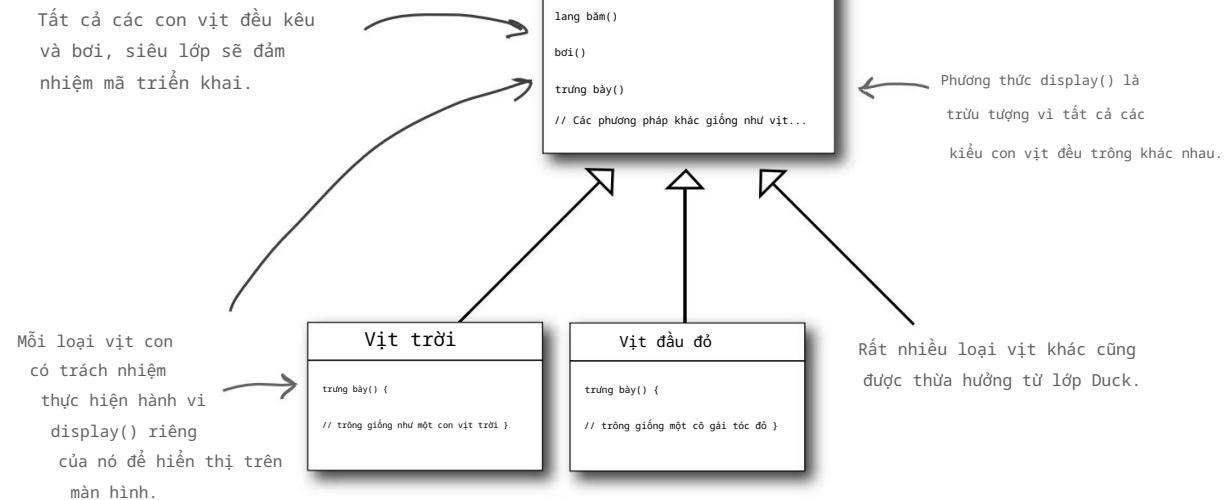
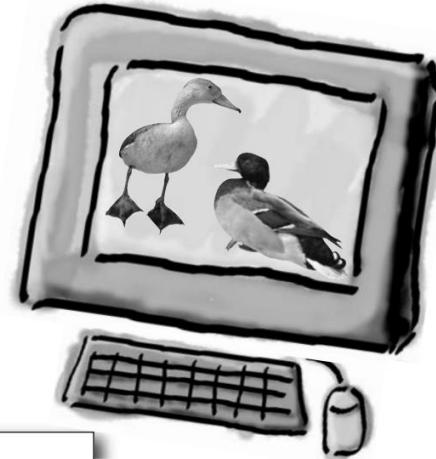


Ai đó đã giải quyết vấn đề của bạn. Trong chương này, bạn sẽ học tại sao (và làm thế nào) bạn có thể khai thác trí tuệ và bài học kinh nghiệm của những nhà phát triển khác đã đã đi xuống cùng một con đường vấn đề thiết kế và sống sót sau chuyến đi. Trước khi chúng ta hoàn thành, chúng ta sẽ xem xét việc sử dụng và lợi ích của các mẫu thiết kế, xem xét một số nguyên tắc thiết kế OO chính và đi qua một ví dụ về cách một mẫu hoạt động. Cách tốt nhất để sử dụng các mẫu là tải nǎo của bạn với chúng và sau đó nhận ra những vị trí trong thiết kế và ứng dụng hiện có của bạn nơi bạn có thể áp dụng chúng. Thay vì sử dụng lại mã, với các mẫu, bạn sẽ có được kinh nghiệm sử dụng lại.

SimUDuck

Mọi chuyện bắt đầu với một ứng dụng SimUDuck đơn giản

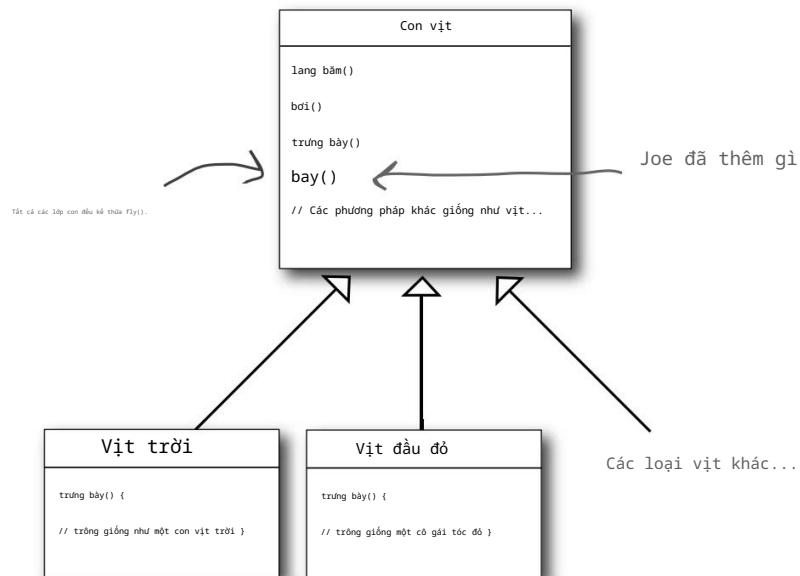
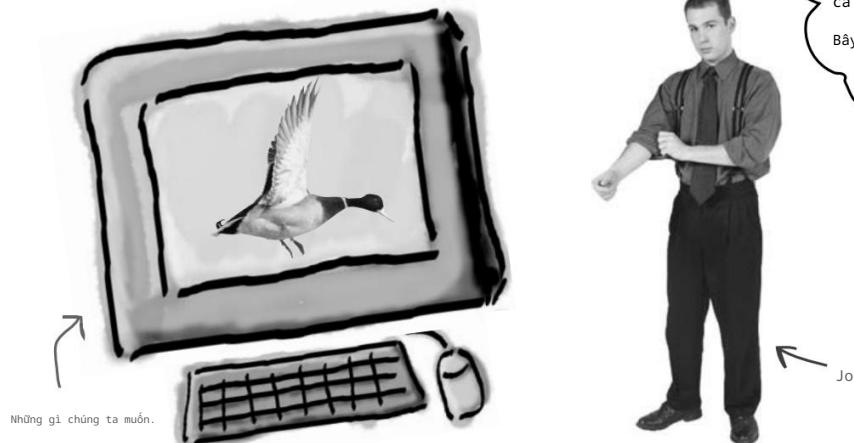
Joe làm việc cho một công ty sản xuất trò chơi mô phỏng ao vịt rất thành công, SimUDuck. Trò chơi có thể hiển thị nhiều loài vịt bơi và kêu tiếng kêu quạc quạc. Những người thiết kế ban đầu của hệ thống đã sử dụng các kỹ thuật OO chuẩn và tạo ra một siêu lớp Duck mà tất cả các loại vịt khác đều kế thừa.



Trong năm qua, công ty đã chịu áp lực ngày càng tăng từ các đối thủ cạnh tranh. Sau một tuần họp động não ngoài công ty về golf, các giám đốc điều hành công ty nghĩ rằng đã đến lúc cần có một sự đổi mới lớn. Họ cần một cái gì đó thực sự ấn tượng để trình bày tại cuộc họp cổ đông sắp tới ở Maui vào tuần tới.

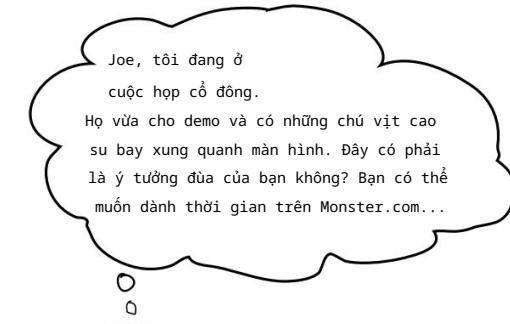
Nhưng bây giờ chúng ta cần những chú vịt BAY

Các giám đốc điều hành đã quyết định rằng vịt bay chính là thứ mà trình mô phỏng cần để đánh bại các đối thủ mô phỏng vịt khác. Và tất nhiên, quản lý của Joe đã nói với họ rằng sẽ không có vấn đề gì nếu Joe chỉ cần làm một cái gì đó trong một tuần. "Rốt cuộc", ông chủ của Joe nói, "anh ấy là một lập trình viên OO... có thể khó đến mức nào?"



có gì đó không ổn

Nhưng có điều gì đó khủng khiếp đã xảy ra...

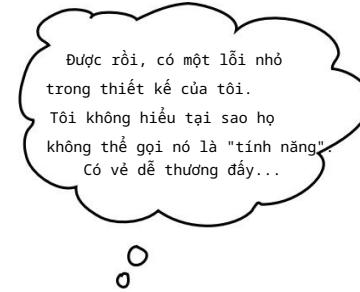
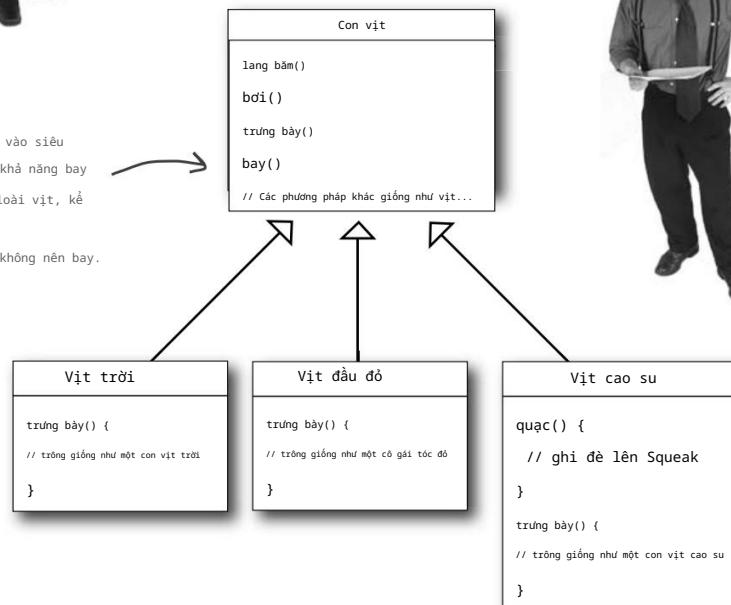


Joe đã không nhận ra rằng không phải tắt cả
các lớp con của Duck phải bay. Khi Joe
thêm hành vi mới vào siêu lớp Duck,
anh ấy cũng thêm hành vi không phù hợp với
một số lớp con Duck. Vậy giờ anh ấy có
các đối tượng vô tri vô giác bay trong
chương trình SimUDuck.

Bản cập nhật cục bộ cho mã đã gây ra tác
dụng phụ không cục bộ (vịt cao su bay)!

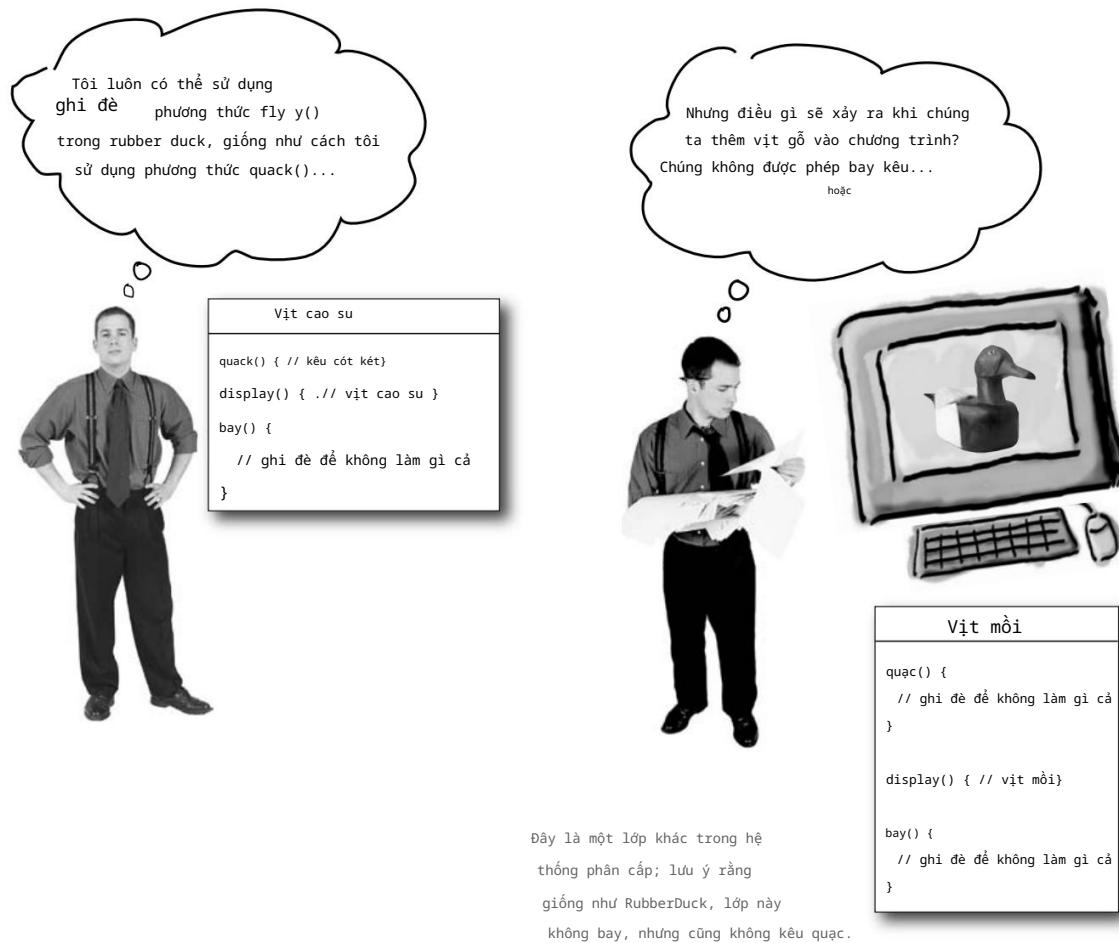
Bằng cách đưa fly() vào siêu
lớp, ông đã trao khả năng bay
cho TẤT CẢ các loài vịt, kể

cá những loài không nên bay.



Vịt cao su không kêu, vì
vậy quack() sẽ được ghi
đè thành "Squeak".

Joe nghĩ về việc thừa kế...



Chuốt bút chì của bạn

Trong các nhược điểm sau đây, nhược điểm nào là của việc sử dụng tính kế thừa để cung cấp hành vi Duck? (Chọn tất cả các đáp án đúng.)

- A. Mã được sao chép giữa các lớp con.
- B. Thay đổi hành vi trong thời gian chạy rất khó khăn.
- C. Chúng ta không thể bắt vịt nhảy múa.
- D. Khó có thể hiểu hết tất cả các hành vi của loài vịt.
- E. Vịt không thể vừa bay vừa kêu cùng một lúc.
- F. Những thay đổi có thể vô tình ảnh hưởng đến những con vịt khác.

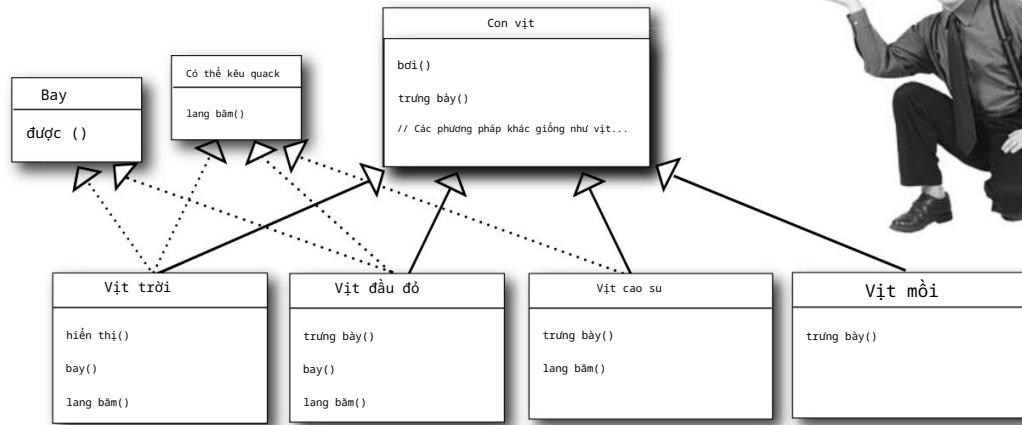
đi truyền không phải là câu trả lời

Thế còn giao diện thì sao?

Joe nhận ra rằng kế thừa có lẽ không phải là câu trả lời, vì anh vừa nhận được một bản ghi nhớ nói rằng các giám đốc điều hành hiện muốn cập nhật sản phẩm sáu tháng một lần (theo cách mà họ vẫn chưa quyết định). Joe biết rằng thông số kỹ thuật sẽ tiếp tục thay đổi và anh sẽ buộc phải xem xét và có thể ghi đè fly() và quack() cho mọi lớp con Duck mới được thêm vào chương trình... mãi mãi.

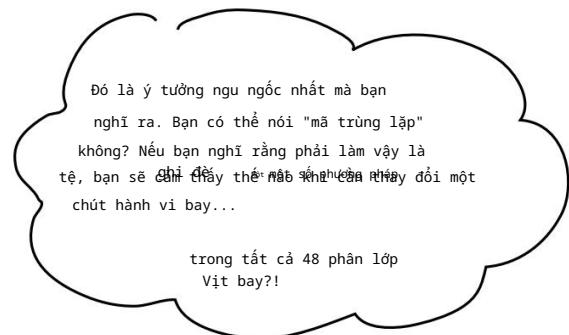
Vì vậy, ông cần một cách sạch hơn để chỉ một số (nhưng không phải tất cả) loài vịt biết bay hoặc kêu.

Tôi có thể lấy fly() ra khỏi siêu lớp Duck và tạo một phương thức Giao diện Flyable() fly y().
Theo cách đó, chỉ những con vịt kêu fly y là giả sử mới triển khai giao diện đó và có phương thức fly y()... và tôi cũng có thể tạo một Quackable, vì không phải tất cả các con vịt đều có thể kêu quack.



BẠN nghĩ gì về thiết kế này?

giới thiệu về Mẫu thiết kế

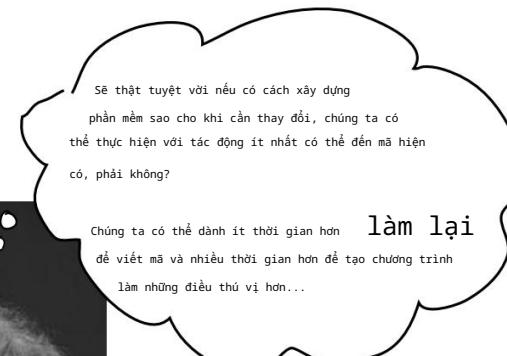


Cái gì sẽ Bạn bạn sẽ làm gì nếu là Joe?

Chúng ta biết rằng không phải tất cả các lớp con đều phải có hành vi bay hoặc kêu, vì vậy kế thừa không phải là câu trả lời đúng. Nhưng trong khi việc để các lớp con triển khai Flyable và/hoặc Quackable giải quyết một phần vấn đề (không có vịt cao su bay không phù hợp), nó hoàn toàn phá hủy việc sử dụng lại mã cho các hành vi đó, vì vậy nó chỉ tạo ra một cơn ác mộng bảo trì. Và tất nhiên có thể có nhiều hơn một loại hành vi bay ngay cả trong số những con vịt biết bay...

Lúc này, bạn có thể đang chờ một Mẫu thiết kế cưỡi ngựa trắng đến và cứu vãn tình hình. Nhưng điều đó có gì vui? Không, chúng ta sẽ tìm ra giải pháp theo cách cũ—

bằng cách áp dụng các nguyên tắc thiết kế phần mềm OO tốt.



sự thay đổi là hằng số

Một hằng số trong phát triển phần mềm

Được rồi, điều gì bạn luôn có thể tin tưởng trong phát triển phần mềm?

Bất kể bạn làm việc ở đâu, đang xây dựng thứ gì hay đang lập trình bằng ngôn ngữ nào, thì hằng số thực sự nào sẽ luôn ở bên bạn?

THAY ĐỔI

(sử dụng gương để xem câu trả lời)

Cho dù bạn thiết kế ứng dụng tốt đến đâu, theo thời gian ứng dụng cũng phải phát triển và thay đổi, nếu không nó sẽ chết.



Chuốt bút chì của bạn

Có nhiều thứ có thể thúc đẩy sự thay đổi. Liệt kê một số lý do khiến bạn phải thay đổi mã trong ứng dụng của mình (chúng tôi đưa ra một số lý do để giúp bạn bắt đầu).

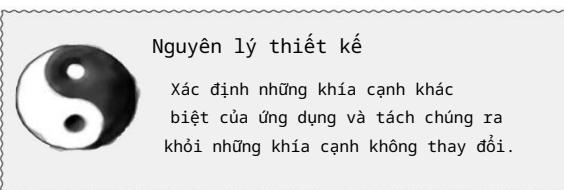
Khách hàng hoặc người dùng của tôi quyết định họ muốn thứ gì đó khác hoặc muốn chức năng mới.

Công ty tôi quyết định sẽ hợp tác với một nhà cung cấp cơ sở dữ liệu khác và cũng mua dữ liệu từ một nhà cung cấp khác sử dụng định dạng dữ liệu khác. Argh!

Tập trung vào vấn đề...

Vì vậy, chúng ta biết rằng việc sử dụng kế thừa không hiệu quả lắm, vì hành vi của vịt liên tục thay đổi giữa các lớp con và không phù hợp để tất cả các lớp con đều có những hành vi đó. Giao diện Flyable và Quackable thoạt đầu nghe có vẻ hứa hẹn—chỉ những con vịt thực sự biết bay mới là Flyable, v.v.—ngoại trừ giao diện Java không có mã triển khai, do đó không thể sử dụng lại mã. Và điều đó có nghĩa là bắt cứ khi nào bạn cần sửa đổi một hành vi, bạn buộc phải theo dõi và thay đổi nó trong tất cả các lớp con khác nhau nơi hành vi đó được định nghĩa, có thể gây ra lỗi mới trong quá trình này!

May mắn thay, có một nguyên tắc thiết kế dành riêng cho tình huống này.



Nguyên tắc thiết kế đầu tiên trong số nhiều
nguyên tắc thiết kế của chúng tôi. Chúng tôi sẽ dành nhiều
thời gian hơn cho những nguyên tắc này trong suốt cuốn sách.

Nói cách khác, nếu có một khía cạnh nào đó trong mã của bạn thay đổi, chẳng hạn như với mỗi yêu cầu mới, thì bạn biết rằng có một hành vi cần được loại bỏ và tách biệt khỏi tất cả những thứ không thay đổi.

Đây là một cách khác để suy nghĩ về nguyên tắc này: lấy các phần thay đổi và đóng gói chúng lại để sau này bạn có thể thay đổi hoặc mở rộng các phần thay đổi mà không ảnh hưởng đến các phần không thay đổi.

Mặc dù khái niệm này đơn giản, nhưng nó tạo thành cơ sở cho hầu hết mọi mẫu thiết kế. Tất cả các mẫu đều cung cấp một cách để một số phần của hệ thống thay đổi độc lập với tất cả các phần khác.

Được rồi, đến lúc đưa hành vi của vịt ra khỏi các lớp Việt!

Lấy những thay đổi và

“đóng gói” chúng lại để nó
không ảnh hưởng đến phần còn lại của mã.

Kết quả là gì? Ít hậu

quả không mong muốn từ việc
thay đổi mã và linh hoạt hơn
trong hệ thống của bạn!

rút ra những gì thay đổi

Phân biệt những gì thay đổi với những gì vẫn giữ nguyên

Chúng ta bắt đầu từ đâu? Theo như chúng tôi biết, ngoài các vấn đề với `fly()` và `quack()`, lớp Duck hoạt động tốt và không có phần nào khác của nó có vẻ thay đổi hoặc thay đổi thường xuyên. Vì vậy, ngoài một vài thay đổi nhỏ, chúng ta sẽ giữ nguyên lớp Duck.

Bây giờ, để tách "các phần thay đổi khỏi các phần không đổi", chúng ta sẽ tạo hai tập hợp các lớp (hoàn toàn tách biệt với Duck), một cho `fly` và một cho `quack`. Mỗi tập hợp các lớp sẽ chứa tất cả các triển khai của hành vi tương ứng của chúng. Ví dụ, chúng ta có thể có một lớp triển khai `quacking`, một lớp khác triển khai `squeaking` và một lớp khác triển khai `silence`.

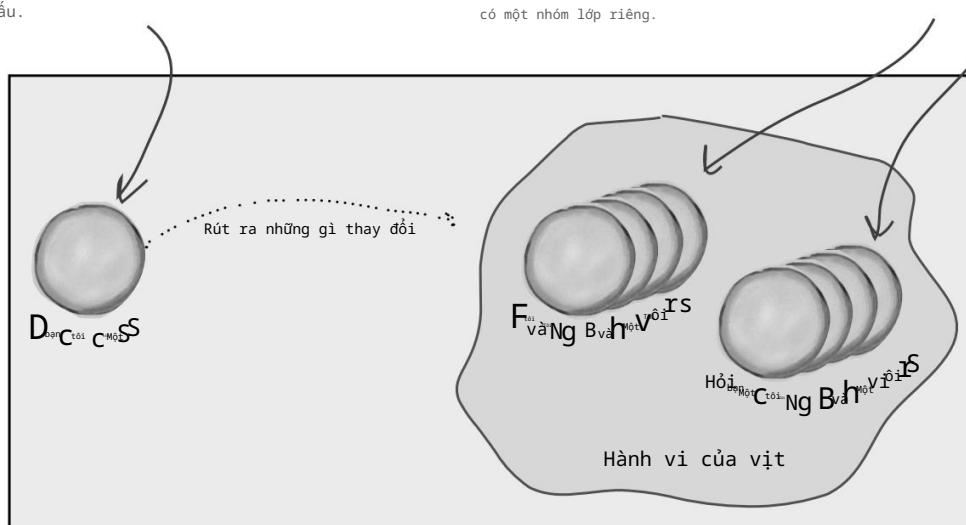
Chúng ta biết rằng `fly()` và `quack()` là các phần của lớp Duck
thay đổi tùy theo từng loài vịt.

Để tách các hành vi này khỏi lớp Duck, chúng ta sẽ kéo cả hai phương pháp vào một tập hợp ngoài của lớp Duck và tạo ra một
các lớp mới để biểu diễn từng hành vi.

Lớp Duck vẫn là siêu lớp của tất
cả các loài vịt, nhưng chúng ta
đang loại bỏ các hành vi bay
và kêu và đưa chúng vào một lớp khác
kết cấu.

Bây giờ, tiếng bay và tiếng quacking đều
có một nhóm lớp riêng.

Nhiều hành vi khác
nhau đang được thực hiện
để sống ở đây.

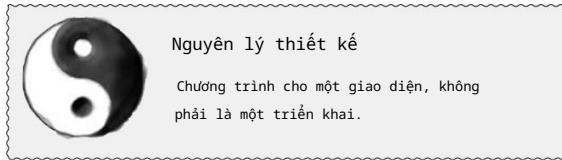


Thiết kế hành vi của vịt

Vậy chúng ta sẽ thiết kế tập hợp các lớp thực hiện hành vi fly và quack như thế nào?

Chúng tôi muốn giữ mọi thứ linh hoạt; sau cùng, chính sự thiếu linh hoạt trong hành vi của vịt đã khiến chúng tôi gặp rắc rối ngay từ đầu. Và chúng tôi biết rằng chúng tôi muốn gán hành vi cho các thể hiện của Duck. Ví dụ, chúng tôi có thể muốn khởi tạo một thể hiện MallardDuck mới và khởi tạo nó bằng một loại hành vi bay cụ thể. Và trong khi chúng tôi ở đó, tại sao không đảm bảo rằng chúng tôi có thể thay đổi hành vi của một con vịt một cách động? Nói cách khác, chúng tôi nên bao gồm các phương thức thiết lập hành vi trong các lớp Duck để chúng tôi có thể, chẳng hạn, thay đổi hành vi bay của MallardDuck khi chạy.

Với những mục tiêu này, chúng ta hãy xem xét nguyên tắc thiết kế thứ hai:



Chúng tôi sẽ sử dụng một giao diện để biểu diễn từng hành vi - ví dụ như FlyBehavior và QuackBehavior - và mỗi lần triển khai một hành vi sẽ triển khai một trong những giao diện đó.

Vì vậy, lần này không phải là các lớp Duck triển khai giao diện flying ying và quacking. Thay vào đó, chúng ta sẽ tạo một tập hợp các lớp có toàn bộ lý do tồn tại là để biểu diễn một hành vi (ví dụ: "squeaking"), và đó là lớp behavior, chứ không phải lớp Duck, sẽ triển khai giao diện behavior.

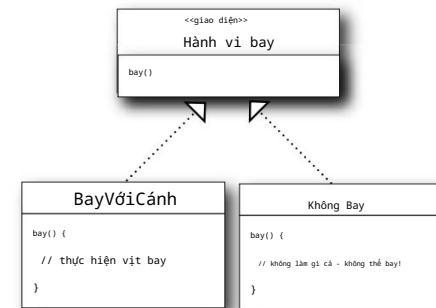
Điều này trái ngược với cách chúng ta làm trước đây, khi một hành vi xuất phát từ một triển khai cụ thể trong siêu lớp Duck hoặc bằng cách cung cấp một triển khai chuyên biệt trong chính lớp con. Trong cả hai trường hợp, chúng ta đều dựa vào một triển khai. Chúng ta bị khóa trong việc sử dụng triển khai cụ thể đó và không có chỗ để thay đổi hành vi (ngoài việc viết thêm mã).

Với thiết kế mới, các lớp con Duck sẽ sử dụng một hành vi được biểu diễn bởi một giao diện (FlyBehavior và QuackBehavior), do đó việc triển khai thực tế của hành vi (nói cách khác, hành vi cụ thể) được mã hóa trong lớp triển khai FlyBehavior hoặc QuackBehavior) sẽ không bị khóa trong lớp con Duck.

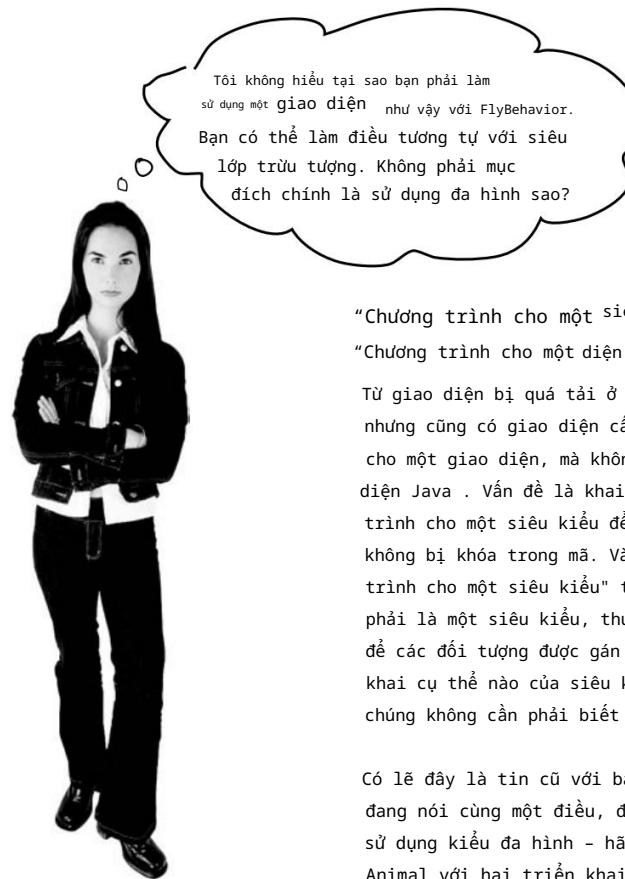
Từ bây giờ, các hành vi của Duck sẽ nằm

trong một lớp riêng biệt -
một lớp triển khai một
giao diện hành vi cụ thể.

Theo cách đó, các lớp Duck sẽ không cần biết bất kỳ chi tiết triển khai nào cho hành vi của riêng chúng.



chương trình đến một giao diện



"Chương trình cho một siêu kiểu giao diện" thực sự có nghĩa là "Chương trình cho một điện".

Từ giao diện bị quá tải ở đây. Có khái niệm về giao diện, nhưng cũng có giao diện cấu trúc Java. Bạn có thể lập trình cho một giao diện, mà không cần phải thực sự sử dụng giao diện Java. Vấn đề là khai thác tính đa hình bằng cách lập trình cho một siêu kiểu để đối tượng thời gian chạy thực tế không bị khóa trong mã. Và chúng ta có thể diễn đạt lại "lập trình cho một siêu kiểu" thành "kiểu đã khai báo của các biến phải là một siêu kiểu, thường là một lớp trừu tượng hoặc giao diện, để các đối tượng được gán cho các biến đó có thể là bất kỳ triển khai cụ thể nào của siêu kiểu, điều đó có nghĩa là lớp khai báo chúng không cần phải biết về các kiểu đối tượng thực tế!"

Có lẽ đây là tin cũ với bạn, nhưng để đảm bảo rằng chúng ta đang nói cùng một điều, đây là một ví dụ đơn giản về việc sử dụng kiểu đa hình - hãy tưởng tượng một lớp trừu tượng Animal với hai triển khai cụ thể là Dog và Cat.

Lập trình để triển khai sẽ là:

siêu kiểu trừu tượng (có thể
giảm thiểu lớp trừu tượng HOẶC

Chó d = Chó mới();
d. vỗ cây();

Việc khai báo biến "d" là kiểu Dog
(một triển khai cụ thể của Animal)
buộc chúng ta phải mã hóa theo
một triển khai cụ thể.

Nhưng lập trình cho một giao diện/siêu kiểu sẽ là:

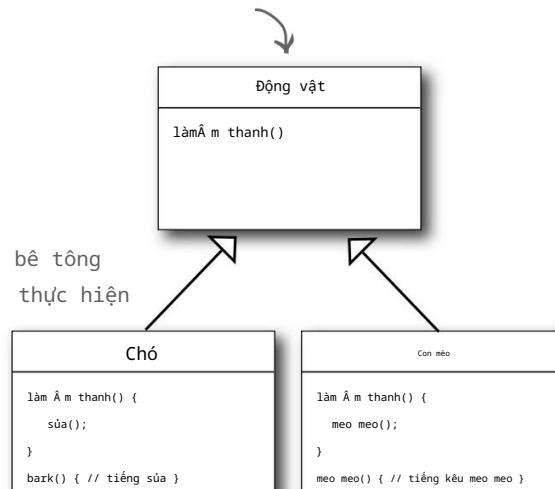
Động vật động vật = new Dog();
animal.makeSound();

Chúng ta biết đó là một con chó,
nhưng bây giờ chúng ta có thể sử dụng con vật
tham chiếu đa hình.

Tốt hơn nữa, thay vì mã hóa cứng việc khởi tạo kiểu con (như new Dog()) vào mã, hãy gán đối tượng triển khai cụ thể khi triển khai khi chạy:

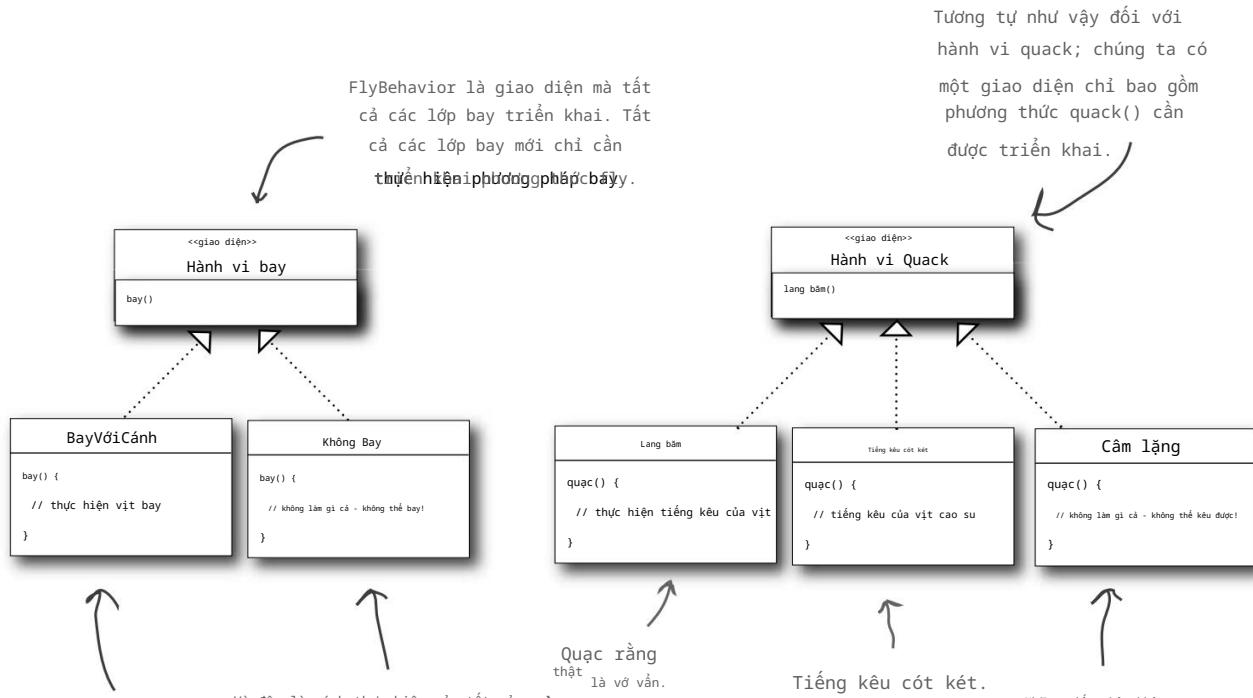
a = getAnimal();
a.makeSound();

Chúng ta không biết loại động vật thực
sự là gì... tất cả những gì chúng ta quan
tâm là nó biết cách phản ứng với
tạo âm thanh().



Thực hiện hành vi của con vịt

Ở đây chúng ta có hai giao diện, FlyBehavior và QuackBehavior cùng với các lớp tương ứng triển khai từng hành vi cụ thể:



Với thiết kế này, các loại đối tượng khác có thể tái sử dụng các hành vi bay và kêu của chúng ta vì các hành vi này không còn bị ẩn trong các lớp Duck nữa!

Và chúng ta có thể thêm các hành vi mới mà không cần sửa đổi bất kỳ lớp hành vi hiện có nào hoặc tác động đến bất kỳ lớp Vịt nào có hành vi bay.

sử dụng

Vì vậy, chúng ta có được lợi ích của việc TÁI SỬ DỤNG mà không phải chịu bất kỳ gánh nặng nào đi kèm với việc thừa kế.

hành vi trong một lớp học

không có Những câu hỏi ngắn

Q: Tôi có phải luôn luôn triển khai ứng dụng của mình trước không, hãy xem nơi mọi thứ đang thay đổi, sau đó quay lại và tách biệt & gói gọn những thứ đó?

A: Không phải lúc nào cũng vậy; thường thì khi bạn thiết kế một ứng dụng, bạn dự đoán những khu vực sẽ thay đổi và sau đó tiếp tục và xây dựng tính linh hoạt để xử lý chúng vào mã của bạn. Bạn sẽ thấy rằng các nguyên tắc và mẫu có thể được áp dụng ở bất kỳ giai đoạn nào của vòng đời phát triển.

H: Chúng ta có nên biến Duck thành một giao diện không?

A: Không phải trong trường hợp này. Như bạn sẽ thấy khi chúng ta có mọi thứ được nối với nhau, chúng ta được hưởng lợi khi Duck không phải là giao diện và có những con vịt cụ thể, như MallardDuck, thừa hưởng các thuộc tính và phương thức chung. Bây giờ chúng ta đã loại bỏ những gì thay đổi khỏi kế thừa Duck, chúng ta có được lợi ích của cấu trúc này mà không có vấn đề gì.

H: Cảm giác có chút kỳ lạ khi có một lớp học chỉ là hành vi. Các lớp không phải được cho là đại diện cho mọi thứ sao? Các lớp không phải được cho là có cả trạng thái và hành vi sao?

A: Trong hệ thống OO, đúng vậy, các lớp biểu diễn những thứ thường có cả trạng thái (biến thể hiện) và phương thức. Và trong trường hợp này, sự vật tình cờ là một hành vi. Nhưng ngay cả một hành vi vẫn có thể có trạng thái và phương thức; một hành vi bay có thể có các biến thể hiện đại diện cho các thuộc tính cho hành vi bay (cánh đậm mỗi phút, độ cao và tốc độ tối đa, v.v.).

 Chuốt bút chì của bạn

1 Sử dụng thiết kế mới của chúng tôi, bạn sẽ làm gì nếu cần thêm chức năng bay bằng tên lửa vào ứng dụng SimUDuck?

2 Bạn có thể nghĩ ra một lớp nào đó có thể muốn sử dụng hành vi Quack mà không phải là vịt không?

thiết bị phát ra tiếng

2) Một ví dụ, tiếng kêu

giao diện.

thực hiện FlyBehavior

Trả lời:

Tích hợp hành vi của vịt

Điều quan trọng là Duck sẽ thực hiện hành vi bay và kêu quack quack, thay vì sử dụng các phương pháp bay và kêu quack quack được định nghĩa trong lớp Duck (hoặc lớp con).

Sau đây là cách thực hiện:

- Đầu tiên chúng ta sẽ thêm hai biến thể hiện vào lớp Duck có tên là flyBehavior và quackBehavior, được khai báo là kiểu giao diện (không phải là kiểu triển khai lớp cụ thể). Mỗi đối tượng duck sẽ thiết lập các biến này theo hình để tham chiếu đến đối tượng cụ thể.

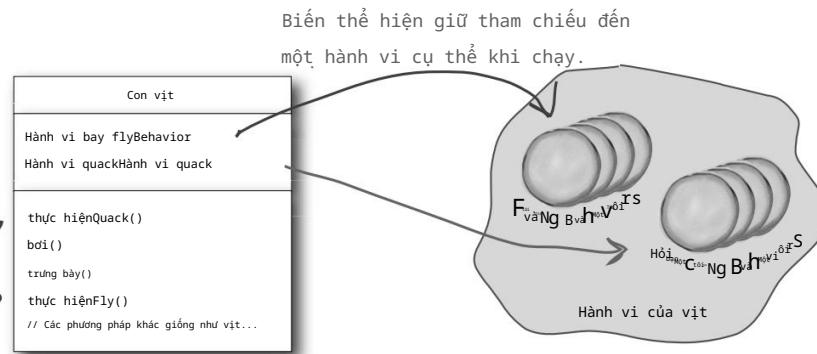
kiểu hành vi mà nó muốn khi chạy (FlyWithWings, Squeak, v.v.).

Chúng tôi cũng sẽ xóa các phương thức fly() và quack() khỏi lớp Duck (và bất kỳ lớp con nào) vì chúng tôi đã chuyển hành vi này ra các lớp FlyBehavior và QuackBehavior.

Chúng ta sẽ thay thế fly() và quack() trong lớp Duck bằng hai phương thức tương tự, được gọi là performFly() và performQuack(); bạn sẽ thấy cách chúng hoạt động ở phần tiếp theo.

Các biến hành vi được khai báo là kiểu GIAO ĐIỆN hành vi.

Các phương thức này thay thế fly() và quack().



- Bây giờ chúng ta triển khai performQuack():

```
lớp công khai Duck {
    Hành vi quack Hành vi quack;
    // hơn

    công khai void performQuack() {
        quackBehavior. quack();
    }
}
```

Mỗi Duck đều có tham chiếu đến thứ gì đó thực hiện giao diện QuackBehavior.

Thay vì xử lý hành vi quack, đối tượng Duck chuyển giao hành vi đó cho đối tượng được tham chiếu bởi Hành vi quack.

Khá đơn giản phải không? Để thực hiện tiếng kêu quack, Duck chỉ cần cho phép đối tượng được quackBehavior tham chiếu kêu quack.

Trong phần mã này, chúng ta không quan tâm đó là loại đối tượng gì, tất cả những gì chúng ta quan tâm là nó có biết cách kêu quack()! hay không.

tích hợp hành vi của vịt

Tích hợp nhiều hơn...

- 3 Được rồi, đến lúc phải lo lắng về cách flyBehavior và

Các biến thể hiện quackBehavior được thiết lập. Chúng ta hãy xem lớp MallardDuck:

```
lớp công khai MallardDuck mở rộng Duck {
```

```
    công khai MallardDuck() {
        quackBehavior = Quack mới();
        flyBehavior = new FlyWithWings();
    }
```

Hãy nhớ rằng, MallardDuck thừa hưởng các biến thể hiện quack-Behavior và flyBehavior từ lớp Duck.

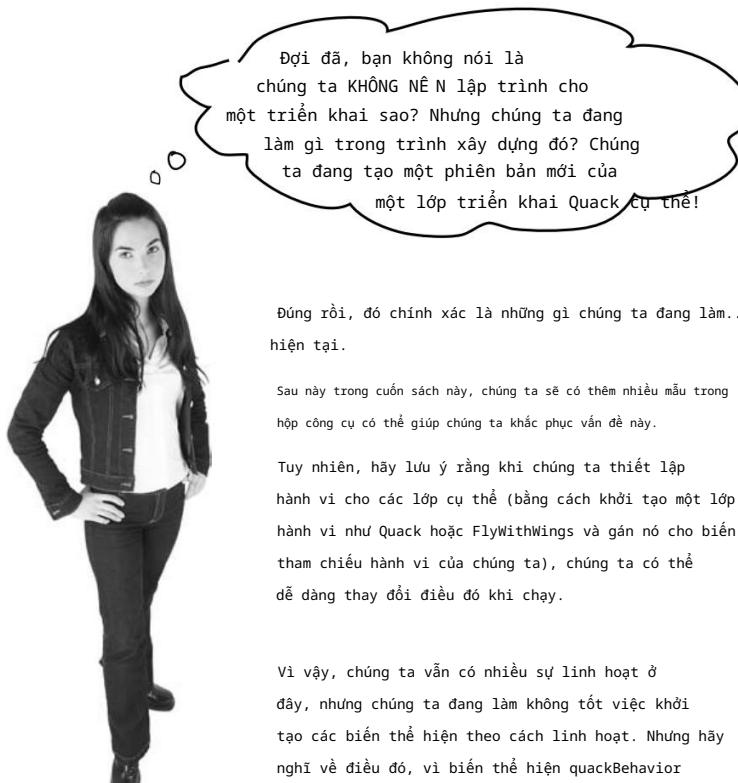
```
    công khai void display() {
        System.out.println("Tôi là một con vịt trời thực sự");
    }
}
```

MallardDuck sử dụng lớp Quack để xử lý quack của nó, vì vậy khi performQuack được gọi, trách nhiệm đổi với quack được chuyển giao cho đối tượng Quack và chúng ta có một gã lang băm thực sự.

Và nó sử dụng FlyWithWings làm kiểu FlyBehavior.

Vậy tiếng kêu của MallardDuck là tiếng kêu của vịt thật, không phải tiếng kêu cót két và không phải tiếng kêu câm. Vậy điều gì xảy ra ở đây? Khi MallardDuck được khởi tạo, hàm tạo của nó sẽ khởi tạo biến thể hiện quackBehavior được kế thừa của MallardDuck thành một thể hiện mới thuộc kiểu Quack (một lớp triển khai cụ thể của QuackBehavior).

Và điều tương tự cũng đúng với hành vi bay của vịt-hàm khởi tạo của MallardDuck khởi tạo biến thể hiện flyBehavior bằng một thể hiện thuộc kiểu FlyWithWings (một lớp triển khai cụ thể của FlyBehavior).



Hãy dành chút thời gian và suy nghĩ về cách bạn sẽ triển khai một con vịt để hành vi của nó có thể thay đổi khi chạy. (Bạn sẽ thấy mã thực hiện điều này ở một vài trang sau.)

kiểm tra hành vi của vịt

Kiểm tra mã Duck

- Nhập và biên dịch lớp Duck bên dưới (Duck.java) và lớp MallardDuck từ hai trang trước (MallardDuck.java).

```

lớp trừu tượng công khai Duck {
    FlyBehavior flyBehavior;
    Hành vi quack Hành vi quack;
    công khai Duck() {
    }

    công khai trừu tượng void display();

    công khai void performFly() {
        flyBehavior. fly();
    }

    công khai void performQuack() {
        quackBehavior. quack();
    }

    công khai void swim() {
        System.out.println("Tất cả vịt đều nổi, kể cả vịt mồi!");
    }
}

```

Annotations for Duck.java:

- A callout arrow points from the line "Khai báo hai biến tham chiếu cho các kiểu giao diện hành vi." to the declaration of `flyBehavior` and `quackBehavior`.
- A callout arrow points from the line "Tất cả các lớp con của duck (trong cùng một gói) đều thừa hưởng những điều này." to the entire class definition.
- A callout arrow points from the line "Ủy quyền cho lớp hành vi." to the line `flyBehavior. fly();`.

- Nhập và biên dịch giao diện FlyBehavior (FlyBehavior.java) và hai lớp triển khai hành vi (FlyWithWings.java và FlyNoWay.java).

```

giao diện công khai FlyBehavior {
    công khai void fly();
}

lớp công khai FlyWithWings triển khai FlyBehavior { public void fly() {
    System.out.println("Tôi đang bay!!");
}
}

lớp công khai FlyNoWay triển khai FlyBehavior {
    công khai void fly() {
        System.out.println("Tôi không thể bay");
    }
}

```

Annotations for FlyBehavior.java and its subclasses:

- A callout arrow points from the line "Giao diện mà tất cả các lớp hành vi bay đều triển khai." to the interface definition.
- A callout arrow points from the line "Thực hiện hành vi bay cho những con vịt CÓ THỂ bay..." to the implementation in FlyWithWings.
- A callout arrow points from the line "Thực hiện hành vi bay cho những con vịt KHÔNG biết bay (như vịt cao su và vịt mồi)." to the implementation in FlyNoWay.

Việc thử nghiệm mã Duck vẫn tiếp tục...

- 3 Nhập và biên dịch giao diện QuackBehavior (QuackBehavior.java) và ba lớp triển khai hành vi (Quack.java, MuteQuack.java và Squeak.java).

```
giao diện công khai QuackBehavior { public void
    quack();
}
```

```
lớp công khai Quack thực hiện QuackBehavior {
    công khai void quack()
    { System.out.println("Quack");
    }
}
```

```
lớp công khai MuteQuack triển khai QuackBehavior { public void quack()
    { System.out.println("<< Silence
        >>");
    }
}
```

```
lớp công khai Squeak triển khai QuackBehavior {
    public void quack()
    { System.out.println("Tiếng rít");
    }
}
```

- 4 Nhập và biên dịch lớp kiểm tra (MiniDuckSimulator.java).

```
lớp công khai MiniDuckSimulator {
    public static void main(String[] args) {
        Vịt trời = new MallardDuck();
        mallard.performQuack();
        mallard.performFly();
    }
}
```

Phương thức này gọi phương thức performQuack() được kế thừa từ MallardDuck, sau đó chuyển giao cho QuackBehavior của đối tượng (tức là gọi quack() trên tham chiếu quackBehavior được kế thừa từ duck).

Sau đó, chúng ta làm điều tương tự với phương thức performFly() kế thừa của MallardDuck.

- 5 Chạy mã!

```
Cửa sổ chỉnh sửa tệp Trợ giúp YadayaDayada
%java MiniDuckSimulator
Lang băm
Tôi đang bay!!
```

vịt có hành vi năng động

Thiết lập hành vi một cách năng động

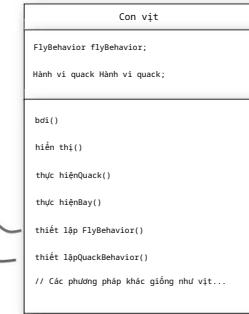
Thật đáng tiếc khi có tất cả tài năng năng động này được tích hợp vào các chú vịt của chúng ta và không sử dụng chúng! Hãy tưởng tượng bạn muốn thiết lập kiểu hành vi của chú vịt thông qua phương thức setter trên lớp con của chú vịt, thay vì tạo ra nó trong hàm tạo của chú vịt.

- Thêm hai phương thức mới vào lớp Duck:

```
công khai void setFlyBehavior(FlyBehavior fb) { flyBehavior =
    fb;
}

công khai void setQuackBehavior(QuackBehavior qb) {
    quackHành vi = qb;
}
```

Chúng ta có thể gọi những phương pháp này bất cứ lúc nào muốn thay đổi hành vi của một con vịt đang bay.



ghi chú của biên tập viên: chơi chữ miễn phí - fi x

- Tạo một loại Duck mới (ModelDuck.java).

```
lớp công khai ModelDuck mở rộng Duck { công khai
    ModelDuck() { flyBehavior
        = new FlyNoWay(); quackBehavior = new
        Quack();
    }

    public void display()
    { System.out.println("Tôi là một con vịt mô hình");
    }
}
```

Mô hình con vịt của chúng ta bắt đầu cuộc sống trên mặt đất... mà không có cách nào để bay.

- Tạo một kiểu FlyBehavior mới (*FlyRocketPowered.java*).

```
lớp công khai FlyRocketPowered triển khai FlyBehavior { public void fly() {
    System.out.println("Tôi đang bay bằng một tên lửa!");
}
}
```

Không sao cả, chúng ta đang tạo ra một hành vi bay có động cơ tên lửa.



- 4 Thay đổi lớp kiểm tra (MiniDuckSimulator.java), thêm ModelDuck và kích hoạt ModelDuck bằng rocket.

```
lớp công khai MiniDuckSimulator {
    public static void main(String[] args) {
        Vịt trời = new MallardDuck();
        mallard.performQuack();
        mallard.performFly();
```

```
    Mô hình vịt = new ModelDuck();
    model.performFly();
    model.setFlyBehavior(new FlyRocketPowered());
    model.performFly();
}
```

Nếu nó hoạt động, mô hình con vịt đã thay đổi hành vi bay của nó một cách năng động! Bạn không thể làm ĐIỀU ĐÓ nếu việc triển khai nằm bên trong lớp vịt.

- 5 Chạy đi!

```
Cửa sổ chỉnh sửa tệp Trợ giúp Yabababadoo
%java MiniDuckSimulator
Lang bäm
Tôi đang bay!!
Tôi không thể bay
Tôi đang bay với một tên lửa
```



trước

Lệnh gọi đầu tiên tới performFly() sẽ chuyển giao cho đối tượng flyBehavior được đặt trong hàm tạo của ModelDuck, đây là một thẻ hiện FlyNoWay.

Điều này sẽ kích hoạt phương thức thiết lập hành vi được thừa hưởng của mô hình và...vâng! Mô hình đột nhiên có khả năng bay như tên lửa!

Để thay đổi hành vi
của vịt khi chạy, chỉ cần
gọi phương thức thiết
lập của vịt cho hành vi đó.

bức tranh lớn

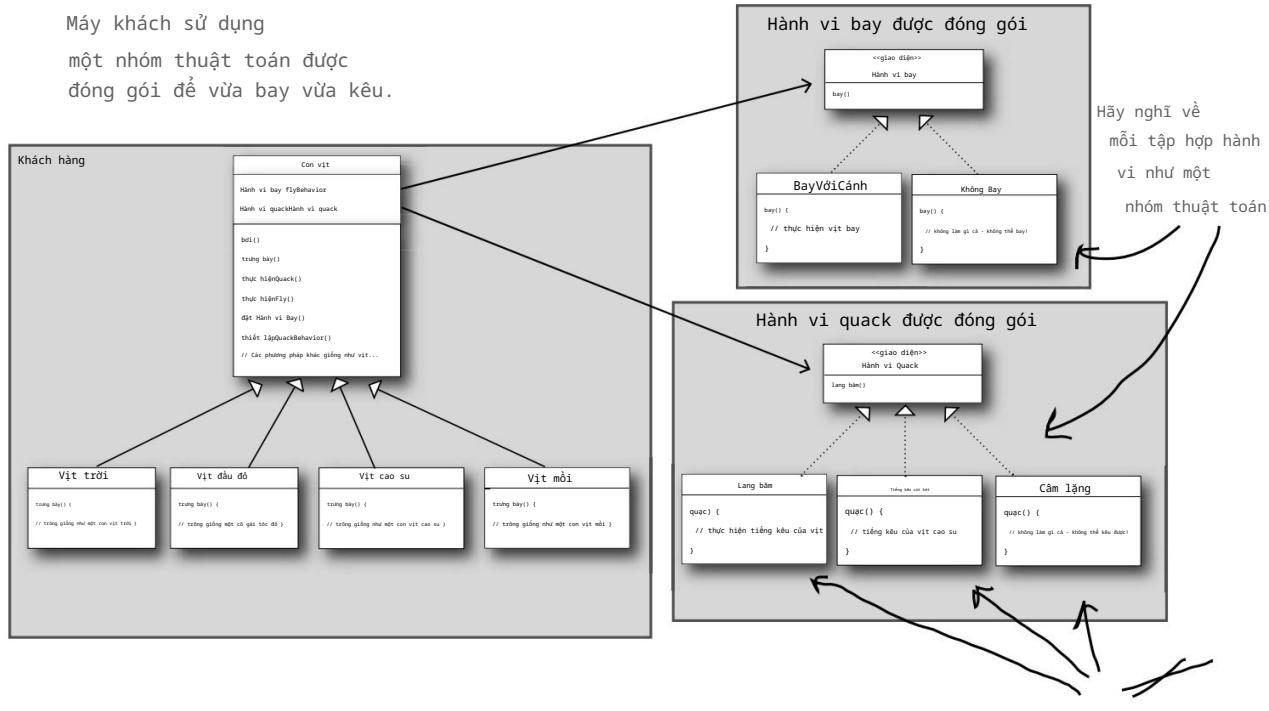
Bức tranh toàn cảnh về hành vi được đóng gói

Được rồi, bây giờ chúng ta đã tìm hiểu sâu về thiết kế mô phỏng vịt, đã đến lúc quay lại và nhìn vào bức tranh toàn cảnh.

Dưới đây là toàn bộ cấu trúc lớp được thiết kế lại. Chúng ta có mọi thứ bạn mong đợi: ducks mở rộng Duck, fly behavior triển khai FlyBehavior và quack behavior triển khai QuackBehavior.

Cũng lưu ý rằng chúng ta đã bắt đầu mô tả mọi thứ theo một cách hơi khác. Thay vì nghĩ về hành vi của vịt như một tập hợp các hành vi, chúng ta sẽ bắt đầu nghĩ về chúng như một họ các thuật toán. Hãy nghĩ về điều này: trong thiết kế SimUDuck, các thuật toán biểu diễn những việc mà một con vịt sẽ làm (các cách kêu hoặc bay khác nhau), nhưng chúng ta cũng có thể dễ dàng sử dụng các kỹ thuật tương tự cho một tập hợp các lớp thực hiện các cách tính thuế bán hàng của tiểu bang theo các tiểu bang khác nhau.

Hãy chú ý cẩn thận đến mối quan hệ giữa các lớp. Trên thực tế, hãy cầm bút và viết mối quan hệ phù hợp (IS-A, HAS-A và IMPLEMENTS) trên mỗi mũi tên trong sơ đồ lớp.



HAS-A có thể tốt hơn IS-A

Mỗi quan hệ HAS-A khá thú vị: mỗi con vịt có một FlyBehavior và một QuackBehavior mà nó chỉ định để bay và kêu.

Khi bạn ghép hai lớp lại với nhau như thế này, bạn đang sử dụng thành phần. Thay vì kế thừa hành vi của chúng, các chú vịt có được hành vi của chúng bằng cách được thành phần với đối tượng hành vi phù hợp.

Đây là một kỹ thuật quan trọng; trên thực tế, chúng tôi đã sử dụng nguyên tắc thiết kế thứ ba:



Nguyên lý thiết kế

Ưu tiên thành phần hơn là thừa kế.

Như bạn đã thấy, việc tạo hệ thống bằng cách sử dụng composition mang lại cho bạn nhiều sự linh hoạt hơn. Nó không chỉ cho phép bạn đóng gói một họ thuật toán vào tập hợp các lớp riêng của chúng mà còn cho phép bạn thay đổi hành vi khi chạy miễn là đối tượng bạn đang composition cùng triển khai giao diện hành vi chính xác.

Bộ cục được sử dụng trong nhiều mẫu thiết kế và bạn sẽ thấy nhiều hơn về ưu điểm cũng như nhược điểm của nó trong suốt cuốn sách.

não Apower

Tiếng kêu vịt là một thiết bị mà thợ săn sử dụng để bắt chước tiếng kêu (quacks) của vịt. Bạn sẽ triển khai tiếng kêu vịt của riêng mình như thế nào mà không kể thừa từ lớp Duck?



Thầy và trò...

Thầy: Cháu cháu, hãy cho thầy biết con đã học được những gì về phương pháp hướng đối tượng.

Học viên: Thưa thầy, con đã học được rằng lợi thế của phương pháp hướng đối tượng là khả năng tái sử dụng.

Sư phụ: Cháu cháu, tiếp tục đi...

Học viên: Thưa thầy, thông qua sự kế thừa, mọi thứ tốt đẹp đều có thể được tái sử dụng, vì vậy chúng ta sẽ cắt giảm đáng kể thời gian phát triển như cách chúng ta chặt tre trong rừng vậy.

Master: Grasshopper, thời gian dành cho code trước hay sau là nhiều hơn phát triển đã hoàn tất?

Học viên: Câu trả lời là sau, thưa thầy. Chúng tôi luôn dành nhiều thời gian để bảo trì và thay đổi phần mềm hơn là phát triển ban đầu.

Thầy: Vậy Grasshopper, liệu có nên tập trung vào khả năng tái sử dụng hơn là khả năng bảo trì và mở rộng không?

Học viên: Thưa thầy, con tin rằng điều này có sự thật.

Sư phụ: Ta thấy con còn nhiều điều phải học. Ta muốn con đi và thiền định thêm về thừa kế.

Như bạn đã thấy, tính kế thừa có những vấn đề riêng và vẫn còn nhiều cách khác để tái sử dụng.

mô hình chiến lược

Nói về Mẫu thiết kế...



Xin chúc mừng mẫu
đầu tiên của bạn!

Bạn vừa áp dụng mẫu thiết kế đầu tiên của mình- mẫu STRATEGY . Đúng vậy, bạn đã sử dụng Mẫu Strategy để làm lại ứng dụng SimUDuck.

Nhờ vào mô hình này, trình mô phỏng đã sẵn sàng cho mọi thay đổi mà các giám đốc điều hành có thể đưa ra trong chuyến công tác tiếp theo tới Vegas.

Bây giờ chúng tôi đã yêu cầu bạn thực hiện một chặng đường dài để áp dụng nó, sau đây là định nghĩa chính thức của mẫu này:

Mẫu chiến lược định nghĩa một họ các thuật toán, đóng gói từng thuật toán và làm cho chúng có thể hoán đổi cho nhau.
Chiến lược cho phép thuật toán thay đổi độc lập với các máy khách sử dụng nó.

Sử dụng định nghĩa NÀY khi bạn cần gây ấn tượng với bạn bè và gây ảnh hưởng đến các giám đốc điều hành quan trọng.



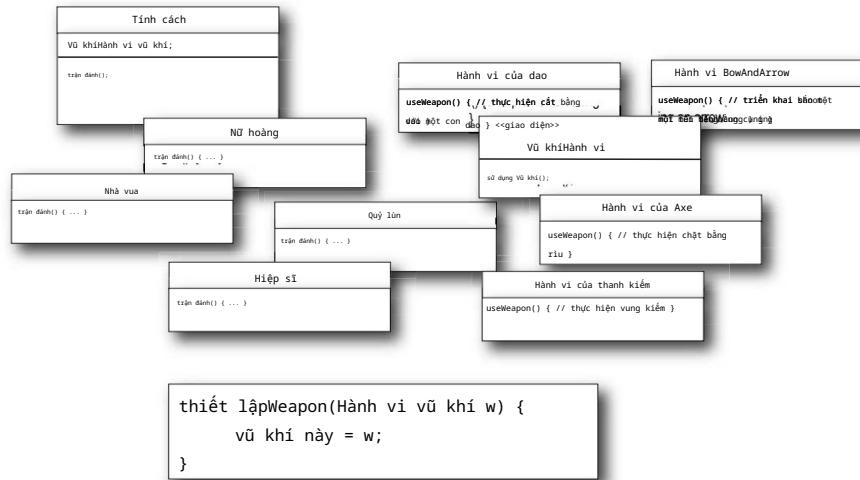
Thiết kế câu đố

Bên dưới bạn sẽ tìm thấy một mớ các lớp và giao diện cho một trò chơi phiêu lưu hành động. Bạn sẽ tìm thấy các lớp cho các nhân vật trong trò chơi cùng với các lớp cho hành vi vũ khí mà các nhân vật có thể sử dụng trong trò chơi. Mỗi nhân vật có thể sử dụng một vũ khí tại một thời điểm, nhưng có thể thay đổi vũ khí bất cứ lúc nào trong trò chơi. Nhiệm vụ của bạn là sắp xếp tất cả...

(Câu trả lời có ở cuối chương.)

Nhiệm vụ của bạn:

- 1 Sắp xếp các lớp học.
- 2 Xác định một lớp trừu tượng, một giao diện và tám lớp.
- 3 Vẽ mũi tên giữa các lớp.
 - a. Vẽ loại mũi tên này để kế thừa ("mở rộng"). →
 - b. Vẽ loại mũi tên này cho giao diện ("thực hiện").→
 - c. Vẽ loại mũi tên này cho "HAS-A". →
- 4 Đặt phương thức setWeapon() vào đúng lớp.



Nghe lén tại quán ăn địa phương...

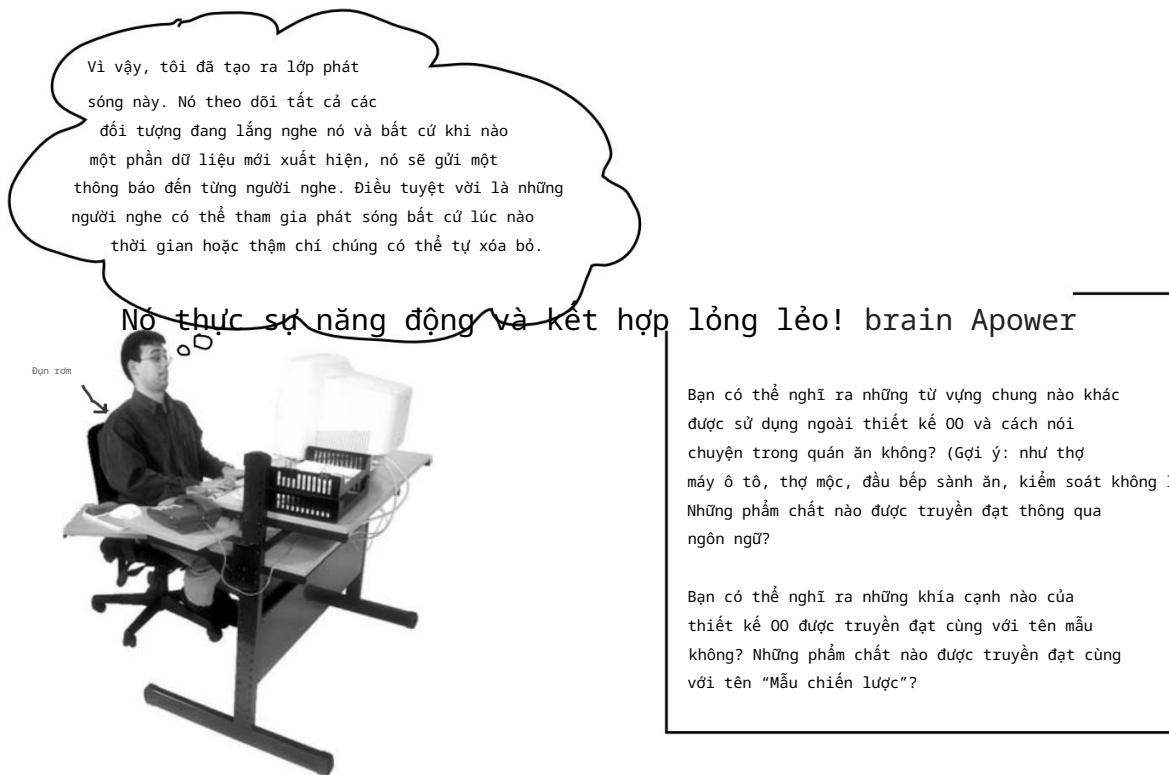


Sự khác biệt giữa hai đơn hàng này là gì? Không có gì cả! Cả hai đều là cùng một đơn hàng, ngoại trừ việc Alice sử dụng gấp đôi số từ và thử thách sự kiên nhẫn của một đầu bếp phục vụ đồ ăn nhanh khó tính.

Flo có gì mà Alice không có? Một vốn từ vựng chung với đầu bếp phục vụ đồ ăn nhanh. Không chỉ dễ giao tiếp hơn với đầu bếp mà còn giúp đầu bếp nhớ ít hơn vì anh ta có tất cả các mẫu thực đơn trong đầu.

Design Patterns cung cấp cho bạn vốn từ vựng chung với các nhà phát triển khác. Khi bạn có vốn từ vựng, bạn có thể dễ dàng giao tiếp với các nhà phát triển khác và truyền cảm hứng cho những người không biết về các mẫu để bắt đầu học chúng. Nó cũng nâng cao tư duy của bạn về kiến trúc bằng cách cho phép bạn suy nghĩ ở cấp độ mẫu, không phải cấp độ đối tượng cụ thể.

Nghe lén ở phòng bên cạnh...



từ vựng chung

Sức mạnh của một vốn từ vựng mẫu chung

Khi bạn giao tiếp bằng cách sử dụng các mẫu câu, bạn không chỉ đơn thuần chia sẻ LINGO.

Các từ vựng theo mẫu chung rất MẠNH MẼ.

Khi bạn giao tiếp với nhà phát triển khác hoặc nhóm của mình bằng cách sử dụng các mẫu, bạn không chỉ truyền đạt tên mẫu mà còn là toàn bộ các phẩm chất, đặc điểm và ràng buộc mà mẫu đó đại diện.

Các mẫu cho phép bạn nói nhiều hơn với ít hơn. Khi bạn sử dụng một mẫu trong mô tả, các nhà phát triển khác sẽ nhanh chóng biết chính xác thiết kế mà bạn đang nghĩ đến.

Nói chuyện ở cấp độ mẫu cho phép bạn ở lại "trong thiết kế" lâu hơn. Nói về các hệ thống phần mềm sử dụng mẫu cho phép bạn giữ cuộc thảo luận ở cấp độ thiết kế, mà không cần phải đi sâu vào các chi tiết cụ thể của việc triển khai các đối tượng và lớp.

Từ vựng chung có thể thúc đẩy nhóm phát triển của bạn. Một nhóm thành thạo các mẫu thiết kế có thể di chuyển nhanh hơn với ít chỗ cho sự hiểu lầm hơn.

Các từ vựng chung khuyến khích nhiều nhà phát triển mới vào nghề hơn để bắt kịp tiến độ. Các nhà phát triển mới vào nghề trộn cậy vào các nhà phát triển có kinh nghiệm. Khi các nhà phát triển cao cấp sử dụng các mẫu thiết kế, các nhà phát triển mới vào nghề cũng có động lực để học chúng. Xây dựng một cộng đồng người dùng mẫu tại tổ chức của bạn.

"Chúng tôi đang sử dụng mô hình chiến lược để thực hiện nhiều hành vi khác nhau của đòn vịt." Điều này cho bạn biết hành vi của vịt đã được đóng gói vào một tập hợp các lớp riêng có thể dễ dàng mở rộng và thay đổi, ngay cả khi chạy nếu cần.

Bạn đã tham gia bao nhiêu cuộc họp thiết kế và nhanh chóng chuyển sang giai đoạn triển khai chi tiết?

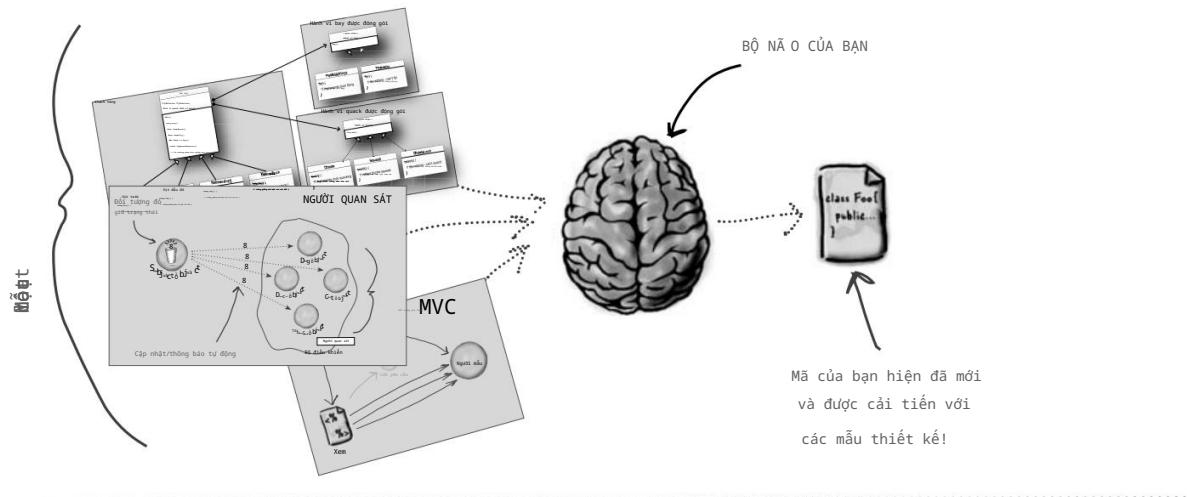
Khi nhóm của bạn bắt đầu chia sẻ ý tưởng thiết kế và kinh nghiệm về mặt mẫu mã, bạn sẽ xây dựng được một cộng đồng của người dùng mẫu.

Hãy nghĩ đến việc thành lập một nhóm nghiên cứu mẫu tại tổ chức của bạn, có thể bạn sẽ được trả lương trong khi học... ;)

Tôi sử dụng Mẫu thiết kế như thế nào?

Chúng ta đều đã sử dụng các thư viện và khung có sẵn. Chúng ta lấy chúng, viết một số mã cho API của chúng, biên dịch chúng thành các chương trình của chúng ta và hưởng lợi từ rất nhiều mã mà người khác đã viết. Hãy nghĩ về các API Java và tất cả các chức năng mà chúng cung cấp cho bạn: mạng, GUI, IO, v.v. Các thư viện và khung đi kèm thường dài hướng tới một mô hình phát triển mà chúng ta có thể chỉ cần chọn và chọn các thành phần và cắm chúng ngay vào. Nhưng... chúng không giúp chúng ta cấu trúc các ứng dụng của riêng mình theo những cách dễ hiểu hơn, dễ bảo trì hơn và linh hoạt hơn. Đó là nơi các Mẫu thiết kế xuất hiện.

Các mẫu thiết kế không đi thẳng vào mã của bạn, trước tiên chúng đi vào não bộ của bạn. Khi bạn đã nạp vào não bộ kiến thức làm việc tốt về các mẫu, sau đó bạn có thể bắt đầu áp dụng chúng vào các thiết kế mới của mình và làm lại mã cũ khi bạn thấy nó đang suy thoái thành một mớ hỗn độn không linh hoạt của mã spaghetti.



không có Những câu hỏi ngớ ngẩn

H: Nếu các mẫu thiết kế tuyệt vời như vậy,

Tại sao không ai xây dựng một thư viện về chúng để tôi không phải làm vậy?

MỘT:

Mẫu thiết kế ở cấp độ cao hơn thư viện. Mẫu thiết kế cho chúng ta biết cách cấu trúc các lớp và đối tượng để giải quyết các vấn đề nhất định và nhiệm vụ của chúng ta là điều chỉnh các thiết kế đó cho phù hợp với ứng dụng cụ thể của mình.

Q: Thư viện và khung không phải là

cũng thiết kế mẫu?

A: Các khuôn khổ và thư viện không phải là

mẫu thiết kế; chúng cung cấp các triển khai cụ thể mà chúng ta liên kết vào mã của mình. Tuy nhiên, đôi khi, các thư viện và khung sử dụng các mẫu thiết kế trong triển khai của chúng. Điều đó thật tuyệt, vì khi bạn hiểu các mẫu thiết kế, bạn sẽ nhanh hơn

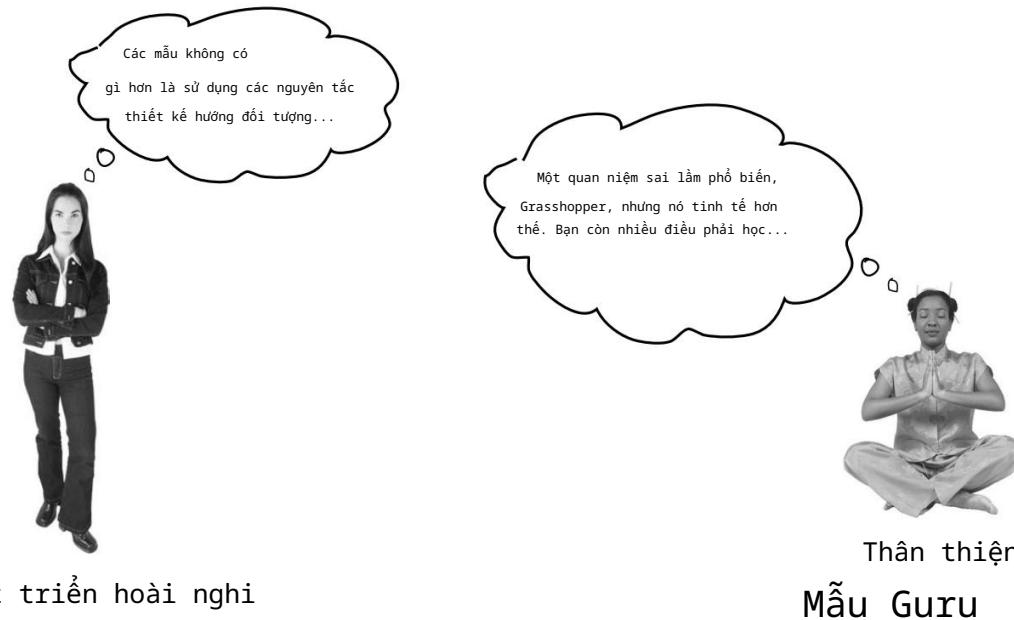
hiểu các API có cấu trúc xung quanh các mẫu thiết kế.

Q: Vậy thì không có thư viện nào cả mẫu thiết kế?

MỘT:

Không, nhưng sau này bạn sẽ tìm hiểu về danh mục mẫu có danh sách các mẫu mà bạn có thể áp dụng cho ứng dụng của mình.

tại sao phải thiết kế mẫu?



Nhà phát triển: Được thôi, hmm, nhưng không phải tất cả những điều này chỉ là thiết kế hướng đối tượng tốt sao; Ý tôi là miễn là tôi tuân theo đóng gói và tôi biết về trừu tượng hóa, kế thừa và đa hình, thì tôi có thực sự cần phải nghĩ về Mẫu thiết kế không? Không phải là khá đơn giản sao? Không phải đây là lý do tại sao tôi học tất cả các khóa học OO đó sao? Tôi nghĩ Mẫu thiết kế hữu ích cho những người không biết thiết kế OO tốt.

Thầy: À, đây là một trong những hiểu lầm thực sự về phát triển hướng đối tượng: rằng bằng cách biết những điều cơ bản về hướng đối tượng, chúng ta sẽ tự động giỏi trong việc xây dựng các hệ thống linh hoạt, có thể tái sử dụng và bảo trì được.

Nhà phát triển: Không à?

Thầy: Không. Thực ra, việc xây dựng các hệ thống hướng đối tượng có những đặc tính này không phải lúc nào cũng dễ dàng và chỉ được khám phá thông qua quá trình làm việc chăm chỉ.

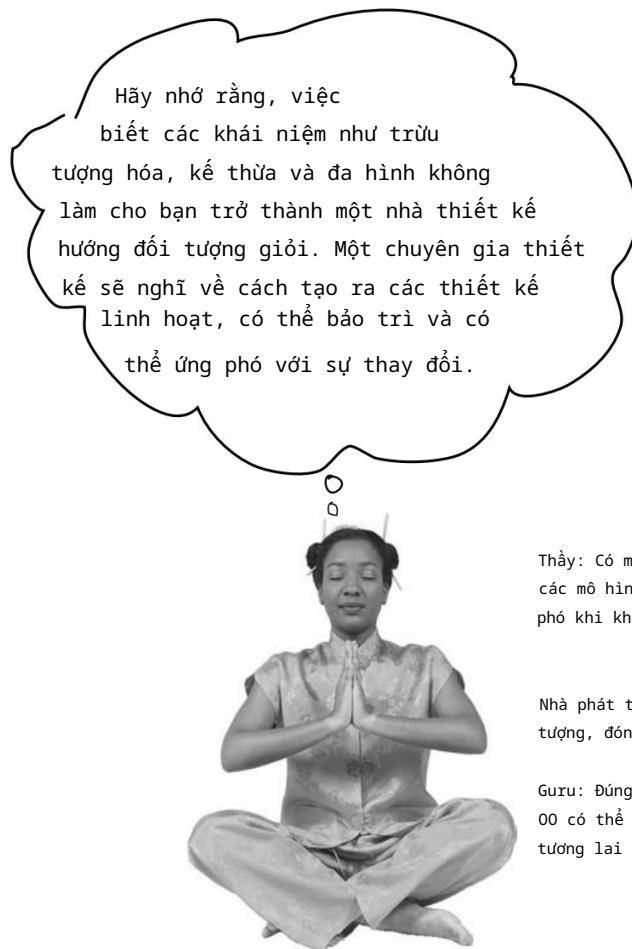
Nhà phát triển: Tôi nghĩ là tôi đã bắt đầu hiểu rồi. Những cách xây dựng hệ thống hướng đối tượng này, đôi khi không rõ ràng, đã được thu thập...

Thầy: ...đúng vậy, thành một tập hợp các mẫu gọi là Mẫu thiết kế.

Nhà phát triển: Vậy, bằng cách biết các mẫu, tôi có thể bỏ qua công việc khó khăn và chuyển thẳng đến những thiết kế luôn hiệu quả?

Guru: Đúng vậy, ở một mức độ nào đó, nhưng hãy nhớ rằng, thiết kế là một nghệ thuật. Sẽ luôn có sự đánh đổi. Nhưng nếu bạn tuân theo các mẫu thiết kế được cân nhắc kỹ lưỡng và thử nghiệm theo thời gian, bạn sẽ tiến xa hơn nhiều.

Nhà phát triển: Tôi phải làm gì nếu không tìm thấy mẫu?

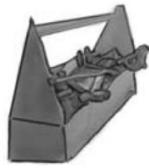


Thầy: Có một số nguyên tắc hướng đối tượng làm nền tảng cho các mô hình và việc biết những nguyên tắc này sẽ giúp bạn ứng phó khi không tìm được mô hình phù hợp với vấn đề của mình.

Nhà phát triển: Nguyên tắc? Ý bạn là vượt ra ngoài sự trừu tượng, đóng gói và...

Guru: Đúng vậy, một trong những bí quyết để tạo ra các hệ thống OO có thể bảo trì là suy nghĩ về cách chúng có thể thay đổi trong tương lai và các nguyên tắc này giải quyết những vấn đề đó.

hộp công cụ thiết kế của bạn



Công cụ cho hộp công cụ thiết kế của bạn

Bạn đã gần hoàn thành chương đầu tiên rồi! Bạn đã đưa một số công cụ vào hộp công cụ 00 của mình; hãy lập danh sách trước khi chuyển sang Chương 2.

Cơ bản về 00

- Trừu tượng
- Đóng gói
- Đa hình
- Heritage*

Nguyên tắc 00

Bao gồm những gì thay đổi.

Ưu tiên thành phần hơn
di truyền.

Chương trình hướng tới giao diện,
không phải triển khai.

Mẫu 00

Chiến lược - định nghĩa một họ các thuật toán,
đóng gói từng thuật toán và làm cho chúng
có thể hoán đổi cho nhau. Chiến lược cho phép
thuật toán thay đổi độc lập với các máy khách sử dụng nó.

Đã xong một, còn nhiều nữa!

Chúng tôi cho rằng bạn biết những điều cơ bản về 00 khi sử dụng các lớp đa hình, cách kế thừa giống như thiết kế theo hợp đồng và cách đóng gói hoạt động. Nếu bạn hơi kém về những điều này, hãy lấy Head First Java của bạn ra và xem lại, sau đó lướt qua phần này chương nữa.

Chúng tôi sẽ xem xét kỹ hơn những điều này trong tương lai và cũng sẽ thêm một vài điều nữa vào danh sách

Trong suốt cuốn sách, hãy suy nghĩ về cách các mô hình dựa trên các nguyên tắc và cơ sở 00.

ĐIỂM ĐẦU TIÊN

B Biết những điều cơ bản về 00

Không làm cho bạn trở thành một nhà thiết kế hướng đối tượng giỏi.

B Thiết kế 00 tốt là có thể tái sử dụng, mở rộng và bảo trì.

B Các mẫu cho bạn biết cách xây dựng các hệ thống có chất lượng thiết kế hướng đối tượng tốt.

B Các mẫu là kinh nghiệm hướng đối tượng đã được chứng minh.

B Các mẫu không cung cấp cho bạn mã, chúng cung cấp cho bạn các giải pháp chung cho các vấn đề thiết kế. Bạn áp dụng chúng vào ứng dụng cụ thể của mình.

B Các mẫu hình không phải được phát minh mà được khám phá.

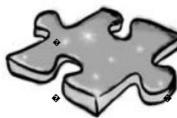
Hầu hết các mẫu và nguyên tắc đều giải quyết các vấn đề về thay đổi trong phần mềm.

Hầu hết các mẫu cho phép một số bộ phận của hệ thống thay đổi độc lập với các bộ phận khác.

Chúng ta thường cố gắng lấy những thay đổi trong một hệ thống và gói gọn nó lại.

B Các mẫu cung cấp một ngôn ngữ chung có thể tối đa hóa giá trị giao tiếp của bạn với các nhà phát triển khác.

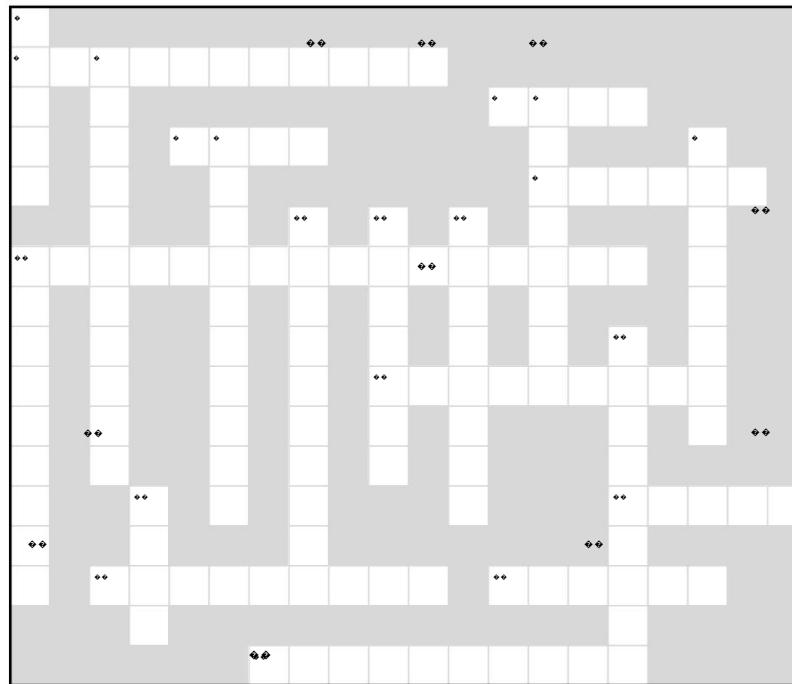
giới thiệu về Mẫu thiết kế



Hãy cho não phải của bạn một việc gì đó để làm.

Đây là trò chơi ô chữ tiêu chuẩn; tất cả các từ giải đều có trong chương này.

♦ ♦ ♦ ♦ ♦

**Sang**

2. _____ những gì thay đổi
4. Mẫu thiết kế _____
6. ~~Mẫu thiết kế~~ Âm thanh
9. Vịt cao su làm một
4. Mẫu thiết kế với pha chế nghĩ rằng họ được gọi
6. Java IO, ~~Mẫu~~ không phải là một triển khai
9. Vịt cao su làm một và _____ của bạn
18. Học hỏi từ người khác
13. Người pha chế nghĩ rằng họ được gọi
15. Chương trình này, không phải là một triển khai
20. Các mẫu _____
17. Các mẫu đi vào _____ của bạn
18. Học hỏi từ người khác
19. Hàng số phát triển
20. Các mẫu cho chúng ta một sự chia sẻ _____

Xuống

1. Các mẫu _____ trong nhiều ứng dụng
3. Ưu tiên hơn kẽ
5. Danh sách _____
7. Hầu hết các mẫu đều theo _____ trong nhiều ứng dụng
8. Không phải _____
10. Thủ công _____
11. Đồ uống yêu thích của Joe
13. Con vịt không biết _____
14. Phô mai nướng với thịt xông khói
16. Bản demo vịt _____
11. Đồ uống yêu thích của Joe
12. Mẫu cố định trình mô phỏng
13. Con vịt không biết _____
14. Phô mai nướng với thịt xông khói
16. Bản demo vịt được đặt ở đâu

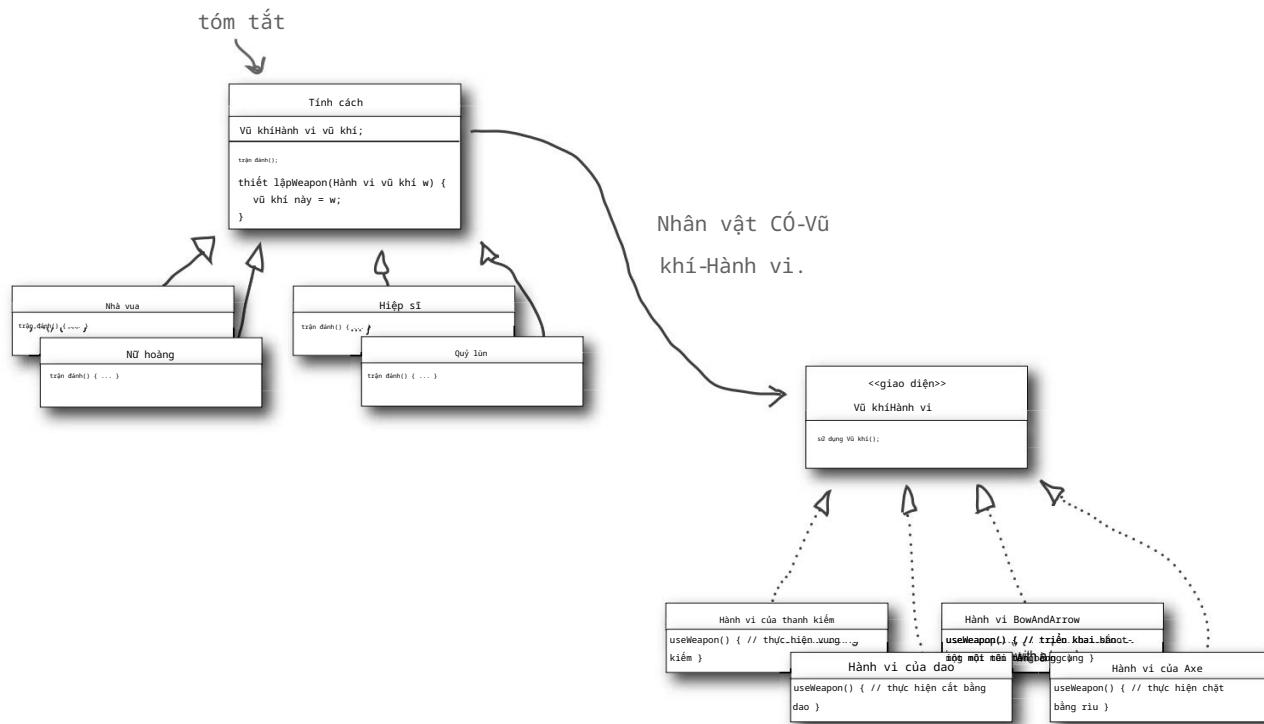
giải pháp thiết kế câu đố



Giải pháp thiết kế câu đố

Character là lớp trừu tượng cho tất cả các nhân vật khác (King, Queen, Knight và Troll) trong khi Weapon là giao diện mà tất cả vũ khí đều triển khai. Vì vậy, tất cả các nhân vật và vũ khí thực tế đều là các lớp cụ thể.

Để chuyển đổi vũ khí, mỗi nhân vật gọi phương thức setWeapon(), được định nghĩa trong siêu lớp Character. Trong một trận chiến, phương thức useWeapon() được gọi trên bộ vũ khí hiện tại của một nhân vật nhất định để gây sát thương lớn lên một nhân vật khác.



Lưu ý rằng BẤT KỲ đối tượng nào cũng có thể triển khai giao diện WeaponBehavior. Ví dụ, một chiếc kẹp giấy, một ống kem đánh răng hoặc một cá mú đột biến.

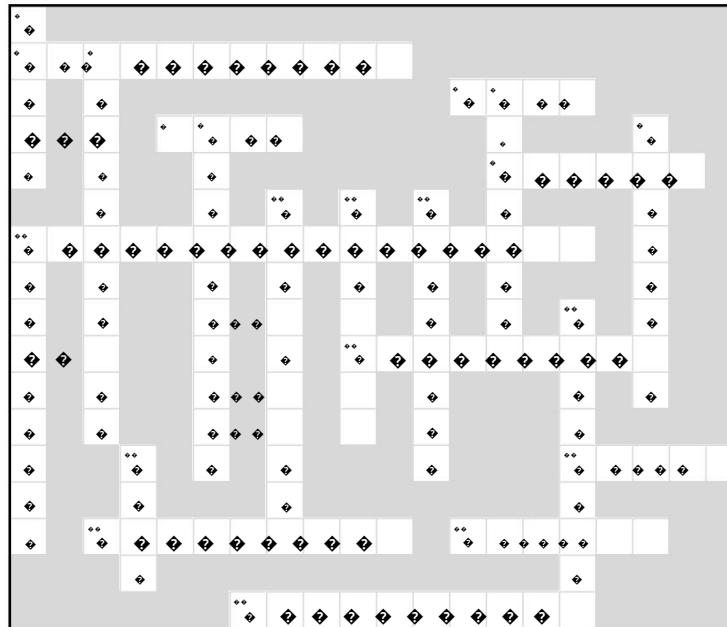
Giải pháp

Chuột bút chì của bạn



Trong các nhược điểm sau đây, nhược điểm nào là của việc sử dụng phân lớp để cung cấp hành vi cụ thể của Duck? (Chọn tất cả các đáp án phù hợp.)

A. Mã được sao chép giữa các lớp con.
 B. Thay đổi hành vi thời gian chạy rất khó khăn.
 C. Chúng ta không thể làm điều nhảy của vịt.
 D. Vịt không thể vừa bay vừa kêu cùng một lúc.
 E. Những thay đổi có thể vô tình ảnh hưởng đến những con vịt khác.



Chuột bút chì của bạn



Một số yếu tố nào thúc đẩy tôi thay đổi cung cấp ứng dụng của bạn? Bạn có thể có danh sách rất khác, nhưng đây là một vài danh sách của chúng tôi. Bạn có thấy quen không?

Khách hàng hoặc người dùng của tôi quyết định họ muốn thử gì đó khác hoặc muốn chức năng mới.

Công ty tôi quyết định sẽ hợp tác với một nhà cung cấp cơ sở dữ liệu khác và cũng mua dữ liệu từ một nhà cung cấp khác, sử dụng định dạng dữ liệu khác... Argh...

Vâng... trong ngày thay đổi và chúng ta phải cập nhật nó của mình để sử dụng các giao thức.

Chúng tôi đã học được nhiều điều khi xây dựng hệ thống và muốn quay lại và làm mọi việc tốt hơn một chút.

2 Mẫu quan sát viên

g Giữ gìn của bạn
h Các đối tượng trong biết
g



Này Jerry, tôi
thông báo với mọi người rằng
cuộc họp của Nhóm Patterns đã
chuyển sang tối thứ Bảy. Chúng ta sẽ
nói về Observer Pattern.
Mẫu đó là tốt nhất! Nó là TỐT NHẤT,
Jerry!

Đừng bỏ lỡ khi có điều gì đó thú vị xảy ra! Chúng tôi có một
mẫu giúp các đối tượng của bạn biết khi có điều gì đó chúng có thể quan tâm xảy ra.
Các đối tượng thậm chí có thể quyết định tại thời điểm chạy xem chúng có muốn được thông báo hay
không. Observer Pattern là một trong những mẫu được sử dụng nhiều nhất trong JDK và nó cực kỳ hữu ích.
Trước khi kết thúc, chúng ta cũng sẽ xem xét các mối quan hệ một-nhiều và liên kết lồng lèo (vâng, đúng
vậy, chúng ta đã nói liên kết). Với Observer, bạn sẽ là linh hồn của Patterns Party.

trạm theo dõi thời tiết

Chúc mừng!

Nhóm của bạn vừa trúng thầu xây dựng Trạm giám sát thời tiết thế hệ tiếp theo trên nền tảng Internet của Weather-O-Rama, Inc.



Công ty TNHH Weather-O-Rama
100 Phố Chính
Hẻm Tornado, OK 45021

Tuyên bố công việc

Xin chúc mừng bạn đã được chọn xây dựng Trạm giám sát thời tiết trực tuyến thế hệ tiếp theo của chúng tôi!

Trạm thời tiết sẽ dựa trên đối tượng WeatherData đang chờ cấp bằng sáng chế của chúng tôi, đối tượng này theo dõi các điều kiện thời tiết hiện tại (nhiệt độ, độ ẩm và áp suất khí quyển). Chúng tôi muốn bạn tạo một ứng dụng ban đầu cung cấp ba thành phần hiển thị: điều kiện hiện tại, số liệu thống kê thời tiết và dự báo đơn giản, tất cả đều được cập nhật theo thời gian thực khi đối tượng WeatherData thu thập các phép đo gần đây nhất.

Hơn nữa, đây là một trạm thời tiết có thể mở rộng. Weather-O-Rama muốn phát hành một API để các nhà phát triển khác có thể viết màn hình thời tiết của riêng họ và cắm chúng vào ngay. Chúng tôi Weather-O-thích bạn cung cấp API đó!

Rama nghĩ rằng chúng tôi có một mô hình kinh doanh tuyệt vời: một khi khách hàng đã bị thu hút, chúng tôi dự định tính phí họ cho mỗi màn hình họ sử dụng. Đây giờ là phần tốt nhất: chúng tôi sẽ trả tiền bạn có quyền chọn cổ phiếu.

Chúng tôi mong muốn được chứng kiến thiết kế và ứng dụng alpha của bạn.

Trân trọng,

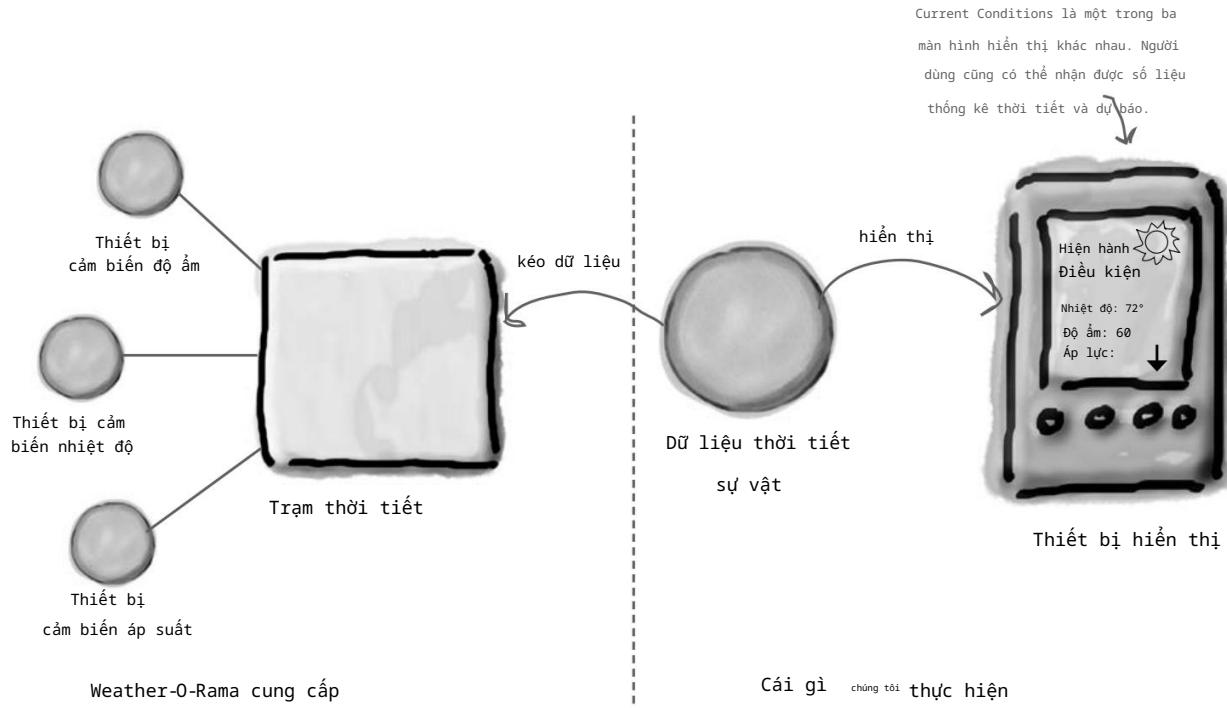
Johnny Hurricane
Johnny Hurricane

Johnny Hurricane, Tổng giám đốc điều hành

PS Chúng tôi sẽ gửi qua đêm các tập tin nguồn WeatherData cho bạn.

Tổng quan về ứng dụng Theo dõi thời tiết

Ba thành phần trong hệ thống là trạm thời tiết (thiết bị vật lý thu thập dữ liệu thời tiết thực tế), đối tượng WeatherData (theo dõi dữ liệu từ Trạm thời tiết và cập nhật màn hình) và màn hình hiển thị cho người dùng tình hình thời tiết hiện tại.



Đối tượng WeatherData biết cách giao tiếp với Trạm thời tiết vật lý để lấy dữ liệu cập nhật. Sau đó, đối tượng WeatherData cập nhật màn hình hiển thị của nó cho ba thành phần hiển thị khác nhau: Điều kiện hiện tại (hiển thị nhiệt độ, độ ẩm và áp suất), Thống kê thời tiết và dự báo đơn giản.

Công việc của chúng ta, nếu chúng ta chọn chấp nhận, là tạo một ứng dụng sử dụng đối tượng WeatherData để cập nhật ba màn hình hiển thị cho điều kiện hiện tại, số liệu thống kê thời tiết và dự báo.

lớp dữ liệu thời tiết

Giải nén lớp WeatherData

Như đã hứa, sáng hôm sau các tập tin nguồn WeatherData sẽ đến.

Nhìn vào bên trong mã, mọi thứ có vẻ khá đơn giản:

```

Dữ liệu thời tiết
lấy Nhiệt Độ()
lấy Độ ẩm()
lấy áp lực()
phép đoĐã thay đổi()
// các phương pháp khác

```

Ba phương pháp này sẽ trả về các số liệu đo thời tiết gần đây nhất về nhiệt độ, độ ẩm và áp suất khí quyển.

Chúng tôi không quan tâm CÁCH các biến này được thiết lập; đối tượng WeatherData biết cách lấy thông tin cập nhật từ Trạm thời tiết.

Các nhà phát triển đối tượng WeatherData đã để lại cho chúng tôi một manh mối về những gì chúng tôi cần thêm...

```

/*
 * Phương pháp này được gọi * bất
 * cứ khi nào các phép đo thời tiết
 * đã được cập nhật
 *
*/
công khai void measurementsChanged() {
    // Mã của bạn ở đây
}

```

WeatherData.java

Hãy nhớ rằng, Điều kiện hiện tại này chỉ là MỘT trong ba màn hình hiển thị khác nhau.



Thiết bị hiển thị

Công việc của chúng ta là triển khai measurementsChanged() để cập nhật ba màn hình hiển thị cho điều kiện hiện tại, số liệu thống kê thời tiết và dự báo.

mẫu quan sát

Cho đến nay chúng ta biết những gì?



Thông số kỹ thuật từ Weather-O-Rama không rõ ràng lắm, nhưng chúng ta phải tìm ra những gì cần làm. Vậy, cho đến nay chúng ta đã biết những gì?

Lớp WeatherData có các phương thức lấy dữ liệu cho ba giá trị đo lường: nhiệt độ, độ ẩm và áp suất khí quyển.

lấy Nhiệt Độ()

lấy Độ Ẩm()

lấy Áp Suất()

Phương thức measurementsChanged() được gọi bất kỳ thời gian có dữ liệu đo thời tiết mới. (Chúng tôi không biết hoặc không quan tâm phương pháp này được gọi là gì; chúng tôi chỉ biết rằng nó là như vậy.)

phép đođã thay đổi()

R Chúng ta cần triển khai ba phần tử hiển thị sử dụng dữ liệu thời tiết: màn hình hiển thị điều kiện hiện tại, màn hình hiển thị thống kê và màn hình hiển thị dự báo. Các màn hình này phải được cập nhật mỗi khi WeatherData có phép đo mới.



R Hệ thống phải có khả năng mở rộng—các nhà phát triển khác có thể tạo các thành phần hiển thị tùy chỉnh mới và người dùng có thể thêm hoặc xóa bao nhiêu thành phần hiển thị tùy ý vào ứng dụng. Hiện tại, chúng tôi chỉ biết về ba loại hiển thị ban đầu (điều kiện hiện tại, thống kê và dự báo).



Màn hình tương lai

lần thử đầu tiên với trạm thời tiết

Lần đầu tiên nhận được SWAG nhầm lẫn tại Trạm thời tiết

Đây là khả năng triển khai đầu tiên-chúng ta sẽ lấy gợi ý từ Weather-O-
Các nhà phát triển Rama và thêm mã của chúng tôi vào phương thức measurementsChanged():

```
lớp công khai WeatherData {  
  
    // khai báo biến thể hiện  
  
    công khai void measurementsChanged() {  
  
        float temp = getTemperature();  
        độ ẩm nổi = getHumidity();  
        áp suất nổi = getPressure();  
  
        currentConditionsDisplay.update(nhiệt độ, độ ẩm, áp suất);  
        statisticsDisplay.update(nhiệt độ, độ ẩm, áp suất);  
        forecastDisplay.update(nhiệt độ, độ ẩm, áp suất);  
    }  
  
    // các phương pháp WeatherData khác ở đây  
}
```

Lấy các phép đo gần đây nhất bằng cách gọi các phương thức lấy dữ liệu của WeatherData (đã triển khai).

Bây giờ hãy cập nhật màn hình...

Gọi từng phần tử hiển thị để cập nhật nội dung hiển thị, truyền cho nó các phép đo gần đây nhất.



Chuốt bút chì của bạn

Dựa trên lần triển khai đầu tiên, điều nào sau đây đúng?
(Chọn tất cả những câu trả lời đúng.)

A. Chúng tôi đang mã hóa theo các triển khai cụ thể, không phải theo giao diện.
B. Đôi với mỗi phần tử hiển thị mới, chúng ta cần dễ thay đổi mã.
C. Chúng tôi không có cách nào để thêm (hoặc xóa) các phần tử hiển thị trong thời gian chạy.
D. Các thành phần hiển thị không triển khai giao diện chung.
E. Chúng ta chưa bao hàm được phần thay đổi.
F. Chúng tôi đang vi phạm việc đóng gói của Lớp WeatherData.

Định nghĩa của SWAG: Khoa học hoang dã A** Đoán

Có vấn đề gì trong cách triển khai của chúng ta?

Hãy nghĩ lại tất cả các khái niệm và nguyên tắc trong Chương 1...

```
công khai void measurementsChanged() {  
  
    float temp = getTemperature();  
    độ ẩm nổi = getHumidity();  
    áp suất nổi = getPressure();  
  
    currentConditionsDisplay.update(nhiệt độ, độ ẩm, áp suất);  
    statisticsDisplay.update(nhiệt độ, độ ẩm, áp suất);  
    forecastDisplay.update(nhiệt độ, độ ẩm, áp suất);  
}
```

Bằng cách mã hóa theo các triển khai cụ thể,
chúng ta không có cách nào để thêm hoặc
xóa các thành phần hiển thị khác mà không
làm thay đổi chương trình.

Khu vực thay đổi, chúng ta
cần phải tóm tắt điều này.



Ít nhất thì chúng ta có vẻ đang sử
dụng một giao diện chung để giao tiếp
với các thành phần hiển thị... tất cả
chúng đều có phương thức update() lấy
các giá trị nhiệt độ, độ ẩm và áp suất.

Ồm, tôi biết mình mới ở
đây, nhưng vì chúng ta đang ở
chương Mẫu quan sát,
có lẽ chúng ta nên bắt đầu sử dụng nó?



Chúng ta sẽ xem xét Observer,
sau đó quay lại và tìm cách áp
dụng nó vào ứng dụng theo dõi thời
tiết.

đáp ứng mẫu quan sát

Gặp gỡ mẫu quan sát viên

Bạn biết cách thức hoạt động của việc đăng ký báo hoặc tạp chí:

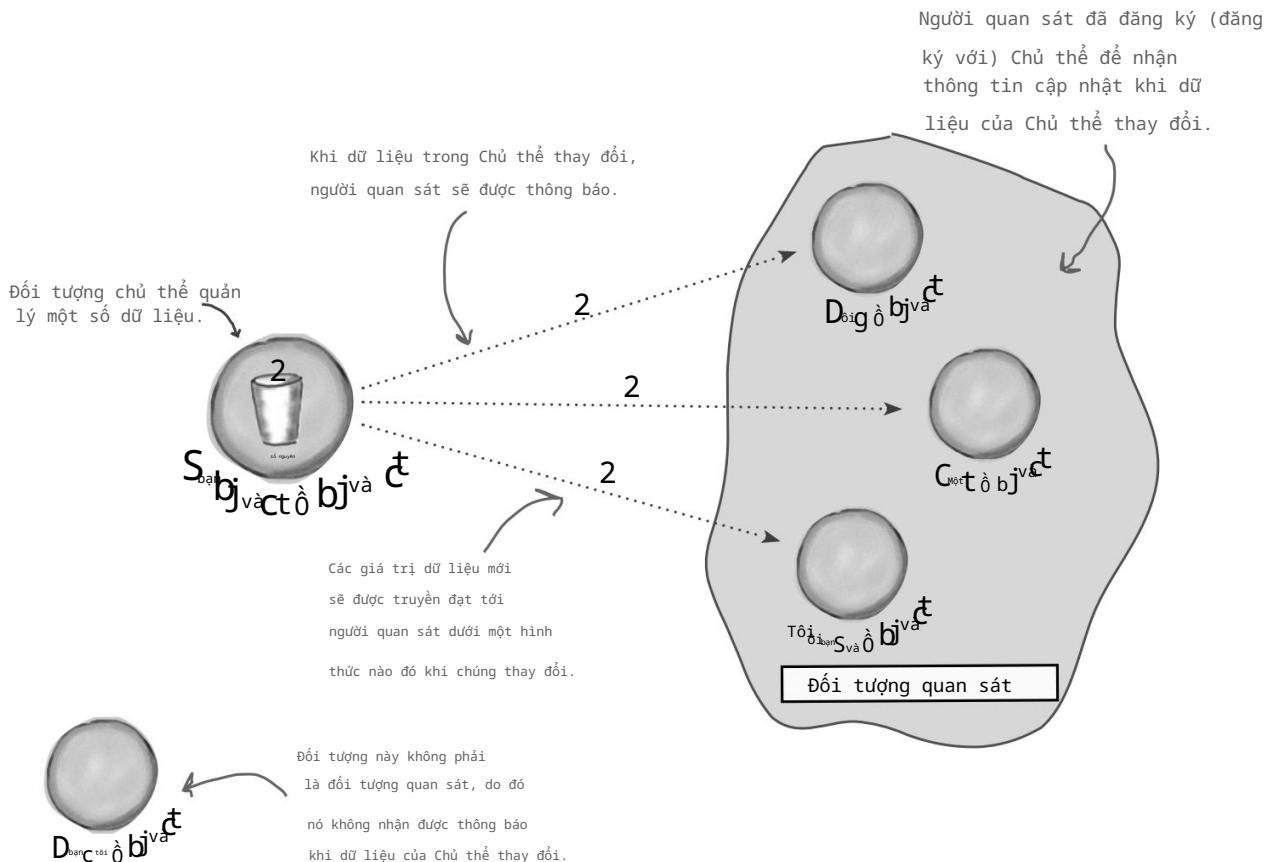
- 1 Một nhà xuất bản báo bắt đầu kinh doanh và xuất bản báo.
- 2 Bạn đăng ký với một nhà xuất bản cụ thể và mỗi lần có phiên bản mới, nó sẽ được gửi đến bạn. Miễn là bạn vẫn là người đăng ký, bạn sẽ nhận được báo mới.
- 3 Bạn hủy đăng ký khi không muốn nhận bài viết nữa và chúng sẽ không còn được gửi đến nữa.
- 4 Trong khi nhà xuất bản vẫn tiếp tục kinh doanh, mọi người, khách sạn, hàng hàng không và các doanh nghiệp khác liên tục đăng ký và hủy đăng ký báo.



Nhà xuất bản + Người đăng ký = Mẫu quan sát

Nếu bạn hiểu về đăng ký báo, bạn sẽ hiểu khá rõ về Mẫu người quan sát, chỉ khác là chúng ta gọi nhà xuất bản là CHỦ ĐỀ và người đăng ký là NGƯỜI QUAN SÁT.

Chúng ta hãy xem xét kỹ hơn:

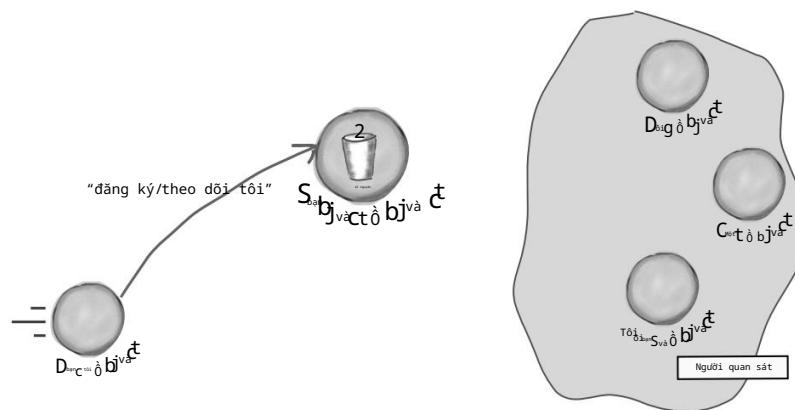


một ngày trong cuộc sống của người quan sát mău

Một ngày trong cuộc sống của Observer Pattern

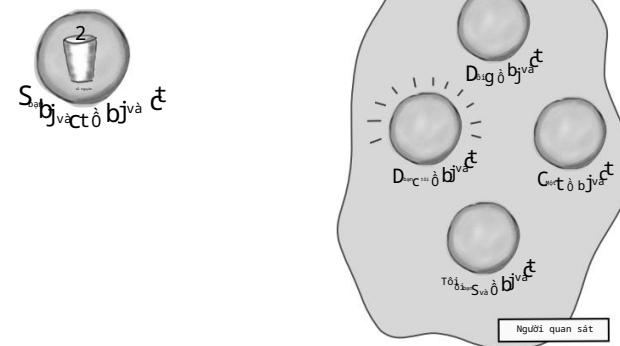
Một vật thể Vịt xuất hiện và nói với Chủ thể rằng nó muốn trở thành người quan sát.

Duck thực sự muốn tham gia vào hành động này; những int mà Subject gửi ra bất cứ khi nào trạng thái của nó thay đổi trông khá thú vị...



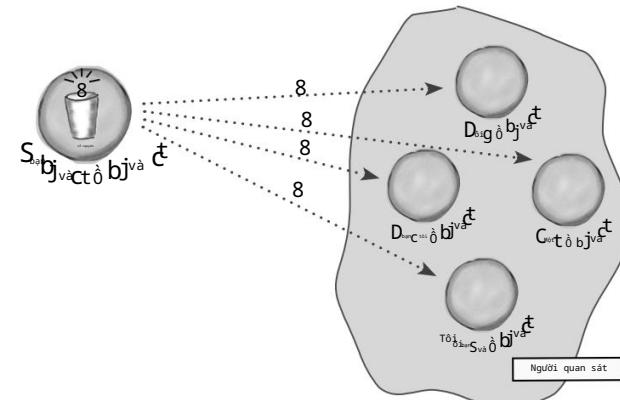
Đối tượng Duck hiện là người quan sát chính thức.

Duck đang rất phấn khích... cậu ấy có tên trong danh sách và đang vô cùng háo hức chờ đợi thông báo tiếp theo để có thể nhận được một bản int.



Chủ thể sẽ nhận được giá trị dữ liệu mới!

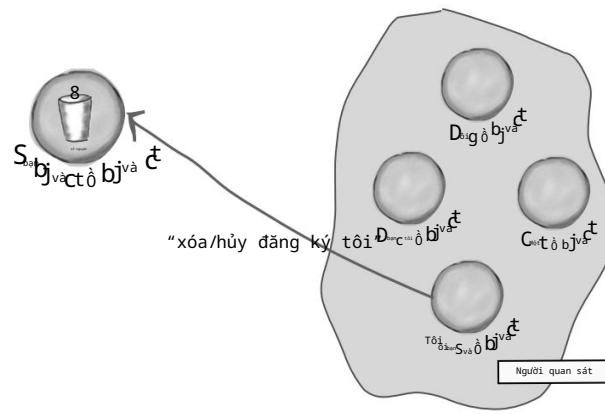
Bây giờ Duck và tất cả những người quan sát còn lại nhận được thông báo rằng Chủ thể đã thay đổi.



mẫu quan sát

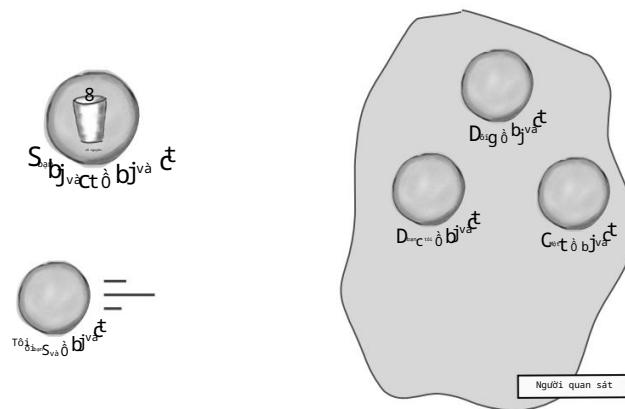
Đối tượng Chuột yêu cầu được xóa khỏi vai trò người quan sát.

Đối tượng Mouse đã nhận được các giá trị int trong nhiều năm và đã chán điều đó, vì vậy nó quyết định đã đến lúc ngừng làm người quan sát.



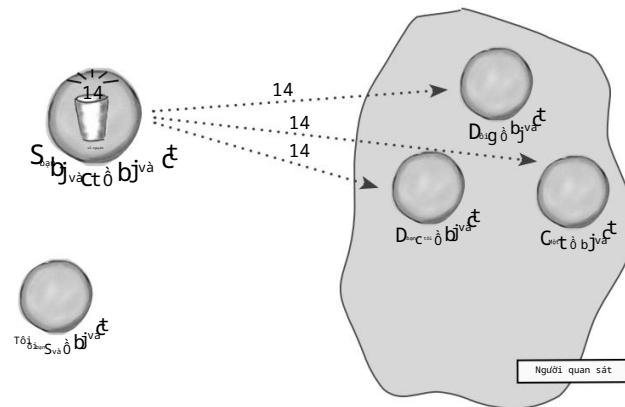
Chuột đã biến mất khỏi đây!

Chủ thẻ xác nhận yêu cầu của Chuột và xóa nó khỏi tập hợp người quan sát.



Chủ ngũ có một int mới.

Tất cả người quan sát đều nhận được thông báo khác, ngoại trừ Chuột không còn được quan sát nữa. Đừng nói với ai, nhưng Chuột thầm nhớ những số nguyên đó... có thể một ngày nào đó nó sẽ lại yêu cầu được làm người quan sát.



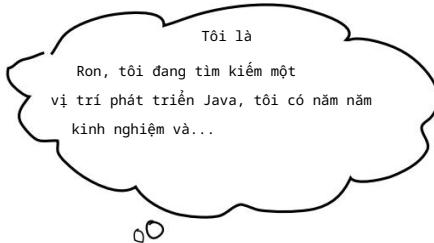
bạn đang ở đây 4 47

kịch năm phút



Kịch năm phút: chủ đề để quan sát

Trong tiểu phẩm ngày hôm nay, hai nhà phát triển phần mềm thời kỳ hậu bong bóng gấp phải một tay săn đầu người thực sự...



1

Phần mềm

Nhà phát triển số 1

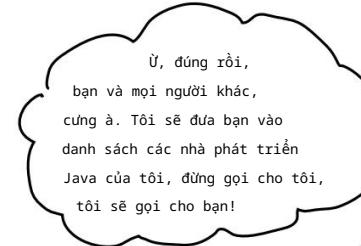
Xin chào, tôi là Jill, tôi đã viết rất nhiều hệ thống EJB, tôi quan tâm đến bất kỳ công việc nào liên quan đến phát triển Java.



3

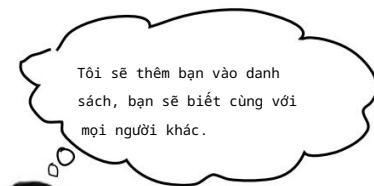
Phần mềm

Nhà phát triển #2



2

Người săn đầu người/Chủ đề

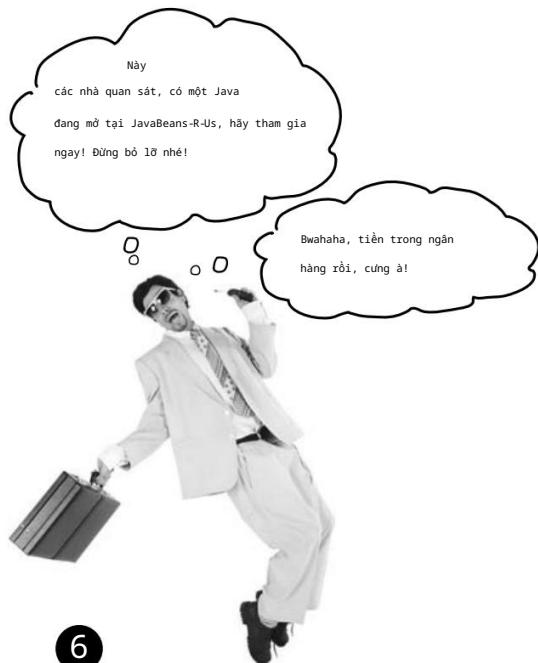


4

Chủ đề

mẫu quan sát

5 Trong khi đó, cuộc sống của Ron và Jill vẫn tiếp diễn; nếu có công việc Java nào đó, họ sẽ được thông báo, sau cùng, họ là người quan sát. máy chủ.



mẫu quan sát được xác định

Hai tuần sau...



Jill yêu cuộc sống và không còn là người quan sát nữa.
Cô ấy cũng đang tận hưởng khoản tiền thưởng ký hợp
đồng hậu hĩnh mà cô nhận được vì công ty không phải
trả tiền cho người săn đầu người.



Nhưng điều gì đã xảy ra với Ron thân yêu của chúng ta? Chúng ta nghe nói
anh ấy đang đánh bại kẻ săn đầu người trong trò chơi của chính mình.
Anh ấy không chỉ vẫn là người quan sát mà còn có danh sách
cuộc gọi riêng và anh ấy đang thông báo cho những người quan sát mình.
Ron vừa là chủ thẻ vừa là người quan sát.

mẫu quan sát

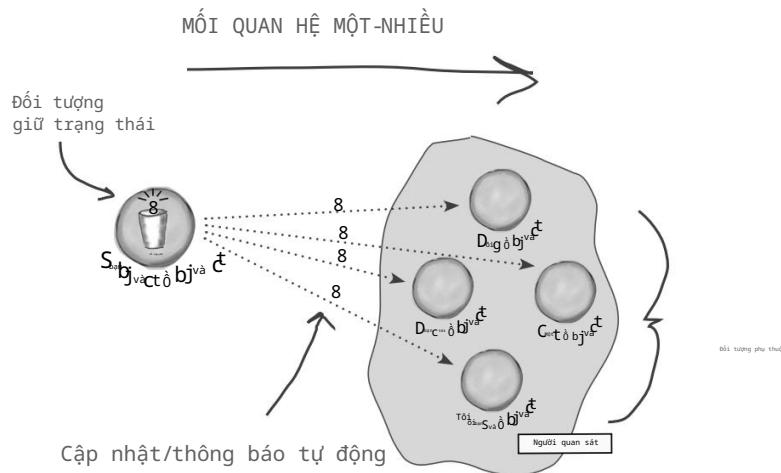
Mẫu quan sát được định nghĩa

Khi bạn đang cố gắng hình dung Mô hình Người quan sát, dịch vụ đăng ký báo với nhà xuất bản và người đăng ký là một cách tốt để hình dung mô hình này.

Tuy nhiên, trong thế giới thực, bạn thường sẽ thấy Mẫu quan sát được định nghĩa như thế này:

Mẫu Observer định nghĩa mối quan hệ phụ thuộc một-nhiều giữa các đối tượng để khi một đối tượng thay đổi trạng thái, tất cả các đối tượng phụ thuộc đều được thông báo và cập nhật tự động.

Chúng ta hãy liên hệ định nghĩa này với cách chúng ta đã nói về mô hình:



Chủ thể và người quan sát xác định mối quan hệ một-nhiều.

Người quan sát phụ thuộc vào chủ thể theo cách mà khi trạng thái của chủ thể thay đổi, người quan sát sẽ được thông báo. Tùy thuộc vào kiểu thông báo, người quan sát cũng có thể được cập nhật các giá trị mới.

Như bạn sẽ khám phá, có một số cách khác nhau để triển khai Mẫu quan sát nhưng hầu hết đều xoay quanh thiết kế lớp bao gồm giao diện Chủ thể và Quan sát viên.

Chúng ta hãy cùng xem nhé...

Mẫu Observer định nghĩa mối quan hệ một-nhiều giữa một tập hợp các đối tượng.

Khi trạng thái của một đối tượng thay đổi, tất cả các đối tượng phụ thuộc vào nó đều được thông báo.

khớp nối lồng léo

Mẫu Observer được định nghĩa sơ đồ lớp

Đây là giao diện Chủ đề.

Các đối tượng sử dụng giao diện này để đăng ký làm người quan sát và cũng để xóa tên mình khỏi danh sách người quan sát.

Một chủ thể cụ thể luôn

triển khai giao diện Chủ đề. Ngoài các phương

thức register và remove, chủ thể cụ thể triển khai

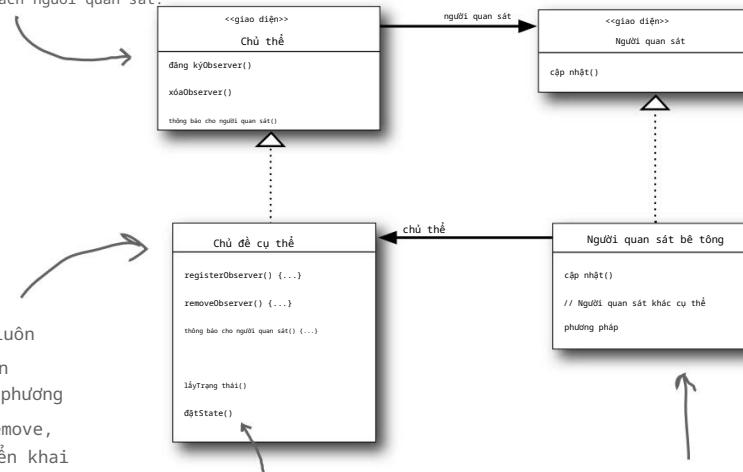
phương thức notifyObservers() được sử dụng để cập nhật tất cả các observer hiện tại

bất cứ khi nào trạng thái thay đổi. (thông tin chi tiết hơn ở phần sau).

Mỗi chủ thể có thể có nhiều người quan sát.

Tất cả các observer tiềm năng cần triển khai giao diện Observer. Giao diện

này chỉ có một phương thức, update(), được gọi khi trạng thái của Subject thay đổi.



Người quan sát cụ thể có thể là bất kỳ lớp nào triển khai giao diện Người quan sát. Mỗi người quan sát đăng ký với một chủ thể cụ thể để nhận cập nhật.

không có Những câu hỏi ngớ ngẩn

H: Điều này có liên quan gì với mối quan hệ môt-nhiều?

H: Sự phụ thuộc đến từ đâu? vào đây?

A: Với mẫu Observer,

Chủ đề là đối tượng chứa trạng thái và điều khiển trạng thái đó. Vì vậy, có MỘT chủ đề có trạng thái. Mặt khác, người quan sát sử dụng trạng thái, ngay cả khi họ không sở hữu nó. Có nhiều người quan sát và họ dựa vào Chủ đề để cho họ biết khi nào trạng thái của nó thay đổi. Vì vậy, có một mối quan hệ giữa MỘT Chủ đề với NHIỀU Người quan sát.

A: Bởi vì chủ ngữ là duy nhất

chủ sở hữu dữ liệu đó, người quan sát phụ thuộc vào chủ đề để cập nhật chúng khi dữ liệu thay đổi. Điều này dẫn đến thiết kế 00 sạch hơn so với việc cho phép nhiều đối tượng kiểm soát cùng một dữ liệu.

Sức mạnh của sự kết nối lỏng lẻo

Khi hai vật thể được kết hợp lỏng lẻo, chúng có thể tương tác nhưng lại có rất ít hiểu biết về nhau.

Mẫu quan sát cung cấp thiết kế đối tượng trong đó chủ thể và người quan sát được kết hợp lỏng lẻo.

Tại sao?

Điều duy nhất mà chủ thể biết về một người quan sát là nó triển khai một giao diện nhất định (giao diện Người quan sát). Nó không cần biết lớp cụ thể của người quan sát, nó làm gì hoặc bất kỳ điều gì khác về nó.

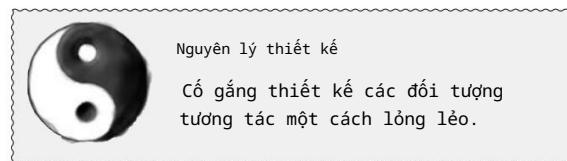
Chúng ta có thể thêm người quan sát mới bất cứ lúc nào. Vì chủ thể chỉ phụ thuộc vào danh sách các đối tượng triển khai giao diện Người quan sát, chúng ta có thể thêm người quan sát mới bất cứ khi nào chúng ta muốn. Trên thực tế, chúng ta có thể thay thế bất kỳ người quan sát nào khi chạy bằng một người quan sát khác và chủ thể sẽ tiếp tục chạy. Tương tự như vậy, chúng ta có thể xóa người quan sát bất cứ lúc nào.

Chúng ta không bao giờ cần phải sửa đổi chủ thể để thêm các loại quan sát viên mới. Giả sử chúng ta có một lớp cụ thể mới xuất hiện cần phải là một quan sát viên. Chúng ta không cần phải thực hiện bất kỳ thay đổi nào đối với chủ thể để phù hợp với loại lớp mới, tất cả những gì chúng ta phải làm là triển khai giao diện Observer trong lớp mới và đăng ký làm một quan sát viên. Chủ thể không quan tâm; nó sẽ gửi thông báo đến bất kỳ đối tượng nào triển khai giao diện Observer.

Chúng ta có thể tái sử dụng chủ thể hoặc người quan sát độc lập với nhau. Nếu chúng ta có cách sử dụng khác cho chủ thể hoặc người quan sát, chúng ta có thể dễ dàng tái sử dụng chúng vì cả hai không liên kết chặt chẽ với nhau.

Sự thay đổi của chủ thể hoặc người quan sát sẽ không ảnh hưởng đến nhau. Vì hai phần này được kết hợp lỏng lẻo nên chúng ta có thể thoải mái thay đổi bất kỳ phần nào, miễn là các đối tượng vẫn đáp ứng được nghĩa vụ triển khai giao diện chủ thể hoặc người quan sát.

Bạn có thể xác định được bao nhiêu loại thay đổi khác nhau ở đây?



Thiết kế lỏng lẻo cho phép chúng ta xây dựng các hệ thống linh hoạt có khả năng xử lý thay đổi vì chúng giảm thiểu sự phụ thuộc lẫn nhau giữa các đối tượng.

lập kế hoạch cho trạm thời tiết



Chuốt bút chì của bạn

Trước khi tiếp tục, hãy thử phác thảo các lớp bạn cần để triển khai Weather Station, bao gồm lớp WeatherData và các thành phần hiển thị của lớp này.

Đảm bảo sơ đồ của bạn cho thấy tất cả các thành phần khớp với nhau như thế nào và cũng cho thấy cách một nhà phát triển khác có thể triển khai phần tử hiển thị của riêng mình.

Nếu bạn cần một chút trợ giúp, hãy đọc trang tiếp theo; các đồng đội của bạn đã thảo luận về cách thiết kế Trạm thời tiết rồi.

Cuộc trò chuyện trong ô

Quay trở lại dự án Trạm thời tiết, các đồng đội của bạn đã bắt đầu suy nghĩ về vấn đề này...



Mary: Vâng, biết được điều này sẽ giúp ích cho chúng ta.

Sue: Đúng rồi... nhưng chúng ta áp dụng nó như thế nào?

Mary: Ừm. Chúng ta hãy xem lại định nghĩa nhé:

Mẫu Observer định nghĩa mối quan hệ phụ thuộc một-nhiều giữa các đối tượng để khi một đối tượng thay đổi trạng thái, tất cả các đối tượng phụ thuộc đều được thông báo và cập nhật tự động.

Mary: Điều đó thực sự có ý nghĩa khi bạn nghĩ về nó. Lớp WeatherData của chúng ta là "một" và "nhiều" của chúng ta là các phần tử hiển thị khác nhau sử dụng thời tiết phép đo.

Sue: Đúng vậy. Lớp WeatherData chắc chắn có trạng thái... đó là nhiệt độ, độ ẩm và áp suất khí quyển, và những thứ đó chắc chắn thay đổi.

Mary: Đúng vậy, và khi những phép đo đó thay đổi, chúng ta phải thông báo cho tất cả các thành phần hiển thị để chúng có thể thực hiện bất cứ điều gì chúng muốn với các phép đo đó.

Sue: Tuyệt, giờ tôi nghĩ tôi hiểu cách áp dụng Mẫu quan sát vào bài toán Trạm thời tiết của chúng ta rồi.

Mary: Vẫn còn một vài điều cần cân nhắc mà tôi vẫn chưa chắc mình đã hiểu rõ.

Sue: Như thế nào cơ?

Mary: Đầu tiên, làm sao chúng ta có thể đưa thông tin do thời tiết vào phần hiển thị?

Sue: Vâng, nhìn lại hình ảnh của Observer Pattern, nếu chúng ta biến đối tượng WeatherData thành chủ thể và các phần tử hiển thị thành người quan sát, thì các màn hình sẽ tự đăng ký với đối tượng WeatherData để có được thông tin chúng muốn, đúng không?

Mary: Đúng vậy... và một khi Trạm thời tiết biết về một thành phần hiển thị, thì nó có thể chỉ cần gọi một phương thức để thông báo về các phép đo.

Sue: Chúng ta phải nhớ rằng mọi thành phần hiển thị đều có thể khác nhau... vì vậy tôi nghĩ đó là lý do cần có một giao diện chung. Mặc dù mọi thành phần có kiểu khác nhau, nhưng tất cả chúng đều phải triển khai cùng một giao diện để đối tượng WeatherData biết cách gửi các phép đo cho chúng.

Mary: Tôi hiểu ý bạn. Vậy thì mọi màn hình sẽ có, chẳng hạn, phương thức update() mà WeatherData sẽ gọi.

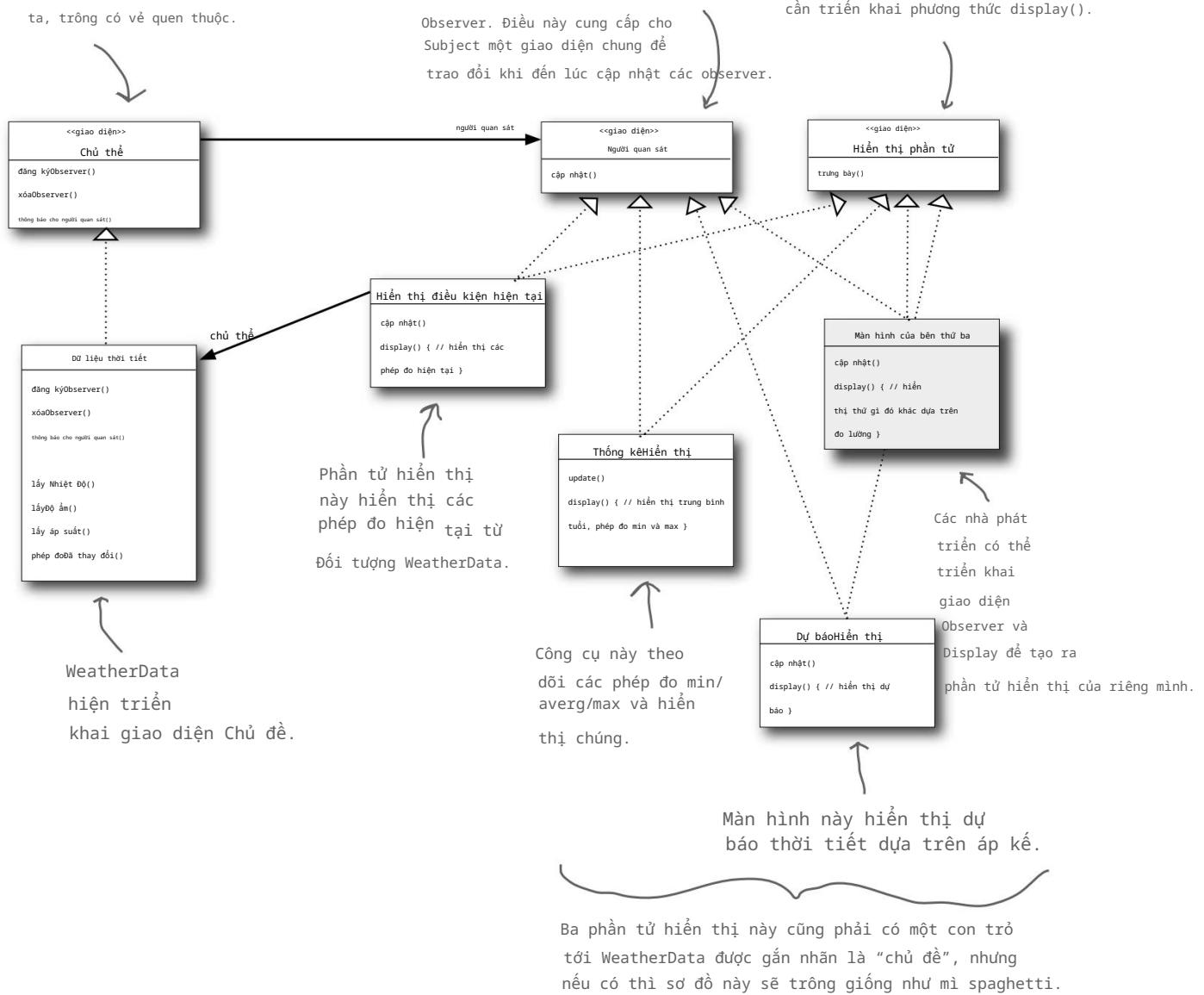
Sue: Và update() được định nghĩa trong một giao diện chung mà tất cả các phần tử đều triển khai.

thiết kế trạm thời tiết

Thiết kế Trạm thời tiết

Biểu đồ này so với biểu đồ của bạn thế nào?

Đây là giao diện chủ đề của chúng ta, trông có vẻ quen thuộc.



Triển khai Trạm thời tiết

Chúng ta sẽ bắt đầu triển khai bằng cách sử dụng sơ đồ lớp và làm theo hướng dẫn của Mary và Sue (từ một vài trang trước). Bạn sẽ thấy sau trong chương này rằng Java cung cấp một số hỗ trợ tích hợp cho mẫu Observer, tuy nhiên, chúng ta sẽ tự mình thực hiện ngay bây giờ. Trong khi trong một số trường hợp, bạn có thể sử dụng hỗ trợ tích hợp của Java, thì trong nhiều trường hợp, việc tự xây dựng sẽ linh hoạt hơn (và không quá khó). Vì vậy, chúng ta hãy bắt đầu với các giao diện:

```
Giao diện công khai Chủ đề {
    public void registerObserver(Observer o);
    công khai void removeObserver(Observer o);
    công khai void notifyObservers();
}
```

Cả hai phương pháp này đều lấy Observer làm đối số; nghĩa là Observer cần được đăng ký hoặc xóa.

```
giao diện công khai Observer {
    public void update(float nhiệt độ, float độ ẩm, float áp suất);
}
```

Phương pháp này được gọi để thông báo cho tất cả người quan sát khi trạng thái của Chủ đề đã thay đổi.

```
giao diện công khai DisplayElement {
    công khai void display();
}
```

Giao diện Observer được triển khai bởi tất cả các observer, vì vậy tất cả họ đều phải triển khai phương thức update(). Ở đây chúng ta đang làm theo Mary và Sue và chuyển các phép đo cho các observer.

Giao diện DisplayElement chỉ bao gồm một phương thức, display(), mà chúng ta sẽ gọi khi cần hiển thị phần tử hiển thị.

não Apower

Mary và Sue nghĩ rằng việc truyền trực tiếp các phép đo cho người quan sát là phương pháp cập nhật trạng thái đơn giản nhất. Bạn có nghĩ điều này là khôn ngoan không? Gợi ý: đây có phải là một lĩnh vực của ứng dụng có thể thay đổi trong tương lai không? Nếu có thay đổi, liệu thay đổi đó có được đóng gói tốt hay sẽ yêu cầu thay đổi ở nhiều phần của mã?

Bạn có thể nghĩ ra cách nào khác để giải quyết vấn đề truyền trạng thái đã cập nhật cho người quan sát không?

Đừng lo lắng, chúng ta sẽ quay lại quyết định thiết kế này sau khi hoàn tất quá trình triển khai ban đầu.

thực hiện trạm thời tiết

Triển khai giao diện Chủ đề trong Dữ liệu thời tiết

Bạn còn nhớ lần đầu tiên chúng ta thử triển khai lớp WeatherData ở đầu chương không? Bạn có thể muốn làm mới trí nhớ của mình. Böyle giờ là lúc quay lại và làm mọi thứ với Observer Pattern trong đầu...

NHỚ RẰNG: chúng tôi không cung cấp các câu lệnh import và package trong danh sách mã.

Lấy mã nguồn đầy đủ từ trang web headfirstlabs. Bạn sẽ tìm thấy URL trên trang xxxiii trong phần Giới thiệu.

```

lớp công khai WeatherData thực hiện Chủ đề {
    người quan sát ArrayList riêng tư;
    nhiệt độ nổi riêng;
    độ ẩm nổi riêng;
    áp suất nổi riêng;

    công khai WeatherData() {
        người quan sát = new ArrayList();
    }

    công khai void registerObserver(Observer o) {
        observers.add(o);
    }

    công khai void removeObserver(Observer o) {
        int i = observers.indexOf(o);
        nếu (i >= 0) {
            observers.remove(i);
        }
    }

    công khai void thông báo cho người quan sát () {
        đối với (int i = 0; i < observers.size(); i++) {
            Người quan sát người quan sát = (Người quan sát)observers.get(i);
            observer.update(nhiệt độ, độ ẩm, áp suất);
        }
    }

    công khai void measurementsChanged() {
        thông báo cho người quan sát();
    }

    public void setMeasurements(nhiệt độ nổi, độ ẩm nổi, áp suất nổi) {
        this.temperature = nhiệt độ;
        this.humidity = độ ẩm;
        this.pressure = áp suất;
        phép đoĐã thay đổi();
    }

    // các phương pháp WeatherData khác ở đây
}

```

WeatherData hiện triển khai giao diện Chủ đề.

Chúng tôi đã thêm một ArrayList để chứa các Observer và chúng tôi tạo nó trong hàm tạo.

Khi một người quan sát đăng ký, chúng ta chỉ cần thêm người đó vào cuối danh sách.

Tương tự như vậy, khi một người quan sát muốn hủy đăng ký, chúng tôi chỉ cần xóa họ khỏi danh sách.

Đây là phần thú vị; đây là nơi chúng tôi cho tất cả người quan sát biết về trạng thái.

Bởi vì tất cả chúng đều là Observer, chúng ta biết tất cả chúng đều triển khai update(), do đó chúng ta biết cách thông báo cho chúng.

Chúng tôi thông báo cho Người quan sát khi nhận được số liệu đo đặc mới nhất từ Trạm thời tiết.

Được rồi, trong khi chúng ta muốn gửi một trạm thời tiết nhỏ xinh với mỗi cuốn sách, nhà xuất bản sẽ không lấy sẽ làm điều đó. Vì vậy, thay vì đọc dữ liệu thời tiết thực tế từ thiết bị, chúng ta sẽ sử dụng phương pháp này để kiểm tra các thành phần hiển thị của mình. Hoặc, để giải trí, bạn có thể viết mã để lấy số đo từ web.

Bây giờ, chúng ta hãy xây dựng các thành phần hiển thị đó

Bây giờ chúng ta đã sắp xếp lớp WeatherData của mình, đã đến lúc xây dựng các Thành phần Hiển thị. Weather-O-Rama đã sắp xếp ba thành phần: màn hình hiển thị điều kiện hiện tại, màn hình hiển thị thông kê và màn hình hiển thị dự báo. Chúng ta hãy xem màn hình hiển thị điều kiện hiện tại; khi bạn đã có cảm nhận tốt về thành phần hiển thị này, hãy kiểm tra các màn hình hiển thị thống kê và dự báo trong thư mục mã đầu tiên. Bạn sẽ thấy chúng rất giống nhau.

```

lớp công khai CurrentConditionsDisplay triển khai Observer, DisplayElement {
    nhiệt độ nổi riêng;
    độ ẩm nổi riêng;
    Chủ đề riêng tư weatherData;

    public CurrentConditionsDisplay(Subject weatherData) {
        this.weatherData = weatherData;
        weatherData.registerObserver(cái này);
    }

    public void update(float nhiệt độ, float độ ẩm, float áp suất) {
        this.temperature = nhiệt độ;
        this.humidity = độ ẩm;
        trưng bày();
    }

    công khai void display() {
        System.out.println("Điều kiện hiện tại: "
            + "Độ F và " + nhiệt độ
            + độ ẩm + "% độ ẩm");
    }
}

```

Màn hình này triển khai Observer để có thể lấy những thay đổi từ đối tượng WeatherData.

Nó cũng triển khai DisplayElement, vì API của chúng ta sẽ yêu cầu tất cả các phần tử hiển thị phải triển khai giao diện này.

Hàm tạo được truyền vào đối tượng weatherData (Chủ thể) và chúng ta sử dụng nó để đăng ký màn hình hiển thị như một trình quan sát.

Khi hàm update() được gọi, chúng ta lưu nhiệt độ và độ ẩm và gọi hàm display().

Phương thức display() chỉ in ra nhiệt độ và độ ẩm gần đây nhất.

Q: Update() có phải là nơi tốt nhất để hiển thị cuộc gọi?

cách dữ liệu được hiển thị. Chúng ta sẽ thấy điều này khi chúng ta đến mô hình model-view-controller.

A: Trong ví dụ đơn giản này nó đã được thực hiện ý nghĩa khi gọi display() khi các giá trị thay đổi. Tuy nhiên, bạn đúng, có nhiều cách tốt hơn để thiết kế

Q: Tại sao bạn lưu trữ một tham chiếu với Chủ ngữ? Có vẻ như bạn không sử dụng nó nữa sau hàm tạo?

MỘT: Đúng vậy, nhưng trong tương lai chúng ta có thể muốn hủy đăng ký mình là người quan sát và sẽ tiện lợi hơn nếu có sẵn tài liệu tham khảo về chủ đề này.

kiểm tra trạm thời tiết

Bật nguồn cho Trạm thời tiết



1 Đầu tiên, chúng ta hãy tạo một dây nịt thử nghiệm

Trạm thời tiết đã sẵn sàng hoạt động, tất cả những gì chúng ta cần là một số mã để dán mọi thứ lại với nhau. Đây là nỗ lực đầu tiên của chúng tôi. Chúng ta sẽ quay lại sau trong cuốn sách và đảm bảo rằng tất cả các thành phần đều có thể dễ dàng cắm vào thông qua tệp cấu hình. Bây giờ, đây là cách mọi thứ hoạt động:

```
lớp công khai WeatherStation {
    public static void main(String[] args) {
        WeatherData weatherData = new WeatherData();
```

Đầu tiên, tạo
đối tượng

WeatherData.

```
Nếu bạn
không
muốn tải mã
xuống, bạn
có thể bình
luận hai dòng
này và chạy nó.
}
}

CurrentConditionsDisplay currentDisplay =
    mới CurrentConditionsDisplay(weatherData);
StatisticsDisplay statisticsDisplay = new StatisticsDisplay(weatherData);
ForecastDisplay forecastDisplay = new ForecastDisplay(weatherData);

weatherData.setMeasurements(80, 65, 30.4f);
weatherData.setMeasurements(82, 70, 29.2f);
weatherData.setMeasurements(78, 90, 29.2f);
```

Tạo ba màn hình
hiển thị và
truyền đối
tượng WeatherData cho chúng.

Mô phỏng các phép đo
thời tiết mới.

2 Chạy mã và để Observer Pattern thực hiện phép thuật của nó

```
Cửa sổ chỉnh sửa tệp Trợ giúp StormyWeather
%java Trạm thời tiết
Điều kiện hiện tại: 80,0F độ và độ ẩm 65,0%
Nhiệt độ trung bình/tối đa/tối thiểu = 80.0/80.0/80.0
Dự báo: Thời tiết sẽ cải thiện!
Điều kiện hiện tại: 82,0F độ và độ ẩm 70,0%
Nhiệt độ trung bình/tối đa/tối thiểu = 81,0/82,0/80,0
Dự báo: Cần chú ý thời tiết mát mẻ và mưa
Điều kiện hiện tại: 78,0F độ và độ ẩm 90,0%
Nhiệt độ trung bình/tối đa/tối thiểu = 80,0/82,0/78,0
Dự báo: Vẫn như cũ
%
```



Chuốt bút chì của bạn

Johnny Hurricane, CEO của Weather-O-Rama vừa gọi điện, họ không thể giao hàng nếu không có phần tử hiển thị Chỉ số nhiệt. Sau đây là thông tin chi tiết:

Chỉ số nhiệt là chỉ số kết hợp nhiệt độ và độ ẩm để xác định nhiệt độ biểu kiến (thực tế cảm thấy nóng như thế nào). Để tính chỉ số nhiệt, bạn lấy nhiệt độ, T, và độ ẩm tương đối, RH, và sử dụng công thức này:

chỉ số nhiệt =

$$\begin{aligned}
 & 16.923 + 1.85212 * 10^{-1} * T + 5.37941 * RH - 1.00254 * 10^{-1} * T \\
 & * RH + 9.41695 * 10^{-3} * T_2 + 7.28898 * 10^{-3} * RH_2 + 3.45372 * 10^{-4} \\
 & * T_2 * RH - 8.14971 * 10^{-4} * T * RH_2 + 1.02102 * 10^{-5} * T_2 * RH_2 - 3.8646 * \\
 & 10^{-5} * T_3 + 2.91583 * 10^{-5} * RH_3 + 1.42721 * 10^{-6} * T_3 * RH + 1.97483 * 10^{-7} \\
 & * T * RH_3 - 2.18429 * 10^{-8} * T_3 * RH_2 + 8.43296 * 10^{-10} * T_2 * RH_3 - 4.81975 \\
 & * 10^{-11} * T_3 * RH_3
 \end{aligned}$$

Vậy hãy bắt đầu gõ nhé!

Đùa thôi. Đừng lo, bạn không cần phải nhập công thức đó đâu; chỉ cần tạo HeatIndexDisplay của riêng bạn thôi. tệp java và sao chép công thức từ heatindex.txt vào đó.

Bạn có thể lấy heatindex.txt từ headfirstlabs.com

Nó hoạt động như thế nào? Bạn phải tham khảo Head First Meteorology hoặc thử hỏi ai đó ở Cơ quan Thời tiết Quốc gia (hoặc thử tìm kiếm trên Google).

Khi hoàn tất, điều ra của bạn sẽ trông như thế này:

```

Cửa sổ chỉnh sửa tập tin Trợ giúp OverdaRainbow
%java Trạm thời tiết
Điều kiện hiện tại: 80,0F độ và độ ẩm 65,0%
Nhiệt độ trung bình/tối đa/tối thiểu = 80,0/80,0/80,0
Dự báo: Thời tiết sẽ cải thiện!
Chỉ số nhiệt là 82.95535

Điều kiện hiện tại: 82,0F độ và độ ẩm 70,0%
Nhiệt độ trung bình/tối đa/tối thiểu = 81,0/82,0/80,0
Dự báo: Cần chú ý thời tiết mát mẻ và mưa
Chỉ số nhiệt là 86.90124

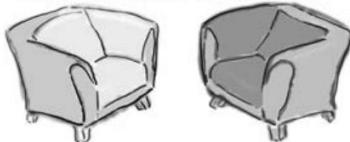
Điều kiện hiện tại: 78,0F độ và độ ẩm 90,0%
Nhiệt độ trung bình/tối đa/tối thiểu = 80,0/82,0/78,0
Dự báo: Vẫn như cũ
Chỉ số nhiệt là 83.64967
%

```

Sau đây là những thay đổi trong kết quả này.

trò chuyện bên lò sưởi: chủ thể và người quan sát

Fireside Chats



Bài nói chuyện tôi nay: Chủ thể và Người quan sát tranh luận về cách thức đúng đắn để truyền đạt thông tin của tiểu bang đến Người quan sát.

Chủ thể

Người quan sát

Tôi rất vui vì cuối cùng chúng ta cũng có cơ hội trò chuyện
người.

Thật sao? Tôi nghĩ là anh không quan tâm nhiều đến chúng tôi,
nhưng Người quan sát.

Vâng, tôi làm công việc của mình, phải không? Tôi luôn nói với bạn
những gì đang diễn ra... Chỉ vì tôi không thực sự biết bạn là ai
không có nghĩa là tôi không quan tâm. Và bên cạnh đó, tôi biết điều
quan trọng nhất về bạn—
Bạn triển khai giao diện Observer.

Vâng, nhưng đó chỉ là một phần nhỏ về con người tôi. Dù sao thì
tôi cũng biết nhiều hơn về bạn...

Ồ đúng rồi, thê nào nhỉ?

Vâng, bạn luôn chuyển trạng thái của mình cho chúng tôi,
nhưng Người quan sát, để chúng tôi có thể thấy những
giá trị đang diễn ra bên trong bạn. Điều này đôi khi hơi khó
chiếu...

Xin lỗi nhé. Tôi phải gửi trạng thái của mình cùng với thông báo để
tất cả những Người quan sát lười biếng biết chuyện gì đã xảy ra!

Được rồi, đợi một chút; thứ nhất, chúng tôi không lười biếng,
chúng tôi chỉ có những việc khác phải làm trong thời gian chờ
thông báo cực kỳ quan trọng của ông, ông Subject, và thứ hai, tại
sao ông không để chúng tôi đến gặp ông để có được trạng thái
chúng tôi mong muốn thay vì đẩy nó ra cho tất cả mọi người?

Vâng... Tôi đoán điều đó có thể hiệu quả. Tuy nhiên, tôi phải mở
rộng bản thân hơn nữa để tất cả các bạn Observers có thể vào và có được
trạng thái mà các bạn cần. Điều đó có thể hơi nguy hiểm. Tôi không thể
để các bạn vào và chỉ rình mò xung quanh để xem mọi thứ tôi có.

mẫu quan sát

Chủ thẻ

Vâng, tôi có thể để bạn kéo trạng thái của tôi. Nhưng điều đó không phải sẽ bất tiện hơn cho bạn sao? Nếu bạn phải đến gặp tôi mỗi khi bạn muốn thử gì đó, bạn có thể phải thực hiện nhiều cuộc gọi phương thức để có được tất cả trạng thái bạn muốn. Đó là lý do tại sao tôi thích push hơn... khi đó bạn có mọi thứ bạn cần trong một thông báo.

Vâng, tôi có thể thấy lợi thế khi thực hiện cả hai cách. Tôi nhận thấy rằng có một Java Observer Pattern tích hợp cho phép bạn sử dụng lệnh đẩy hoặc lệnh kéo.

Tuyệt... có lẽ tôi sẽ được chứng kiến một ví dụ điển hình về sự thay đổi suy nghĩ.

Người quan sát

Tại sao bạn không viết một số phương thức getter công khai cho phép chúng ta rút ra trạng thái mà chúng ta nhu cầu?

Đừng thúc ép quá! Có rất nhiều loại Người quan sát khác nhau, không có cách nào bạn có thể dự đoán mọi thứ chúng tôi cần. Hãy để chúng tôi đến với bạn để có được trạng thái chúng tôi cần. Theo cách đó, nếu một số người trong chúng tôi chỉ cần một chút trạng thái, chúng tôi không bị buộc phải có tất cả. Nó cũng giúp mọi thứ dễ dàng hơn để sửa đổi sau này. Ví dụ, bạn mở rộng bản thân và thêm một số trạng thái nữa, nếu bạn sử dụng pull, bạn không phải di xung quanh và thay đổi các lệnh gọi cập nhật trên mọi người quan sát, bạn chỉ cần thay đổi bản thân để cho phép nhiều phương thức getter hơn truy cập vào trạng thái bổ sung của chúng tôi.

Ồ thật sao? Tôi nghĩ chúng ta sẽ xem xét điều đó
Ké tiếp....

Cái gì, chúng ta đồng ý về điều gì đó à? Tôi đoán là vẫn luôn có hy vọng.

mẫu quan sát tích hợp của java

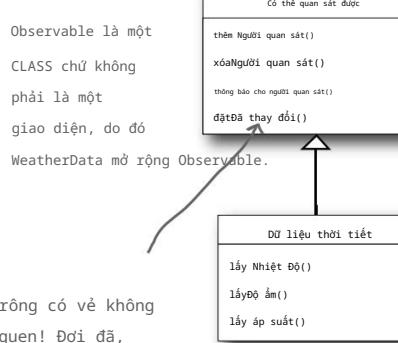
Sử dụng Java tích hợp sẵn Mẫu quan sát

Cho đến nay chúng ta đã triển khai mã của riêng mình cho Observer Pattern, nhưng Java đã tích hợp hỗ trợ trong một số API của nó. Phổ biến nhất là giao diện Observer và lớp Observable trong gói java.util. Chúng khá giống với giao diện Subject và Observer của chúng ta, nhưng cung cấp cho bạn rất nhiều chức năng ngay khi cài đặt. Bạn cũng có thể triển khai kiểu cập nhật push hoặc pull cho observers của mình, như bạn sẽ thấy.

Để hiểu rõ hơn về `java.util.Observer` và `java.util.Observable`, hãy xem thiết kế 00 được thiết kế lại này cho WeatherStation:



Lớp Observable theo dõi tất
cả người quan sát của bạn
và thông báo cho họ thay bạn.



Trong có vẻ không
quen! Để đã,
chúng ta sẽ nói đến
diều này sau...

Đây là Subject của chúng ta, mà giờ
đây chúng ta cũng có thể gọi là
Observable. Chúng ta không cần các
phương thức `register()`, `remove()`
và `notifyObservers()` nữa; chúng ta
thứa hưởng hành vi đó từ siêu lớp.

Điều này có vẻ quen thuộc.

Trên thực tế, nó giống hệt với
sơ đồ lớp trước của chúng ta!

Chúng tôi đã bỏ qua
giao diện
`DisplayElement`, nhưng
tất cả các màn hình
vẫn triển khai giao diện này.

Sẽ có một vài thay đổi đối với phương thức `update()`
trong Observer cụ thể, nhưng về cơ bản thì ý
tưởng vẫn như vậy... chúng ta có một giao diện
Observer chung, với phương thức `update()` được gọi bởi Subject.

Cách thức hoạt động của Observer Pattern tích hợp của Java

Observer Pattern tích hợp hoạt động hơi khác so với triển khai mà chúng tôi sử dụng trên Weather Station. Sự khác biệt rõ ràng nhất là WeatherData (chủ đề của chúng tôi) hiện mở rộng lớp Observable và kế thừa các phương thức add, delete và notify Observer (cùng một số phương thức khác). Sau đây là cách chúng tôi sử dụng phiên bản Java:

Để một Đối tượng trở thành người quan sát...

Như thường lệ, hãy triển khai giao diện Observer (lần này là giao diện `java.util.Observer`) và gọi `addObserver()` trên bất kỳ đối tượng `Observable` nào. Tương tự như vậy, để xóa bạn khỏi danh sách người quan sát, chỉ cần gọi `deleteObserver()`.

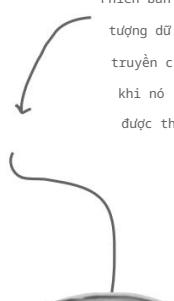
Để Observable gửi thông báo...

Trước hết, bạn cần `Observable` bằng cách mở rộng siêu lớp `java.util.Observable`. Từ đó, quá trình gồm hai bước:

- 1 Đầu tiên bạn phải gọi phương thức `setChanged()` để biểu thị rằng trạng thái đã thay đổi trong đối tượng của bạn

- 2 Sau đó, gọi một trong hai phương thức `notifyObservers()`:

hoặc `notifyObservers()` hoặc `notifyObservers(Object arg)`



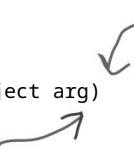
Phiên bản này lấy một đối tượng dữ liệu tùy ý được truyền cho mỗi Observer khi nó được thông báo.

Để Người quan sát nhận được thông báo...

Nó triển khai phương thức cập nhật như trước, nhưng chữ ký của phương thức này hơi khác một chút:

`cập nhật(Observable o, Object arg)`

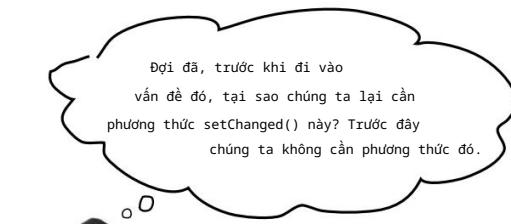
Chủ thể gửi thông báo
được truyền vào dưới dạng
đối số này.



Đây sẽ là đối tượng dữ liệu được truyền
cho `notifyObservers()` hoặc null nếu
không chỉ định đối tượng dữ liệu.

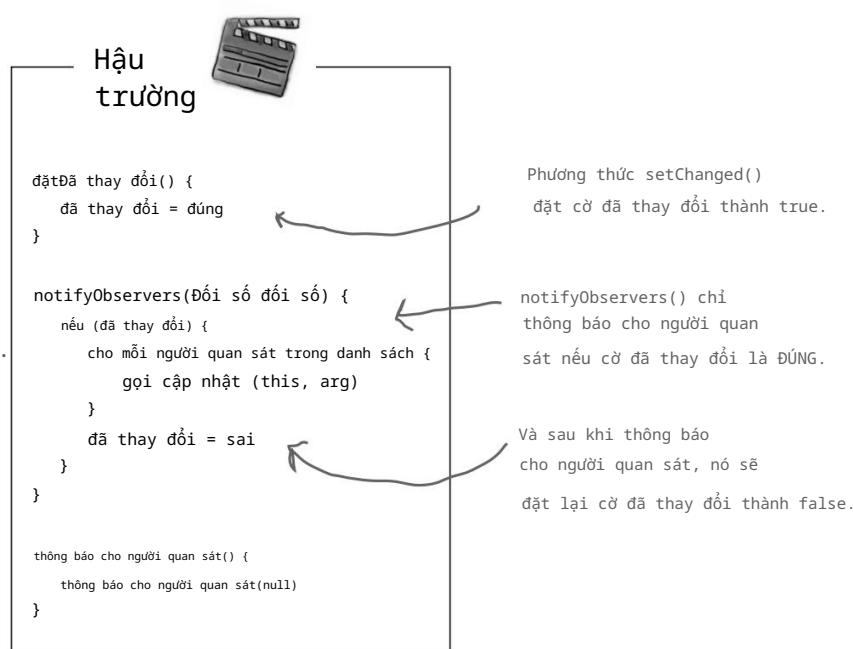
Nếu bạn muốn "đẩy" dữ liệu đến các observer, bạn có thể truyền dữ liệu dưới dạng đối tượng dữ liệu đến phương thức `notifyObserver(arg)`. Nếu không, Observer phải "kéo" dữ liệu mà nó muốn từ đối tượng `Observable` được truyền cho nó. Làm thế nào? Hãy làm lại Weather Station và bạn sẽ thấy.

hậu trường



Phương thức `setChanged()` được sử dụng để biểu thị rằng trạng thái đã thay đổi và `notifyObservers()`, khi được gọi, sẽ cập nhật các observer của nó. Nếu `notifyObservers()` được gọi mà không gọi `setChanged()` trước, các observer sẽ KHÔNG được thông báo. Hãy cùng xem hậu trường của `Observable` để xem cách thức hoạt động của nó:

Mã giả cho lớp `Observable`.

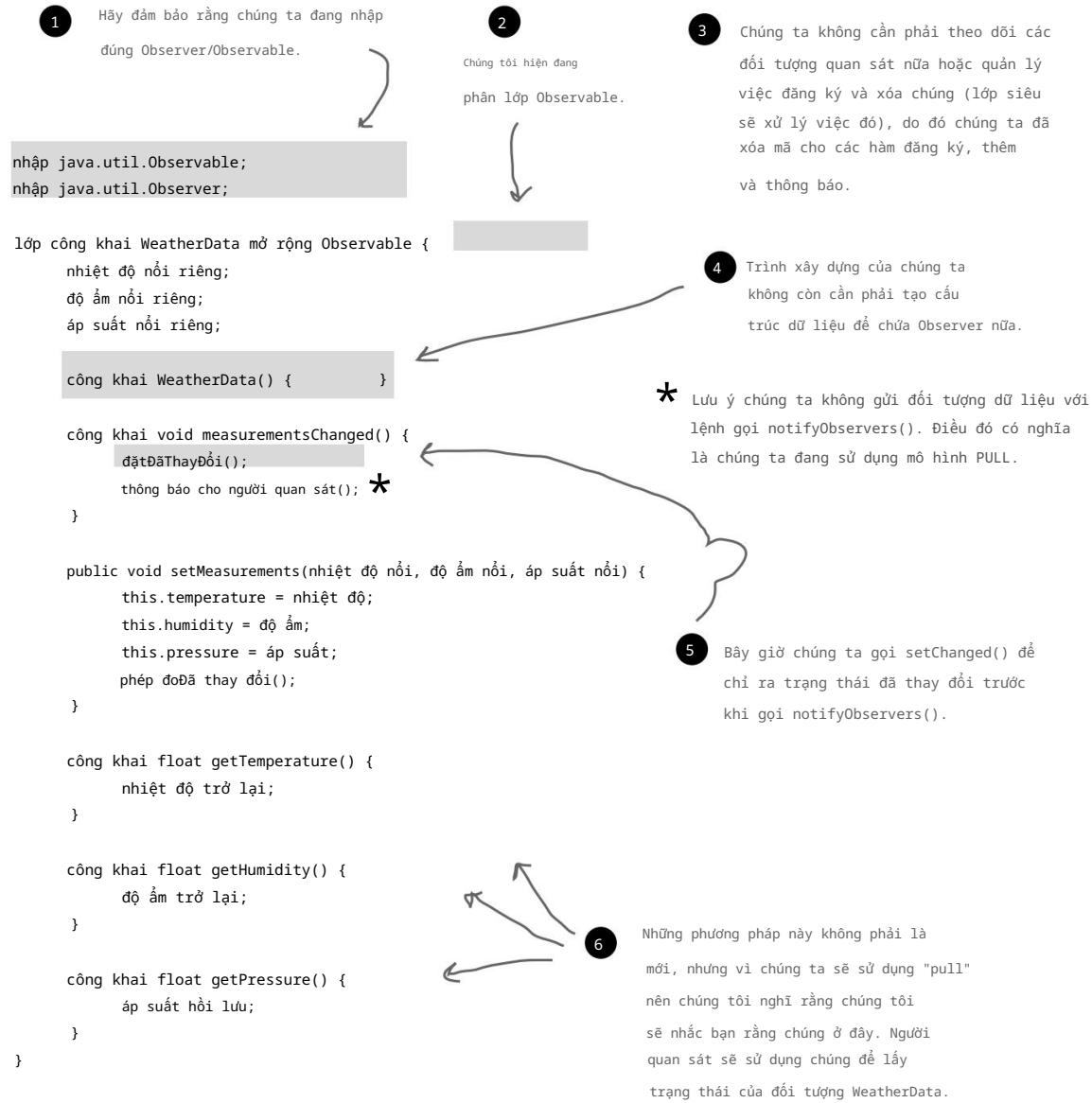


Tại sao điều này lại cần thiết? Phương thức `setChanged()` có mục đích cung cấp cho bạn nhiều sự linh hoạt hơn trong cách bạn cập nhật các quan sát viên bằng cách cho phép bạn tối ưu hóa các thông báo. Ví dụ, trong trạm thời tiết của chúng ta, hãy tưởng tượng nếu các phép đo của chúng ta quá nhạy cảm đến mức các số liệu nhiệt độ liên tục dao động vài phần mười độ. Điều đó có thể khiến đối tượng `WeatherData` liên tục gửi thông báo. Thay vào đó, chúng ta có thể muốn chỉ gửi thông báo nếu nhiệt độ thay đổi hơn nửa độ và chúng ta chỉ có thể gọi `setChanged()` sau khi điều đó xảy ra.

Bạn có thể không sử dụng chức năng này thường xuyên, nhưng nó sẽ có nếu bạn cần. Trong cả hai trường hợp, bạn cần gọi `setChanged()` để thông báo hoạt động. Nếu chức năng này hữu ích với bạn, bạn cũng có thể muốn sử dụng phương thức `clearChanged()`, phương thức này sẽ đặt lại trạng thái `đã thay đổi` thành `false` và phương thức `hasChanged()`, phương thức này sẽ cho bạn biết trạng thái hiện tại của cờ `đã thay đổi`.

Làm lại Trạm thời tiết với sự hỗ trợ tích hợp

Đầu tiên, chúng ta hãy làm lại WeatherData để sử dụng `java.util.Observable`



điều kiện hiện tại làm lại

Bây giờ, chúng ta hãy làm lại CurrentConditionsDisplay

- 1 Một lần nữa, hãy đảm bảo rằng chúng ta đang nhập đúng Observer/Observable.

```
nhập java.util.Observable;
nhập java.util.Observer;

lớp công khai CurrentConditionsDisplay triết khai Observer, DisplayElement {
    Có thể quan sát được;
    nhiệt độ nổi riêng;
    độ ẩm nổi riêng;

    public CurrentConditionsDisplay(Có thể quan sát được) {
        this.observable = có thể quan sát được;
        observable.addObserver(cái này);
    }

    public void update(Observable obs, Object arg) {
        nếu (obs instanceof WeatherData) {
            Dữ liệu thời tiết weatherData = (WeatherData)obs;
            this.temperature = weatherData.getTemperature();
            this.humidity = weatherData.getHumidity();
            trưng bày();
        }
    }
}

công khai void display() {
    System.out.println("Điều kiện hiện tại: " + nhiệt độ + độ ẩm + "% độ ẩm");
    + "Độ F và "
}
```

- 2 Bây giờ chúng ta đang triển khai giao diện Observer từ java.util.

- 3 Bây giờ, hàm tạo của chúng ta sẽ lấy một Observable và sử dụng nó để thêm đối tượng điều kiện hiện tại làm Observer.

- 4 Chúng tôi đã thay đổi phương thức update() để lấy cả Observable và đối số dữ liệu tùy chọn.

- 5 Trong update(), trước tiên chúng ta đảm bảo observable có kiểu là WeatherData và sau đó chúng ta sử dụng các phương thức getter của nó để lấy các phép đo nhiệt độ và độ ẩm. Sau đó chúng ta gọi display().



Mã Nam Châm

mẫu quan sát

Lớp ForecastDisplay bị xáo trộn trên tủ lạnh. Bạn có thể xây dựng lại các đoạn mã để làm cho nó hoạt động không? Một số dấu ngoặc nhọn rơi xuống sàn và chúng quá nhỏ để nhận, vì vậy hãy thoải mái thêm nhiều dấu ngoặc nhọn như bạn cần!

```
công khai ForecastDisplay(Observable  
    có thể quan sát được) {  
  
    observable.addObserver(cái này);  
  
    //...  
}  
  
trưng bày();
```

nếu (thể hiện quan sát được của WeatherData) {

lớp công khai ForecastDisplay thực hiện
Người quan sát, DisplayElement {

```
công khai void display() {  
    // hiển thị mã ở đây  
}  
  
    +pressure;  
    +.getPressure()  
currentPressure; currentPressure = weatherData.getPressure();
```

~~lastPressure = currentPressure; currentPressure = weatherData.getPressure();~~

private float CurrentAngle; // current angle

Dữ liệu thời tiết Dữ liệu thời tiết =
(WeatherData) có thể quan sát được;

```
public void update(Có thẻ quan sát được,  
Đối tượng arg) {
```

```
nhập java.util.Observable;  
nhập java.util.Observer;
```

lái thử

Chạy mã mới

Để chắc chắn hơn, chúng ta hãy chạy mã mới...

```
Cửa sổ chỉnh sửa tệp Trợ giúp TryThisAtHome

%java Trạm thời tiết
Dự báo: Thời tiết sẽ cải thiện!
Nhiệt độ trung bình/tối đa/tối thiểu = 80.0/80.0/80.0
Điều kiện hiện tại: 80,0F độ và độ ẩm 65,0%
Dự báo: Cần chú ý thời tiết mát mẻ và mưa
Nhiệt độ trung bình/tối đa/tối thiểu = 81,0/82,0/80,0
Điều kiện hiện tại: 82,0F độ và độ ẩm 70,0%
Dự báo: Vẫn như cũ
Nhiệt độ trung bình/tối đa/tối thiểu = 80,0/82,0/78,0
Điều kiện hiện tại: 78,0F độ và độ ẩm 90,0%
%
%
```

Hmm, bạn có nhận thấy điều gì khác biệt không? Hãy nhìn lại lần nữa...

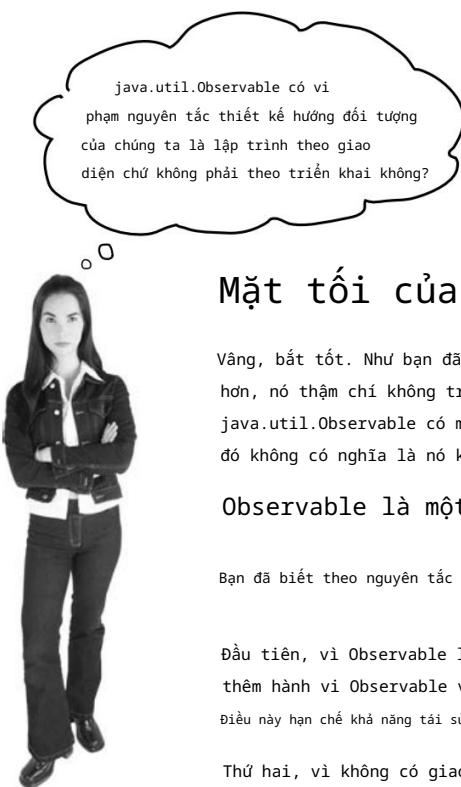
Bạn sẽ thấy tất cả các phép tính giống nhau, nhưng thật kỳ lạ, thứ tự đầu ra của văn bản lại khác. Tại sao điều này lại xảy ra? Hãy suy nghĩ một phút trước khi đọc tiếp...

**Không bao giờ phụ thuộc vào thứ tự đánh giá của
Thông báo của người quan sát**

java.util.Observable đã triển khai phương thức `notifyObservers()` sao cho các `Observer` được thông báo theo thứ tự khác với cách triển khai của chúng ta. Ai đúng?

Không cái nào cả; chúng tôi chỉ chọn cách thực hiện mọi thứ theo những cách khác nhau.

Tuy nhiên, điều không đúng là nếu chúng ta viết mã của mình phụ thuộc vào thứ tự thông báo cụ thể. Tại sao? Bởi vì nếu bạn cần thay đổi các triển khai `Observable/Observer`, thứ tự thông báo có thể thay đổi và ứng dụng của bạn sẽ tạo ra kết quả không chính xác. Böyle giờ, đó chắc chắn không phải là những gì chúng ta coi là kết hợp lồng léo.



Mặt tối của java.util.Observable

Vâng, bắt tốt. Như bạn đã thấy, Observable là một lớp, không phải là một giao diện, và tệ hơn, nó thậm chí không triển khai một giao diện. Thật không may, việc triển khai java.util.Observable có một số vấn đề hạn chế tính hữu ích và khả năng tái sử dụng của nó. Điều đó không có nghĩa là nó không cung cấp một số tiện ích, nhưng có một số ô gà lớn cần phải chú ý.

Observable là một lớp

Bạn đã biết theo nguyên tắc của chúng tôi đây là một ý tưởng tồi, nhưng nó thực sự gây hại như thế nào?

Đầu tiên, vì Observable là một lớp, bạn phải phân lớp nó. Điều đó có nghĩa là bạn không thể thêm hành vi Observable vào một lớp hiện có đã mở rộng một siêu lớp khác. Điều này hạn chế khả năng tái sử dụng của nó (và đó không phải là lý do tại sao chúng ta sử dụng các mẫu ngay từ đầu sao?).

Thứ hai, vì không có giao diện Observable, bạn thậm chí không thể tạo ra triển khai của riêng mình hoạt động tốt với API Observer tích hợp của Java. Bạn cũng không có tùy chọn hoán đổi triển khai java.util cho một triển khai khác (ví dụ, một triển khai đa luồng mới).

Observable bảo vệ các phương pháp quan trọng

Nếu bạn nhìn vào API Observable, phương thức `setChanged()` được bảo vệ. Vậy thì sao? Vâng, điều này có nghĩa là bạn không thể gọi `setChanged()` trừ khi bạn đã phân lớp Observable. Điều này có nghĩa là bạn thậm chí không thể tạo một thể hiện của lớp Observable và kết hợp nó với các đối tượng của riêng bạn, bạn phải phân lớp. Thiết kế vi phạm nguyên tắc thiết kế thứ hai ở đây. Điều này kết hợp hơn là kế thừa.

Phải làm gì?

Observable có thể đáp ứng nhu cầu của bạn nếu bạn có thể mở rộng `java.util.Observable`. Mặt khác, bạn có thể cần phải triển khai riêng như chúng tôi đã làm ở đầu chương. Trong cả hai trường hợp, bạn đều biết rõ về Observer Pattern và bạn có thể làm việc tốt với bất kỳ API nào sử dụng pattern này.

người quan sát và đú

Những nơi khác bạn sẽ tìm thấy Observer Pattern trong JDK

Triển khai java.util của Observer/Observable không phải là nơi duy nhất bạn sẽ tìm thấy Observer Pattern trong JDK; cả JavaBeans và Swing cũng cung cấp các triển khai riêng của mẫu này. Đến thời điểm này, bạn đã hiểu đủ về observer để tự mình khám phá các API này; tuy nhiên, chúng ta hãy làm một ví dụ Swing đơn giản, nhanh chóng chỉ để giải trí.

Một chút thông tin cơ bản...

Chúng ta hãy xem xét một phần đơn giản của Swing API, JButton. Nếu bạn nhìn vào siêu lớp của JButton, AbstractButton, bạn sẽ thấy rằng nó có rất nhiều add/remove listener methods. Các phương thức này cho phép bạn thêm và xóa observers, hay như được gọi trong Swing, listeners, để lắng nghe các loại sự kiện khác nhau xảy ra trên thành phần Swing. Ví dụ, ActionListener cho phép bạn "lắng nghe" bất kỳ loại hành động nào có thể xảy ra trên một nút, như nhấn nút. Bạn sẽ tìm thấy nhiều loại listeners khác nhau trên khắp Swing API.

Một ứng dụng nhỏ thay đổi cuộc sống

Được rồi, ứng dụng của chúng tôi khá đơn giản. Bạn có một nút có nội dung "Tôi có nên làm điều đó không?" và khi bạn nhấp vào nút đó, người nghe (người quan sát) có thể trả lời câu hỏi theo bất kỳ cách nào họ muốn. Chúng tôi đang triển khai hai trình lắng nghe như vậy, được gọi là AngelListener và DevilListener. Đây là cách ứng dụng hoạt động:

Nếu bạn tò mò về Observer Pattern trong JavaBeans, hãy xem qua giao diện PropertyChangeListener.



Và mǎ...

Ứng dụng thay đổi cuộc sống này cần rất ít mã. Tất cả những gì chúng ta cần làm là tạo một đối tượng JButton, thêm nó vào JFrame và thiết lập trình nghe của chúng ta. Chúng ta sẽ sử dụng các lớp bên trong cho trình nghe, đây là một kỹ thuật phổ biến trong lập trình Swing. Nếu bạn chưa biết về các lớp bên trong hoặc Swing, bạn có thể muốn xem lại chương "Getting GUI" của Head First Java.

```

        Úng dụng Swing đơn giản
lớp công khai SwingObserverExample {
    Khung JFrame;
    chỉ tạo một khung và
    thêm một nút vào đó.

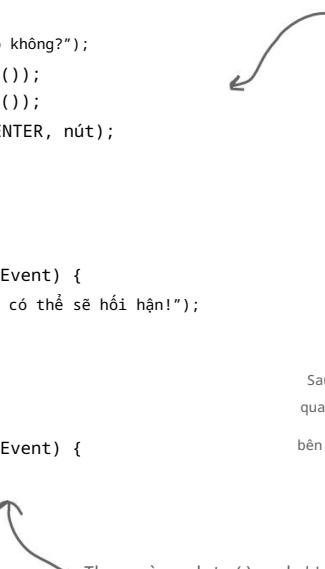
    public static void main(String[] args) {
        Ví dụ SwingObserverExample = new SwingObserverExample();
        ví dụ.go();
    }

    công khai void go() {
        khung = JFrame mới();
        Nút JButton = new JButton("Tôi có nên làm điều đó không?");
        button.addActionListener(AngelListener mới());
        button.addActionListener(new DevilListener());
        frame.getContentPane().add(BorderLayout.CENTER, nút);
        // Đặt thuộc tính khung ở đây
    }

    lớp AngelListener triển khai ActionListener {
        public void actionPerformed(sự kiện ActionEvent) {
            System.out.println("Đừng làm điều đó, bạn có thể sẽ hối hận!");
        }
    }

    lớp DevilListener thực hiện ActionListener {
        public void actionPerformed(sự kiện ActionEvent) {
            System.out.println("Nào, làm đi!");
        }
    }
}

```

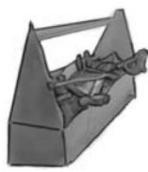


Biến các vật thể thiên
thần và ác quỷ thành
người nghe (người quan sát) của nút.

Sau đây là định nghĩa lớp cho các trình
quan sát, được định nghĩa là các lớp
bên trong (nhưng không nhất thiết phải như vậy).

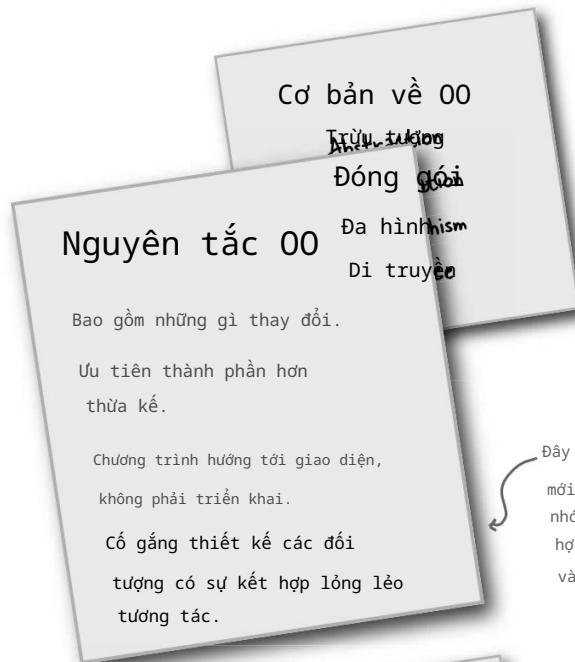
Thay vì update(), phương
thức actionPerformed()
được gọi khi trạng thái
trong chủ thẻ (trong
trường hợp này là nút) thay đổi.

hộp công cụ thiết kế của bạn



Công cụ cho hộp công cụ thiết kế của bạn

Chào mừng đến với phần cuối của Chương 2.
Bạn đã thêm một vài thứ mới vào hộp công cụ 00 của
mình...



Māu Patterns

Strategy - định nghĩa một họ thuật toán,
Observer - định nghĩa một-nhiều đóng gói từng
tùng thuẫn, tuân theo cách đóng gói chung
cho nhau. **Strategy** cho phép thuật toán khanh dải
đối tượng thay đổi trạng thái, tất cả các đối
tượng khang bia. **Observer** là một cách để khai thác cách dùng báo cáo để sử dụng nó.
người phụ thuộc được thông báo và cập nhật

Một mẫu mới để truyền đạt trạng thái đến một tập hợp các đối tượng theo cách liên kết lỏng lẻo. Chúng ta vẫn chưa thấy mẫu Observer cuối cùng - hãy đợi cho đến khi chúng ta nói về MVC.



— ĐIỂM ĐẦU TIÊN

Mẫu Observer định nghĩa mối quan hệ
một-nhiều giữa các đối tượng.

⇒ Chủ thể, hay chúng ta còn gọi là
Đối tượng quan sát, cập nhật
Đối tượng quan sát bằng một
giao diện chung.

- β Người quan sát được kết nối lồng lèo ở chỗ Observable không biết gì về họ, ngoại trừ việc họ triển khai Giao diện người quan sát.

β Bạn có thể đẩy hoặc kéo dữ liệu từ Observable khi sử dụng mảng (kéo được coi là “đúng” hơn).

B Ðừng phụ thuộc vào thứ tự thông báo cụ thể cho Người quan sát của bạn.

β Java có một số triển khai của Observer bao gồm cả `java.util.Observer` và `java.util.Observable`.

B Hãy chú ý đến các vấn đề liên quan đến việc triển khai Java.util.Observable

Đừng ngại tạo ra triển khai
Observable của riêng bạn

⇒ Swing sử dụng rất nhiều Observer Pattern, giống như nhiều khuôn khổ GUIT khác

Bạn cũng sẽ tìm thấy mô hình này ở nhiều nơi khác, bao gồm JavaBeans và RMT



Bài tập



mẫu quan sát

Thách thức về nguyên lý thiết kế

Đối với mỗi nguyên tắc thiết kế, hãy mô tả cách Observer Pattern sử dụng nguyên tắc đó.

Nguyên lý thiết kế

Xác định những khía cạnh khác biệt của ứng dụng và tách chúng ra khỏi những khía cạnh không thay đổi.

Nguyên lý thiết kế

Chương trình cho một giao diện, không phải là một triển khai.

Nguyên lý thiết kế

Ưu tiên thành phần hơn là thừa kế.

Đây là một câu hỏi khó, gợi ý: hãy nghĩ về cách người quan sát

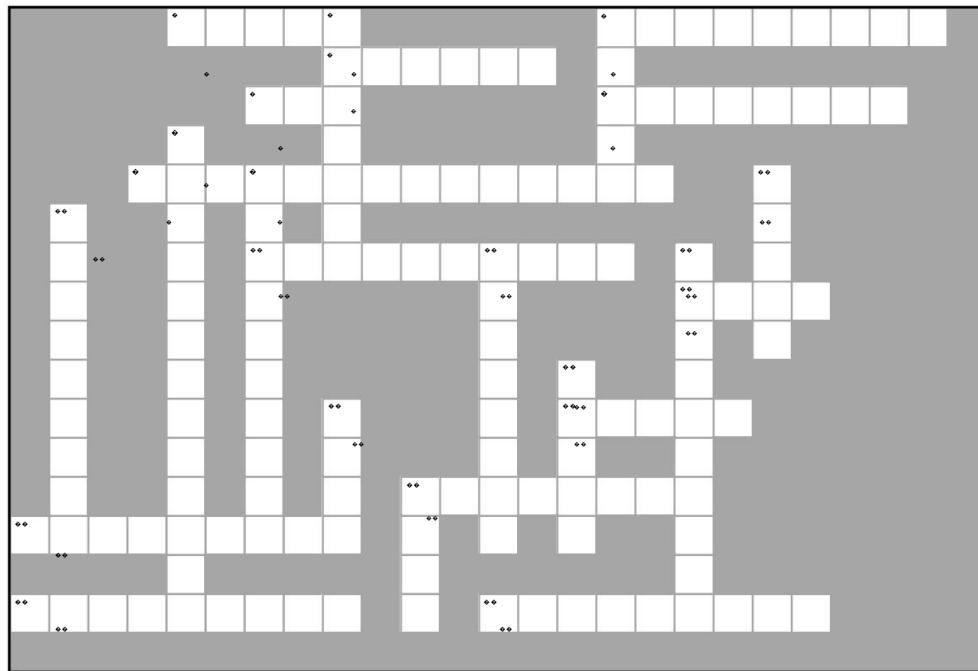
và các chủ thể làm việc cùng nhau.

trò chơi ô chữ



Đã đến lúc cho não phải của bạn hoạt động trở lại

Lần này tất cả các từ giải đều nằm trong chương 2



mẫu quan sát



Giải pháp bài tập

Chuốt bút chì của bạn

Dựa trên lần triển khai đầu tiên, điều nào sau đây đúng?
(Chọn tất cả những câu trả lời đúng.)

A. Chúng tôi đang mã hóa theo các triển khai cụ thể, không phải theo giao diện.
 B. Đối với mỗi phần tử hiển thị mới, chúng ta cần thay đổi mã.
 C. Chúng tôi không có cách nào để thêm các phần tử hiển thị trong thời gian chạy.
 D. Các thành phần hiển thị không triển khai giao diện chung.
 E. Chúng tôi chưa gói gọn những thay đổi.
 F. Chúng tôi đang vi phạm việc đóng gói của Lớp WeatherData.



Thiết kế Nguyên tắc Thủ thách

Nguyên lý thiết kế

Xác định những khía cạnh khác biệt của ứng dụng và tách chúng ra khỏi những khía cạnh không thay đổi.

Điều thay đổi trong Observer Pattern là trạng thái của

Subject và số lượng và loại Observer. Với pattern này, bạn

có thể thay đổi các đối tượng phụ thuộc vào trạng thái của

Subject mà không cần phải thay đổi Subject đó. Đó gọi là lập kế

hoạch trước!

Cả Chủ thể và Người quan sát đều sử dụng giao diện.

Subject theo dõi các đối tượng triển khai giao diện

Observer, trong khi các observer đăng ký với và nhận thông

báo từ giao diện Subject. Như chúng ta đã thấy, điều này

giúp mọi thứ trở nên tốt đẹp và được kết nối lỏng lẻo.

Mẫu Người quan sát sử dụng thành phần để kết hợp bất kỳ số

lượng Người quan sát nào với Chủ thể của họ.

Những mối quan hệ này không được thiết lập theo một loại phân

cấp kể từ nào đó. Không, chúng được thiết lập tại thời

điểm chạy bằng cách biên soạn!

Nguyên lý thiết kế

Chương trình cho một giao diện, không phải là một triển khai.

Nguyên lý thiết kế

Ưu tiên thành phần hơn là thừa kế.

giải bài tập



Mã Nam Châm

```

nhập java.util.Observable;
nhập java.util.Observer;

lớp công khai ForecastDisplay thực hiện
Người quan sát, DisplayElement {

    phao riêng hiện tại Áp suất = 29,92f;
    phao riêng cuối cùng Áp suất;

    công khai ForecastDisplay(Đó thể quan sát được)
    {

        observable.addObserver(cái này);
    }

    public void update(Observable observable,
        Đối tượng arg) {

        nếu (thể hiện quan sát được của WeatherData) {

            Dữ liệu thời tiết Dữ liệu thời tiết =
            (WeatherData) có thể quan sát được;

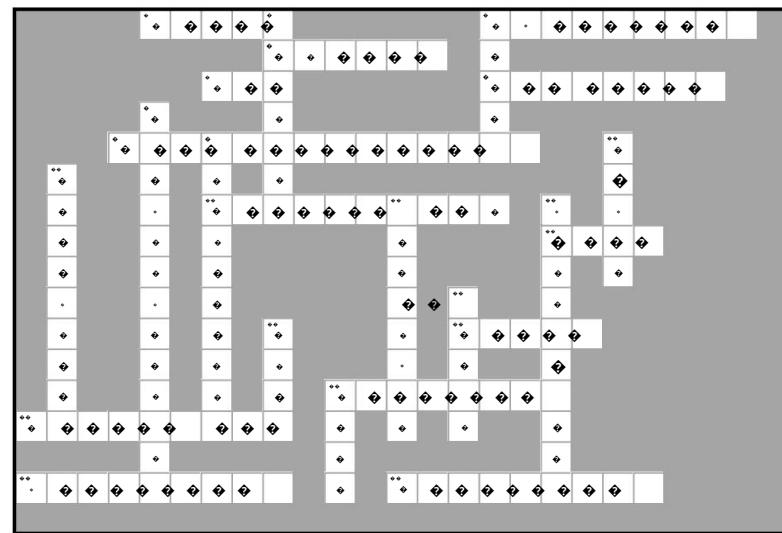
            lastPressure = currentPressure;
            currentPressure = weatherData.getPressure();
            // hiển thị bảng();

            công khai void display() {
                // hiển thị mã ở đây
            }
        }
    }
}

```



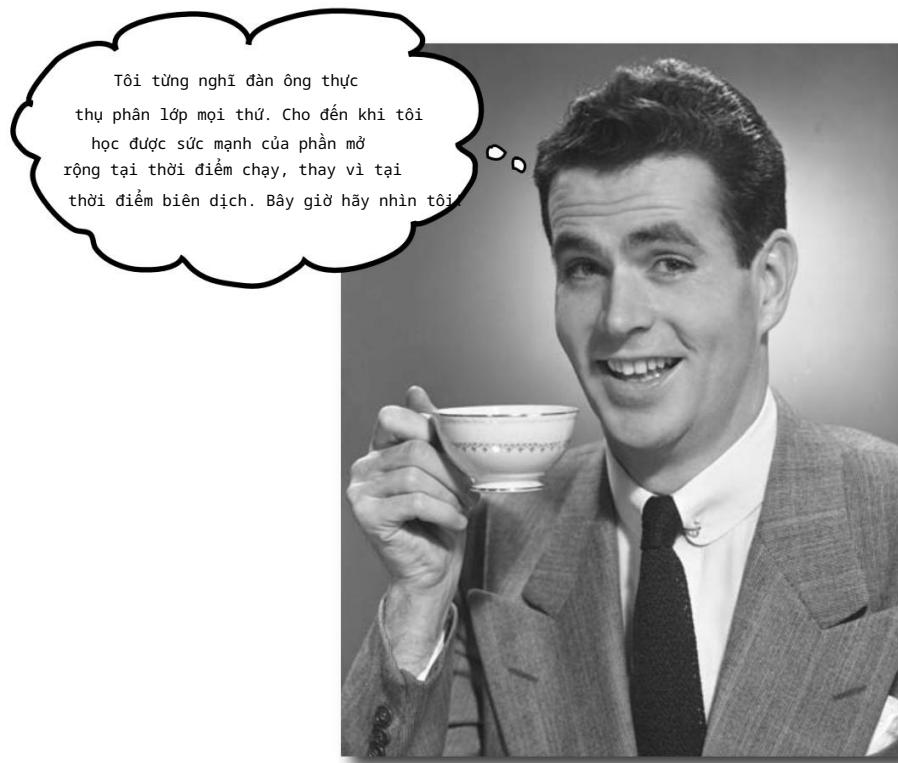
Giải pháp bài tập



3 mău trang trí

h

g Trang trí Đối tượng g



Hãy gọi chương này là “Con mắt thiết kế dành cho người thừa kế”.

Chúng ta sẽ xem xét lại việc sử dụng thừa kế quá mức thông thường và bạn sẽ học cách trang trí các lớp của mình khi chạy bằng một dạng thành phần đối tượng. Tại sao? Khi bạn biết các kỹ thuật trang trí, bạn sẽ có thể cung cấp cho các đối tượng của mình (hoặc của người khác) các trách nhiệm mới mà không cần thực hiện bất kỳ thay đổi mã nào đối với các lớp cơ bản.

câu chuyện starbuzz

Chào mừng đến với Starbuzz Coffee

Starbuzz Coffee đã tạo dựng được tên tuổi là quán cà phê phát triển nhanh nhất.

Nếu bạn thấy một quán ở góc phố địa phương, hãy nhìn sang bên kia đường; bạn sẽ thấy một quán khác.

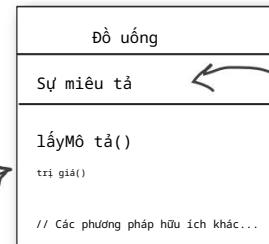


Vì phát triển quá nhanh nên họ đang phải vội vã cập nhật hệ thống đặt hàng của mình để phù hợp với các loại đồ uống họ cung cấp.

Khi họ mới bắt đầu kinh doanh, họ đã thiết kế lớp học như thế này...

Đồ uống là một lớp trừu tượng, được phân lớp thành tất cả các loại đồ uống được cung cấp trong quán cà phê.

Phương thức cost() là trừu tượng; các lớp con cần phải xác định cách triển khai riêng của chúng.



Biến thể hiện mô tả được đặt trong mỗi lớp con và chứa mô tả về đồ uống, ví dụ như "Rang đậm tuyệt hảo nhất".

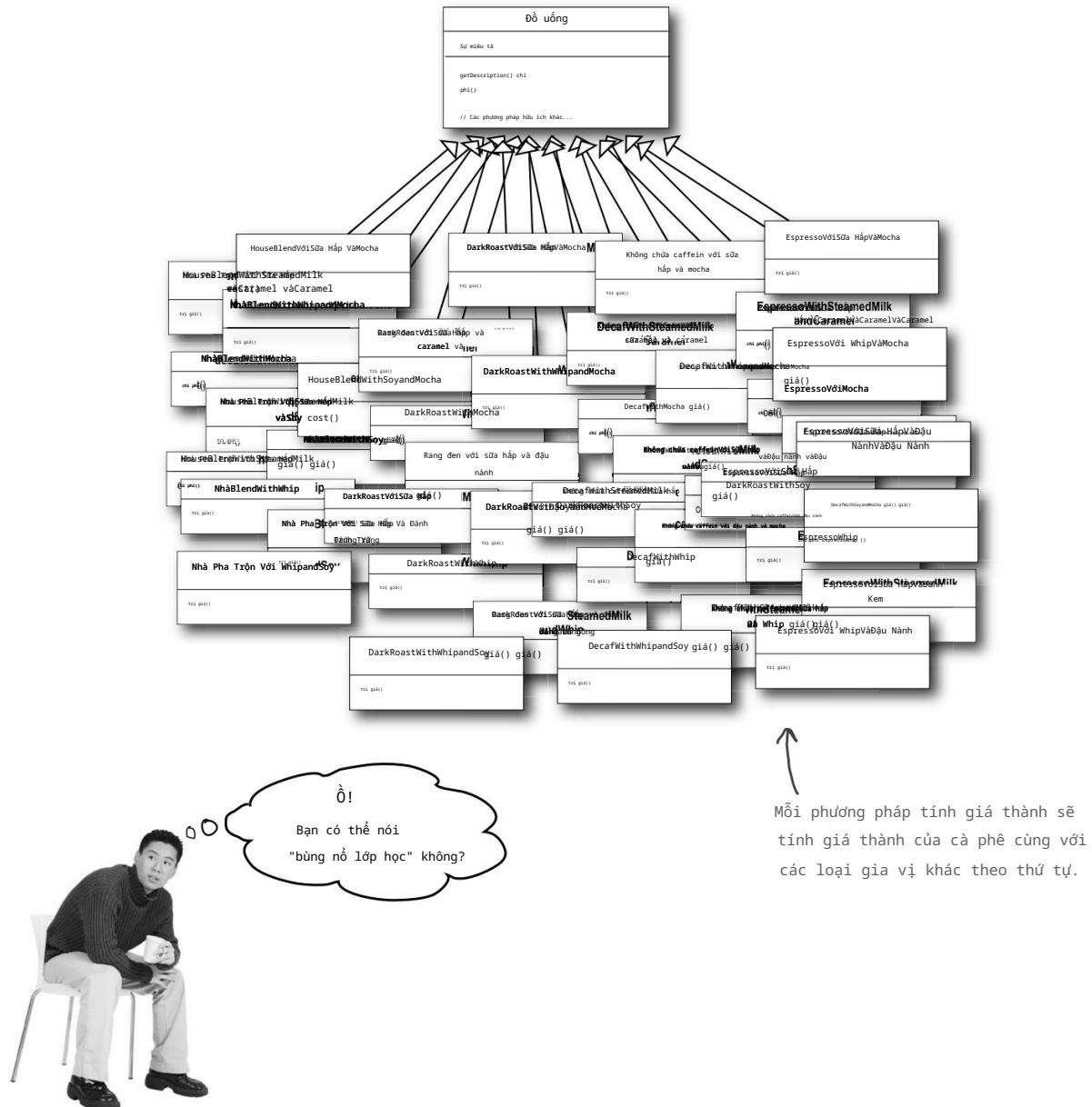
Phương thức getDescription() trả về mô tả.

Mỗi lớp con triển khai cost() để trả về chi phí của đồ uống.

mẫu trang trí

Ngoài cà phê, bạn cũng có thể yêu cầu một số loại gia vị như sữa hấp, đậu nành và mocha (hay còn gọi là sô cô la), và phủ lên trên tất cả bằng sữa đánh bông. Starbuzz tính một chút phí cho mỗi loại này, vì vậy họ thực sự cần đưa chúng vào hệ thống đặt hàng của mình.

Đây là nỗ lực đầu tiên của họ...



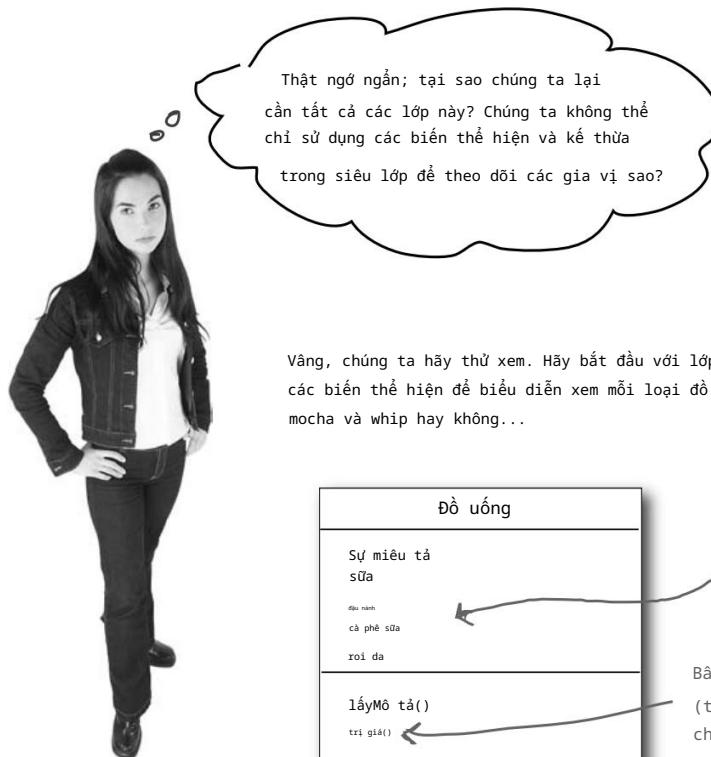
vi phạm nguyên tắc thiết kế

não Apower

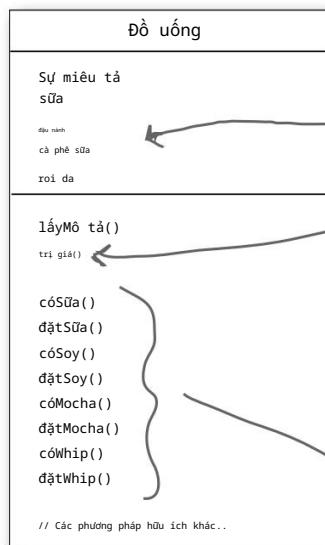
Rõ ràng là Starbuzz đã tự tạo ra cơn ác mộng về bảo trì cho chính họ. Điều gì sẽ xảy ra khi giá sữa tăng? Họ sẽ làm gì khi thêm lớp phủ caramel mới?

Nghĩ xa hơn vấn đề bảo trì, họ đang vi phạm nguyên tắc thiết kế nào mà chúng ta đã đề cập cho đến nay?

Gợi ý: họ đang vi phạm nghiêm trọng hai điều



Vâng, chúng ta hãy thử xem. Hãy bắt đầu với lớp cơ sở Beverage và thêm các biến thể hiện để biểu diễn xem mỗi loại đồ uống có sữa, đậu nành, mocha và whip hay không...

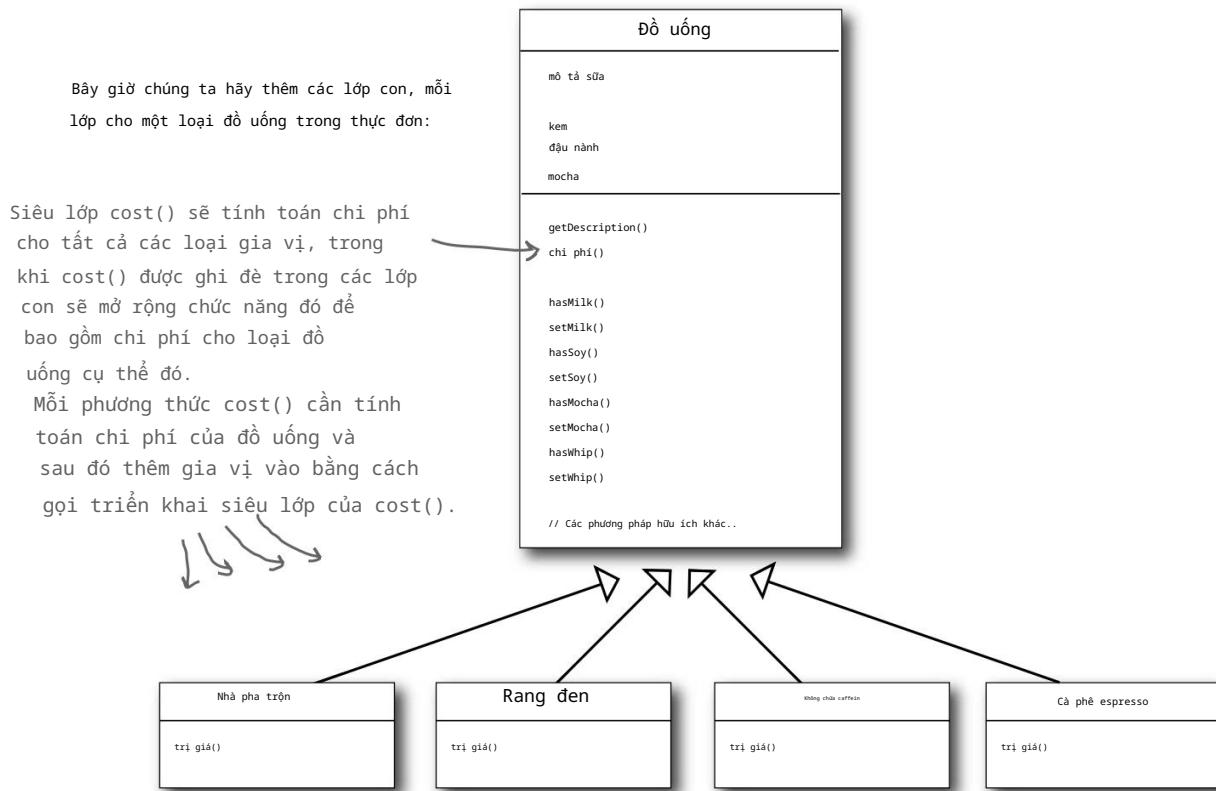


Giá trị boolean mới cho mỗi loại gia vị.

Bây giờ chúng ta sẽ triển khai cost() trong Beverage (thay vì giữ nó trống), để nó có thể tính toán chi phí liên quan đến gia vị cho một thể hiện đồ uống cụ thể. Các lớp con vẫn sẽ ghi đè cost(), nhưng chúng cũng sẽ gọi phiên bản siêu để chúng có thể tính toán tổng chi phí của đồ uống cơ bản cộng với chi phí của các gia vị được thêm vào.

Chúng lấy và đặt các giá trị boolean cho gia vị.

mẫu trang trí



Chuốt bút chì của bạn

Viết phương thức cost() cho các lớp sau (có thể sử dụng giả Java):

```

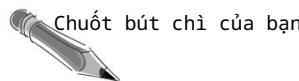
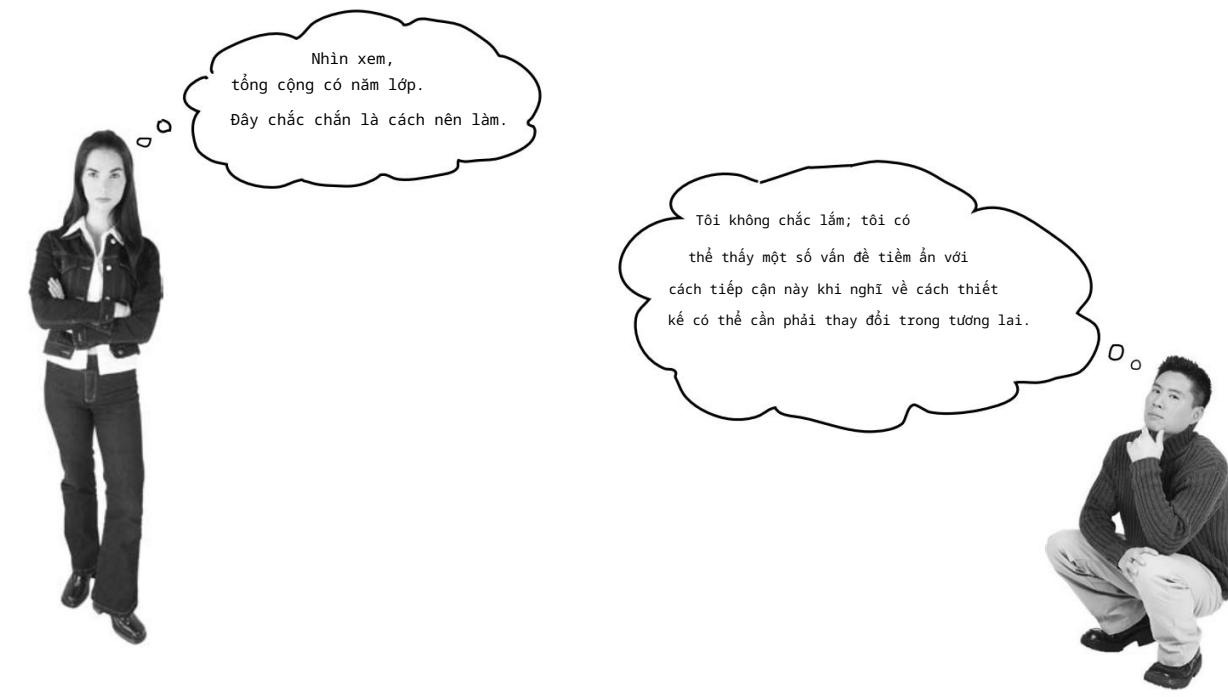
lớp công khai Beverage
{ public double cost() {
    ...
}

lớp công khai DarkRoast mở rộng Beverage {
    công khai DarkRoast() {
        mô tả = "Rang đậm tuyệt hảo nhất";
    }

    công khai double cost() {
        ...
    }
}

```

tác động của sự thay đổi



Những yêu cầu hoặc yếu tố nào khác có thể thay đổi và tác động đến thiết kế này?

Vì việc thay đổi giá cả sẽ buộc chúng tôi phải thay đổi quy định hiện hành.

Các gia vị mới sẽ buộc ta phải thêm các phương thức mới và thay đổi phương thức chi phí trong siêu lớp.

Chúng ta có thể có đồ uống mới. Đổi với một trong số những đồ uống này (trà đá?), các loại gia vị
số người có thể không phù hợp, nhưng lớp con Tea vẫn sẽ kế thừa các phương thức như hasWhip().

Như chúng ta đã thấy ở Chương

1, đây là một ý tưởng rất tệ!

Nếu khách hàng muốn uống double mocha thì sao?

Đến lượt bạn:



Thầy và trò...

Sư phụ: Châu chấu, đã lâu rồi chúng ta chưa gặp nhau. Con đã thiền định sâu sắc về di truyền chưa?

Học viên: Vâng, thưa Thầy. Mặc dù thưa kệ là mạnh mẽ, nhưng tôi đã học được rằng nó không phải lúc nào cũng dẫn đến sự linh hoạt nhất hoặc thiết kế có thể bảo trì được.

Sư phụ: À đúng rồi, con đã có tiến bộ. Vậy, hãy cho ta biết học trò của ta, vậy thì con sẽ đạt được sự tái sử dụng như thế nào nếu không thông qua sự kệ thưa?

Học viên: Thưa thầy, em đã học được rằng có nhiều cách để "thưa hướng" hành vi khi chạy thông qua thành phần và ủy quyền.

Sư phụ: Xin hãy tiếp tục...

Học viên: Khi tôi kệ thưa hành vi bằng cách phân lớp, hành vi đó được thiết lập tĩnh tại thời điểm biến dịch. Ngoài ra, tất cả các lớp con phải kệ thưa cùng một hành vi. Tuy nhiên, nếu tôi có thể mở rộng hành vi của đối tượng thông qua thành phần, thì tôi có thể thực hiện điều này một cách động tại thời điểm chạy.

Thầy: Tốt lắm, Châu Chấu, con bắt đầu thấy được sức mạnh của bồ cục rồi đấy.

Học viên: Có, tôi có thể thêm nhiều trách nhiệm mới vào các đối tượng thông qua kỹ thuật này, bao gồm cả những trách nhiệm mà ngay cả người thiết kế siêu lớp cũng không nghĩ đến. Và tôi không cần phải động đến mã của họ!

Thầy: Bạn đã học được gì về tác động của việc soạn thảo đối với việc duy trì mã của mình?

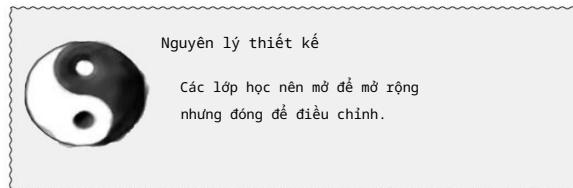
Học viên: Vâng, đó chính là điều tôi muốn nói. Bằng cách kết hợp các đối tượng một cách động, tôi có thể thêm chức năng mới bằng cách viết mã mới thay vì thay đổi mã hiện có. Vì tôi không thay đổi mã hiện có nên khả năng đưa lỗi hoặc gây ra các tác dụng phụ không mong muốn vào mã hiện có sẽ giảm đi nhiều.

Sư phụ: Tốt lắm. Hôm nay thế là đủ rồi, Grasshopper. Tôi muốn bạn đi và thiền thêm về chủ đề này... Hãy nhớ rằng, mã phải đóng (để thay đổi) như hoa sen vào buổi tối, nhưng mở (để mở rộng) như hoa sen vào buổi sáng.

nguyên lý mở-đóng

Nguyên lý Mở-Đóng

Grasshopper đã áp dụng một trong những nguyên tắc thiết kế quan trọng nhất:



Hãy đến; chúng tôi đang
mở cửa. Hãy thoải mái mở rộng
các lớp học của chúng tôi với bất kỳ hành vi mới
nào bạn thích. Nếu nhu cầu hoặc yêu cầu của bạn thay đổi (và chúng
tôi biết là sẽ thay đổi), hãy tiếp tục và tự tạo phần mở rộng
của riêng bạn.



Xin lỗi, chúng tôi đã đóng cửa.
Đúng vậy, chúng tôi đã
dành nhiều thời gian để sửa lỗi và đảm bảo mã này
chính xác, vì vậy chúng tôi không thể để bạn thay đổi mã hiện có.
Nó phải đóng để sửa lỗi. Nếu bạn không thích, bạn có thể nói
chuyện với người quản lý.

Mục tiêu của chúng tôi là cho phép mở rộng các lớp một cách dễ
dàng để kết hợp hành vi mới mà không cần sửa đổi mã hiện có.
Chúng ta sẽ nhận được gì nếu thực hiện được điều này? Thiết kế có khả
năng phục hồi khi thay đổi và đủ linh hoạt để đảm nhận chức năng
mới nhằm đáp ứng các yêu cầu thay đổi.

không có Những câu hỏi ngắn

Q: Mở rộng và đóng
để sửa đổi? Nghe có vẻ rất mâu thuẫn. Làm sao
một thiết kế có thể là cả hai?

A: Đó là một câu hỏi rất hay. Nó
Throat nghe có vẻ mâu thuẫn.
Rốt cuộc, thứ gì càng khó sửa đổi thì càng khó mở
rộng, đúng không?

Tuy nhiên, hóa ra có một số kỹ thuật OO thông
minh cho phép hệ thống được mở rộng, ngay
cả khi chúng ta không thể thay đổi mã cơ bản. Hãy
nghĩ về Observer Pattern (trong Chương 2)...
bằng cách thêm Observer mới, chúng ta có thể mở
rộng Subject bất kỳ lúc nào mà không cần thêm
mã vào Subject. Bạn sẽ thấy khá nhiều cách mở rộng
hành vi với các kỹ thuật thiết kế OO khác.

H: Được rồi, tôi hiểu rồi Observable,
nhưng làm sao tôi có thể thiết kế một
cái gì đó có thể mở rộng nhưng vẫn đóng để sửa đổi?

A: Nhiều mẫu cho chúng ta
thiết kế đã được kiểm tra theo thời gian bảo vệ
mã của bạn khỏi bị sửa đổi bằng cách cung cấp
phương tiện mở rộng. Trong chương này, bạn sẽ
thấy một ví dụ hay về việc sử dụng mẫu Decorator
để tuân theo nguyên tắc Open-Closed.

Q: Làm thế nào tôi có thể làm cho mọi bộ phận của
Thiết kế của tôi có tuân theo Nguyên tắc Mở-
Đóng không?

A: Thông thường, bạn không thể. Làm OO

thiết kế linh hoạt và mở rộng mà không cần sửa
đổi mã hiện tại tồn thời gian và công sức.
Nhìn chung, chúng ta không có đủ khả năng ràng
buộc mọi phần trong thiết kế của mình (và có lẽ
sẽ lãng phí). Việc tuân theo Nguyên tắc Mở-Đóng
thường đưa ra các cấp độ trừu tượng mới,
làm tăng thêm tính phức tạp cho mã của
chúng ta.

Bạn muốn tập trung vào những khu vực đó
có nhiều khả năng thay đổi trong thiết kế của
bạn và áp dụng các nguyên tắc ở đó.

Q: Làm sao tôi biết được khu vực nào của
thay đổi có quan trọng hơn không?

A: Đó một phần là vấn đề của
kinh nghiệm trong việc thiết kế các hệ thống OO và
cũng là vấn đề hiểu biết về lĩnh vực bạn đang
làm việc. Xem xét các ví dụ khác sẽ giúp bạn
học cách xác định các lĩnh vực cần thay đổi trong
thiết kế của riêng bạn.

**Mặc dù có vẻ mâu thuẫn, nhưng vẫn
có những kỹ thuật cho phép mở rộng mã
mà không cần sửa đổi trực tiếp.**

Hãy cẩn thận khi chọn các vùng mã cần
mở rộng; việc áp dụng Nguyên tắc Mở-
Đóng Ở MỌI NƠI là lãng phí, không
cần thiết và có thể dẫn đến mã phức
tạp, khó hiểu.

gặp gỡ người trang trí mẫu

Gặp gỡ mẫu trang trí

Được rồi, chúng ta đã thấy rằng việc biểu diễn sơ đồ định giá đồ uống công với gia vị bằng cách kế thừa không hiệu quả lắm - chúng ta nhận được các vụ nở lớp, thiết kế cứng nhắc hoặc chúng ta thêm chức năng vào lớp cơ sở không phù hợp với một số lớp con.

Vậy thì, đây là những gì chúng ta sẽ làm thay vào đó: chúng ta sẽ bắt đầu với một thức uống và "trang trí" nó bằng các loại gia vị khi chạy. Ví dụ, nếu khách hàng muốn Dark Roast với Mocha và Whip, thì chúng ta sẽ:

- 1 Lấy một đối tượng DarkRoast
- 2 Trang trí nó bằng một vật thể Mocha
- 3 Trang trí nó bằng một vật thể Whip
- 4 Gọi phương thức cost() và dựa vào phái đoàn để thêm vào chi phí gia vị

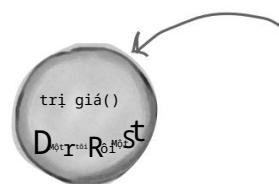
Được rồi, đủ rồi về "Câu lạc bộ thiết kế hương đối tượng". Chúng ta có vấn đề thực sự ở đây! Bạn còn nhớ chúng tôi không? Starbuzz Coffee? Bạn có nghĩ rằng bạn có thể sử dụng một số nguyên tắc thiết kế đó để thực sự giúp chúng tôi không?



Được thôi, nhưng làm thế nào để bạn "trang trí" một đối tượng và làm thế nào để ủy quyền tham gia vào việc này? Một gợi ý: hãy nghĩ về các đối tượng trang trí như "wrapper". Hãy xem cách thức hoạt động của nó...

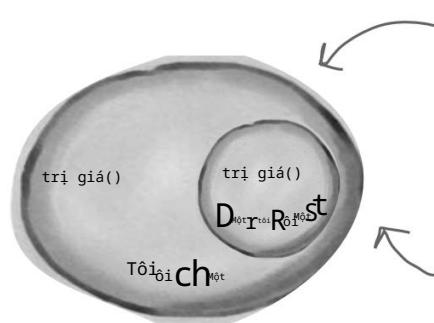
Xây dựng đơn đặt hàng đồ uống với Decorators

- 1 Chúng ta bắt đầu với đối tượng DarkRoast.



Hãy nhớ rằng DarkRoast kế thừa từ Beverage và có phương thức cost() để tính toán chi phí của đồ uống.

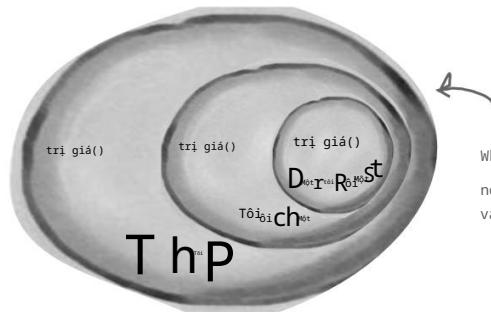
- 2 Khách hàng muốn dùng Mocha, vì vậy chúng tôi tạo một đối tượng Mocha và bao bọc nó xung quanh DarkRoast.



Đối tượng Mocha là một trình trang trí. Kiểu của nó phản ánh đối tượng mà nó đang trang trí, trong trường hợp này là một Đồ uống. (Bằng "gương", chúng tôi muốn nói đến cùng một kiểu..)

Vì vậy, Mocha cũng có phương thức cost() và thông qua đa hình, chúng ta có thể coi bất kỳ Beverage nào được gói trong Mocha cũng là một Beverage (vì Mocha là một kiểu con của Beverage).

- 3 Khách hàng cũng muốn có Whip, vì vậy chúng tôi tạo một trình trang trí Whip và bao bọc Mocha bằng trình trang trí này.

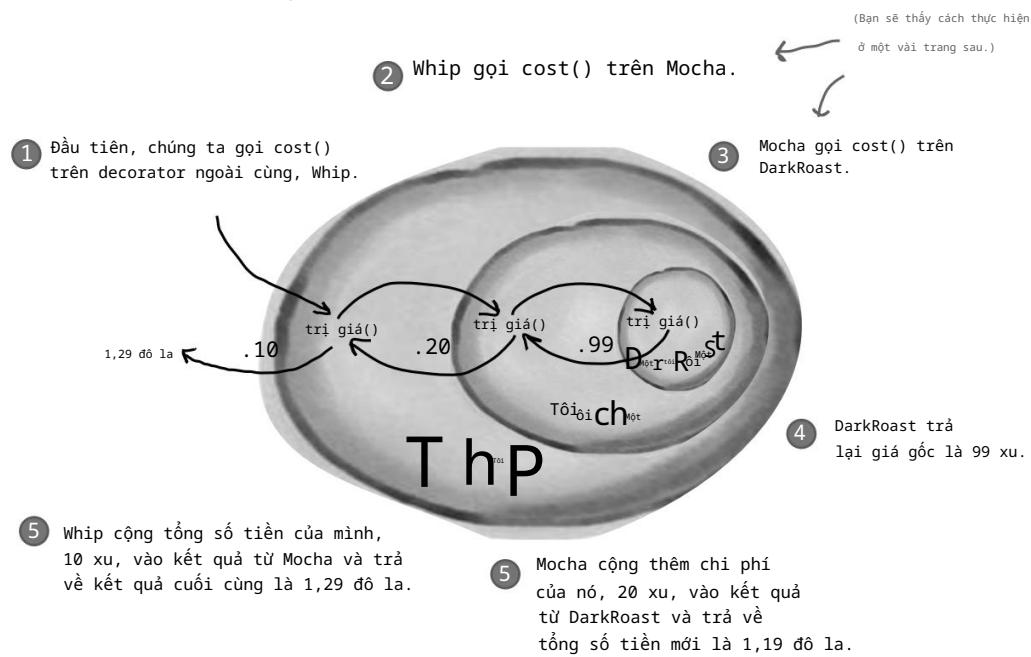


Whip là một trình trang trí, do đó nó cũng phản ánh kiểu của DarkRoast và bao gồm phương thức cost().

Vì vậy, DarkRoast được bao bọc trong Mocha và Whip vẫn là một Beverage và chúng ta có thể làm bất cứ điều gì với nó, bao gồm cả việc gọi phương thức cost() của nó.

đặc điểm trang trí

- 4 Bây giờ là lúc tính toán chi phí cho khách hàng. Chúng ta thực hiện điều này bằng cách gọi `cost()` trên decorator ngoài cùng, Whip, và Whip sẽ chuyển giao việc tính toán chi phí cho các đối tượng mà nó trang trí.
Khi nó có chi phí, nó sẽ cộng thêm chi phí của Whip.



Được rồi, đây là những gì chúng ta biết cho đến nay...

Þ Trình trang trí có cùng siêu kiểu với các đối tượng mà chúng trang trí.

Þ Bạn có thể sử dụng một hoặc nhiều trình trang trí để bao bọc một đối tượng.

Þ Vì trình trang trí có cùng siêu kiểu với đối tượng mà nó trang trí, nên chúng ta có thể truyền một đối tượng được trang trí thay cho đối tượng gốc (được gọi).

Trình trang trí thêm hành vi của riêng nó trước và/hoặc sau khi ủy quyền cho đối tượng mà nó trang trí để thực hiện phần công việc còn lại.

Điểm chính!

Þ Đối tượng có thể được trang trí bất cứ lúc nào, vì vậy chúng ta có thể trang trí đối tượng một cách động khi chạy với nhiều trình trang trí tùy thích.

Bây giờ chúng ta hãy xem cách thức hoạt động thực sự của nó bằng cách xem định nghĩa Mẫu trang trí và viết một số mã.

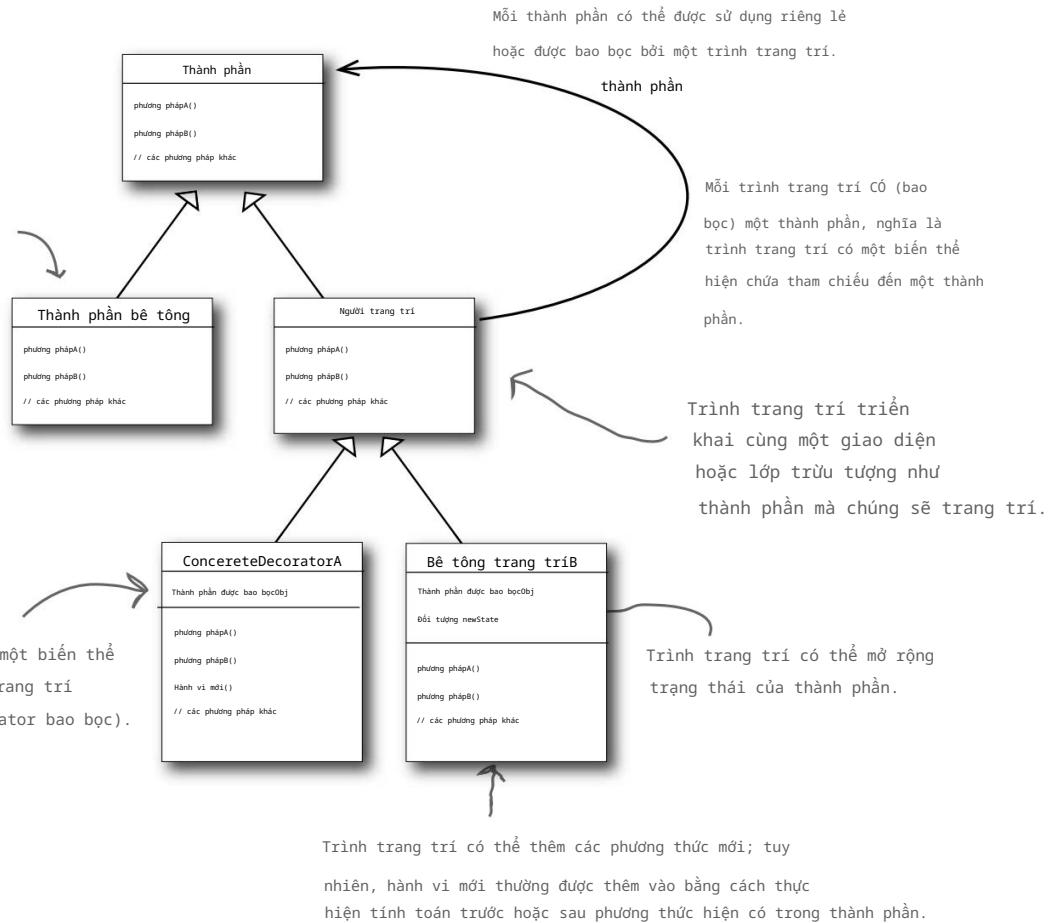
Mẫu trang trí được định nghĩa

Trước tiên chúng ta hãy xem qua mô tả về Mẫu trang trí:

Mẫu Decorator gắn thêm các trách nhiệm bổ sung cho một đối tượng một cách động.
Trình trang trí cung cấp một giải pháp thay thế linh hoạt cho việc phân lớp để mở rộng chức năng.

Mặc dù mô tả vai trò của Mẫu trang trí, nhưng nó không cung cấp cho chúng ta nhiều hiểu biết sâu sắc về cách chúng ta áp dụng mẫu vào triển khai của riêng mình. Hãy cùng xem sơ đồ lớp, có thể tiết lộ nhiều hơn một chút (ở trang tiếp theo, chúng ta sẽ xem cùng cấu trúc được áp dụng cho bài toán đồ uống).

ConcreteComponent là đối tượng mà chúng ta sẽ thêm hành vi mới một cách động. Nó mở rộng Component.



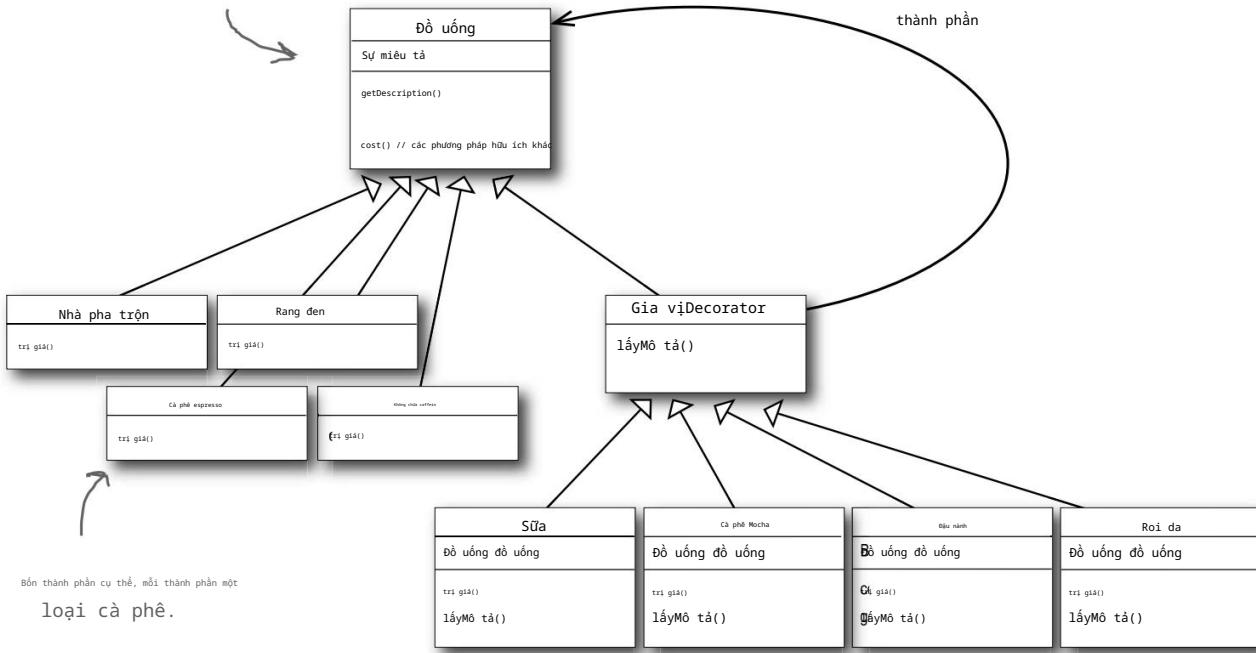
đồ uống trang trí

Trang trí đồ uống của chúng tôi

Được rồi, chúng ta hãy đưa đồ uống Starbuzz vào khuôn khổ này nhé...

Beverage đóng vai trò là lớp

thành phần trừu tượng của chúng ta.



Bốn thành phần cụ thể, mỗi thành phần một
loại cà phê.

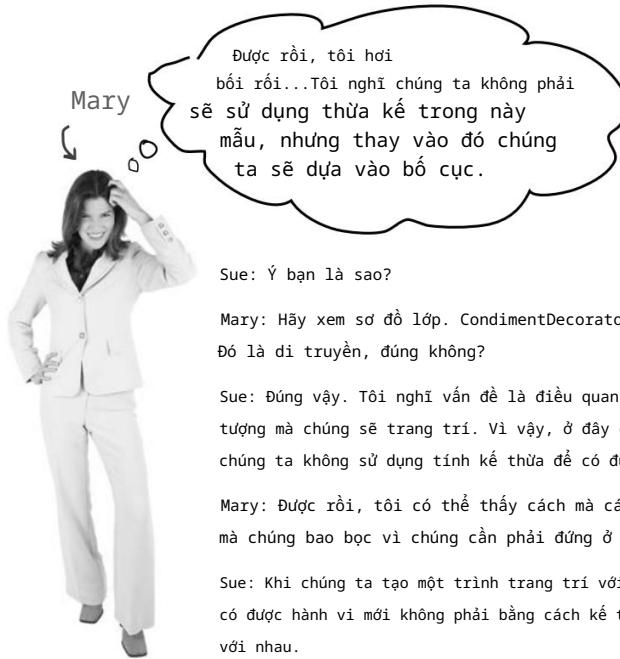
Và đây là các trình trang trí giả vị của chúng ta; lưu
ý rằng chúng cần triển khai không chỉ `cost()` mà còn
`getDescription()`. Chúng ta sẽ xem lý do tại sao sau đây...

não Apower

Trước khi đi xa hơn, hãy nghĩ về cách bạn sẽ triển khai phương thức `cost()`
của **coffee**s và **spicements**. Cũng hãy nghĩ về cách bạn sẽ triển khai phương
thức `getDescription()` của **spicements**.

Cuộc trò chuyện trong ô

Một số nhầm lẫn giữa Kế thừa và Thành phần



Sue: Ý bạn là sao?

Mary: Hãy xem sơ đồ lớp. CondimentDecorator đang mở rộng lớp Beverage.

Đó là di truyền, đúng không?

Sue: Đúng vậy. Tôi nghĩ vấn đề là điều quan trọng là các trình trang trí phải có cùng kiểu với các đối tượng mà chúng sẽ trang trí. Vì vậy, ở đây chúng ta sử dụng tính kế thừa để đạt được sự khớp kiểu, nhưng chúng ta không sử dụng tính kế thừa để có được hành vi.

Mary: Được rồi, tôi có thể thấy cách mà các trình trang trí cần cùng một "giao diện" như các thành phần mà chúng bao bọc vì chúng cần phải đứng ở vị trí của thành phần. Nhưng hành vi xuất hiện ở đâu?

Sue: Khi chúng ta tạo một trình trang trí với một thành phần, chúng ta đang thêm hành vi mới. Chúng ta đang có được hành vi mới không phải bằng cách kế thừa nó từ một lớp siêu, mà bằng cách kết hợp các đối tượng lại với nhau.

Mary: Được rồi, vì vậy chúng ta đang phân lớp trừu tượng Beverage để có đúng kiểu, không phải để kế thừa hành vi của nó. Hành vi xuất hiện thông qua việc kết hợp các trình trang trí với các thành phần cơ sở cũng như các trình trang trí khác.

Sue: Đúng vậy.

Mary: Ooooh, tôi hiểu rồi. Vì vì chúng ta đang sử dụng thành phần đối tượng, chúng ta có nhiều sự linh hoạt hơn về cách pha trộn và kết hợp gia vị và đồ uống. Rất mượt mà.

Sue: Đúng vậy, nếu chúng ta dựa vào kế thừa, thì hành vi của chúng ta chỉ có thể được xác định tĩnh tại thời điểm biên dịch. Nói cách khác, chúng ta chỉ nhận được bất kỳ hành vi nào mà siêu lớp cung cấp cho chúng ta hoặc chúng ta ghi đè. Với thành phần, chúng ta có thể kết hợp và kết hợp các trình trang trí theo bất kỳ cách nào chúng ta thích... tại thời điểm chạy.

Mary: Và theo tôi hiểu, chúng ta có thể triển khai các trình trang trí mới bất kỳ lúc nào để thêm hành vi mới.

Nếu chúng ta dựa vào kế thừa, chúng ta sẽ phải vào và thay đổi mã hiện có bất kỳ lúc nào chúng ta muốn có hành vi mới.

Sue: Chính xác.

Mary: Tôi chỉ có một câu hỏi nữa. Nếu tất cả những gì chúng ta cần kế thừa là kiểu của thành phần, tại sao chúng ta không sử dụng giao diện thay vì lớp trừu tượng cho lớp Beverage?

Sue: Vâng, hãy nhớ rằng khi chúng ta nhận được mã này, Starbuzz đã có một lớp Beverage trừu tượng.

Theo truyền thống, Decorator Pattern chỉ định một thành phần trừu tượng, nhưng trong Java, rõ ràng là chúng ta có thể sử dụng một giao diện. Nhưng chúng ta luôn cố gắng tránh thay đổi mã hiện có, vì vậy đừng "sửa" nó nếu lớp trừu tượng sẽ hoạt động tốt.

đào tạo trang trí

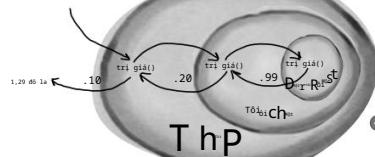
Đào tạo barista mới

Hãy vẽ một bức tranh về những gì xảy ra khi đơn hàng là đồ uống "double mocha soy latte with whip". Sử dụng menu để có giá chính xác và vẽ bức tranh của bạn bằng cùng định dạng mà chúng tôi đã sử dụng trước đó (từ một vài trang trước):

② Whip gọi cost() trên Mocha.

① Đầu tiên, chúng ta gọi cost()
trên decorator ngoài cùng, Whip.

③ Mocha gọi cost() trên
DarkRoast.



←

Bức ảnh này dành cho
một loại đồ uống "cà
phê mocha rang đậm".

④

DarkRoast trả
lại giá gốc là 99 xu.

⑤ Whip cộng tổng số tiền của mình,
.10 xu, vào kết quả từ Mocha và trả
về kết quả cuối cùng là 1,29 đ/la.

⑤ Mocha cộng thêm chi phí của
nó, .20 xu, vào kết quả từ
DarkRoast và trả về tổng số
tiền mới là 1,19 đ/la.

Chuốt bút chì của bạn

Vẽ hình ảnh của bạn ở đây.



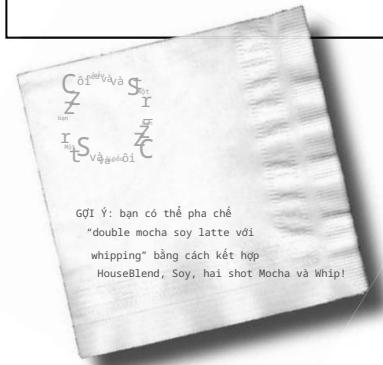
Cà phê Starbuzz

Cà phê

Nhà pha trộn	.89
Rang đậm	.99
Không chứa caffeine	1,05
Cà phê espresso	1,99

Gia vị

Sữa hắp	.10
Cà phê Mocha	.20
Đậu nành	.15
Roi da	.10



GỢI Ý: bạn có thể pha chế
"double mocha soy latte với
"whipping" bằng cách kết hợp
HouseBlend, Soy, hai shot Mocha và Whip!

Viết mã Starbuzz

Đã đến lúc biến thiết kế này thành mã thực sự.



Chúng ta hãy bắt đầu với lớp Beverage, lớp này cần phải thay đổi so với thiết kế ban đầu của Starbuzz. Chúng ta hãy cùng xem:

```

lớp trừu tượng công khai Đồ uống {
    mô tả chuỗi = "Đồ uống không xác định";
}

công khai String getDescription() {
    mô tả trả về;
}

tóm tắt công khai chi phí gấp đôi();
}

```

Beverage là một lớp trừu tượng có hai phương thức `getDescription()` và `cost()`.

`getDescription` đã được triển khai cho chúng ta, nhưng chúng ta cần triển khai `cost()` trong các lớp con.

Beverage khá đơn giản. Chúng ta hãy triển khai lớp trừu tượng cho Condiments (Decorator):

```

lớp trừu tượng công khai CondimentDecorator mở rộng Beverage {
    công khai trừu tượng String getDescription();
}

```

Đầu tiên, chúng ta cần có khả năng hoán đổi với Beverage, vì vậy chúng ta mở rộng lớp Beverage.

Chúng tôi cũng sẽ yêu cầu tất cả các trình trang trí già vì triển khai lại phương thức `getDescription()`. Một lần nữa, chúng ta sẽ xem lý do tại sao sau một giây...

thực hiện các đồ uống

Đồ uống mã hóa

Bây giờ chúng ta đã hoàn thành các lớp cơ sở, hãy triển khai một số đồ uống. Chúng ta sẽ bắt đầu với Espresso.

Hãy nhớ rằng, chúng ta cần thiết lập mô tả cho loại đồ uống cụ thể và cũng phải triển khai phương thức cost().

```
lớp công khai Espresso mở rộng Beverage {
```

```
    công khai Espresso() {
        mô tả = "Espresso";
    }

    công khai double cost() {
        trả về 1,99;
    }
}
```

Đầu tiên chúng ta mở rộng lớp Beverage vì đây là một loại đồ uống.

Để chăm sóc mô tả, chúng ta đặt điều này trong hàm tạo cho lớp. Hãy nhớ rằng biến thẻ hiện mô tả được kế thừa từ Beverage.

Cuối cùng, chúng ta cần tính giá của một ly Espresso. Chúng ta không cần phải lo lắng về việc thêm giá vị vào lớp này, chúng ta chỉ cần trả về giá của một ly Espresso: 1,99 đô la.

```
lớp công khai HouseBlend mở rộng Beverage {
    công khai HouseBlend() {
        mô tả = "Cà phê pha trộn tại nhà";
    }

    công khai double cost() {
        trả về .89;
    }
}
```

↑ Được rồi, đây là một loại đồ uống khác. Tất cả những gì chúng ta làm là thiết lập mô tả phù hợp, "House Blend Coffee", rồi trả về giá đúng: 89 xu.

Cà phê Starbuzz

<u>Cà phê</u>	.89
<u>Nhà pha trộn</u>	.99
<u>Rang đậm</u>	1,05
Không chứa caffeine	1,99
Cà phê espresso	
<u>Gia vị</u>	.10
<u>Sữa</u> hấp	.20
Cà phê Mocha	.15
Bột nành	.10
Roi da	

Bạn có thể tạo hai loại đồ uống khác (DarkRoast và Decaf) theo cách hoàn toàn tương tự.

Mã hóa gia vị

Nếu bạn nhìn lại sơ đồ lớp Decorator Pattern, bạn sẽ thấy chúng ta đã viết thành phần trầu tượng (Beverage), chúng ta có thành phần cụ thể (HouseBlend) và chúng ta có decorator trầu tượng (CondimentDecorator). Bây giờ là lúc triển khai các decorator cụ thể. Đây là Mocha:

```
Mocha là một trình trang trí, vì
vậy chúng ta mở rộng CondimentDecorator.

lớp công khai Mocha mở rộng CondimentDecorator {
    Đồ uống giải khát;

    công cộng Mocha (đồ uống giải khát) {
        this.beverage = đồ uống;
    }

    công khai String getDescription() {
        trả về đồ uống.getDescription() + ", Mocha";
    }

    công khai double cost() {
        trả về .20 + chi phí đồ uống();
    }
}

Bây giờ chúng ta cần tính toán chi phí cho đồ uống của
mình với Mocha. Đầu tiên, chúng ta chuyển lệnh gọi
đến đối tượng chúng ta đang trang trí, để nó có thể tính
tổng chi phí; sau đó, chúng ta thêm chi phí của Mocha vào kết quả.
```

Hãy nhớ rằng CondimentDecorator mở rộng Beverage.

Chúng ta sẽ khởi tạo Mocha bằng cách tham
chiều đến Beverage bằng cách sử dụng:

(1) Một biến thể hiện để chứa đồ uống mà
chúng ta đang gói.
(2) Một cách để thiết lập biến thể
hiện này cho đối tượng mà chúng
ta đang gói. Ở đây, chúng ta sẽ truyền đồ
uống mà chúng ta đang gói cho hàm tạo
của trình trang trí.

Chúng tôi muốn mô tả của mình không
chỉ bao gồm đồ uống - ví dụ "Dark
Roast" - mà còn bao gồm từng món
đồ trang trí đồ uống, ví dụ, "Dark
Roast, Mocha". Vì vậy, trước tiên
chúng tôi chuyển giao cho đối tượng mà
chúng tôi đang trang trí để có được mô
tả của nó, sau đó thêm ", Mocha" vào mô tả đó.

Ở trang tiếp theo, chúng ta sẽ thực sự tạo ra đồ uống và gói nó
bằng tất cả các loại gia vị (đồ trang trí), nhưng trước tiên...



Chuột bút chì của bạn

Viết và biên dịch mã cho các loại gia vị Soy and
Whip khác. Bạn sẽ cần chúng để hoàn thiện và kiểm tra ứng dụng.

thử nghiệm đồ uống

Phục vụ một số loại cà phê

Xin chúc mừng. Đã đến lúc ngồi lại, gọi vài tách cà phê và chiêm ngưỡng thiết kế linh hoạt mà bạn đã tạo ra bằng Mẫu trang trí.

Sau đây là một số mã kiểm tra để thực hiện lệnh:

```

lớp công khai StarbuzzCoffee {
    public static void main(String args[]) {
        Đồ uống đồ uống = new Espresso();
        System.out.println(beverage.getDescription() " $" +
            + beverage.cost());
        Đồ uống beverage2 = new DarkRoast();
        beverage2 = new Mocha(beverage2);
        beverage2 = new Mocha(beverage2);
        beverage2 = new Whip(beverage2);
        System.out.println(beverage2.getDescription() " $" +
            + beverage2.cost());
        Đồ uống beverage3 = new HouseBlend();
        beverage3 = new Soy(beverage3);
        beverage3 = new Mocha(beverage3);
        beverage3 = new Whip(beverage3);
        System.out.println(beverage3.getDescription() " $" +
            + beverage3.cost());
    }
}

```

Gọi một tách espresso, không có
gia vị và in mô tả và giá của tách.

Tạo đối tượng DarkRoast.

Gói nó bằng một chiếc Mocha.

Bọc nó trong một chiếc Mocha thứ hai.

Quấn nó trong một chiếc roi.

Cuối cùng, hãy pha chế
HouseBlend với Soy, Mocha và Whip.

* Chúng ta sẽ thấy một cách tốt hơn nhiều
để tạo ra các vật thể trang trí khi chúng ta
tìm hiểu về Mẫu nhà máy (và Mẫu xây dựng,
được đề cập trong phần phụ lục).

Bây giờ, chúng ta hãy thực hiện các lệnh sau:

```

Cửa sổ chỉnh sửa tệp Trợ giúp CloudsInMyCoffee
% java StarbuzzCoffee
Cà phê espresso $1.99
Cà phê rang đậm, Mocha, Mocha, Whip $1.49
Cà phê House Blend, đậu nành, Mocha, kem tươi $1.34
%

```

không có Những câu hỏi ngắn

H: Tôi hơi lo lắng về mã

có thể kiểm tra một thành phần bê tông cụ thể – chẳng hạn như HouseBlend – và thực hiện một số hành động, như đưa ra mức giảm giá. Sau khi tôi bọc HouseBlend bằng các trình trang trí, điều này sẽ không còn hiệu quả nữa.

A: Đúng vậy. Nếu bạn có

mã dựa vào kiểu của thành phần cụ thể, trình trang trí sẽ phá vỡ mã đó.

Miễn là bạn chỉ viết mã cho loại thành phần trúu tượng, việc sử dụng các trình trang trí sẽ vẫn trong suốt đối với mã của bạn. Tuy nhiên, khi bạn bắt đầu viết mã cho các thành phần cụ thể, bạn sẽ muốn xem xét lại thiết kế ứng dụng và cách sử dụng các trình trang trí của mình.

H: Liệu có dễ dàng cho một số người không?

khách hàng của một loại đồ uống để kết thúc bằng một trình trang trí không phải là trình trang trí ngoài cùng? Giống như nếu tôi có DarkRoast với Mocha, Soy và Whip, sẽ dễ dàng viết mã mà bằng cách nào đó kết thúc bằng một tham chiếu đến Soy thay vì Whip, điều đó có nghĩa là nó sẽ không bao gồm Whip trong thứ tự.

A: Bạn chắc chắn có thể lập luận rằng

bạn phải quản lý nhiều đối tượng hơn với Decorator Pattern và do đó có nhiều khả năng lỗi mã hóa sẽ gây ra các loại vấn đề mà bạn đã xuất. Tuy nhiên, decorator thường được tạo bằng cách sử dụng các mẫu khác như Factory và Builder. Sau khi chúng ta đã tìm hiểu các mẫu này, bạn sẽ thấy rằng việc tạo thành phần cụ thể với decorator của nó được "đóng gói tốt" và không dẫn đến những loại vấn đề này.

Q: Người trang trí có thể biết về các đồ trang trí khác trong chuỗi? Ví dụ, tôi muốn phương thức getDecription() của mình in ra "Whip, Double Mocha" thay vì "Mocha, Whip, Mocha"? Điều đó đòi hỏi trình trang trí ngoài cùng của tôi phải biết tất cả các trình trang trí mà nó đang bao bọc.

A: Các trình trang trí có mục đích là thêm vào hành vi đối với đối tượng mà chúng bao bọc. Khi bạn cần xem qua nhiều lớp trong chuỗi trình trang trí, bạn đang bắt đầu đầy trình trang trí vượt ra ngoài mục đích thực sự của nó. Tuy nhiên, những điều như vậy là có thể. Hãy tưởng tượng một trình trang trí CondimentPrettyPrint phân tích cú pháp mô tả cuối cùng và có thể in "Mocha, Whip, Mocha" thành "Whip, Double Mocha". Lưu ý rằng getDecription() có thể trả về một ArrayList các mô tả để thực hiện việc này dễ dàng hơn.

Chuốt bút chì của bạn



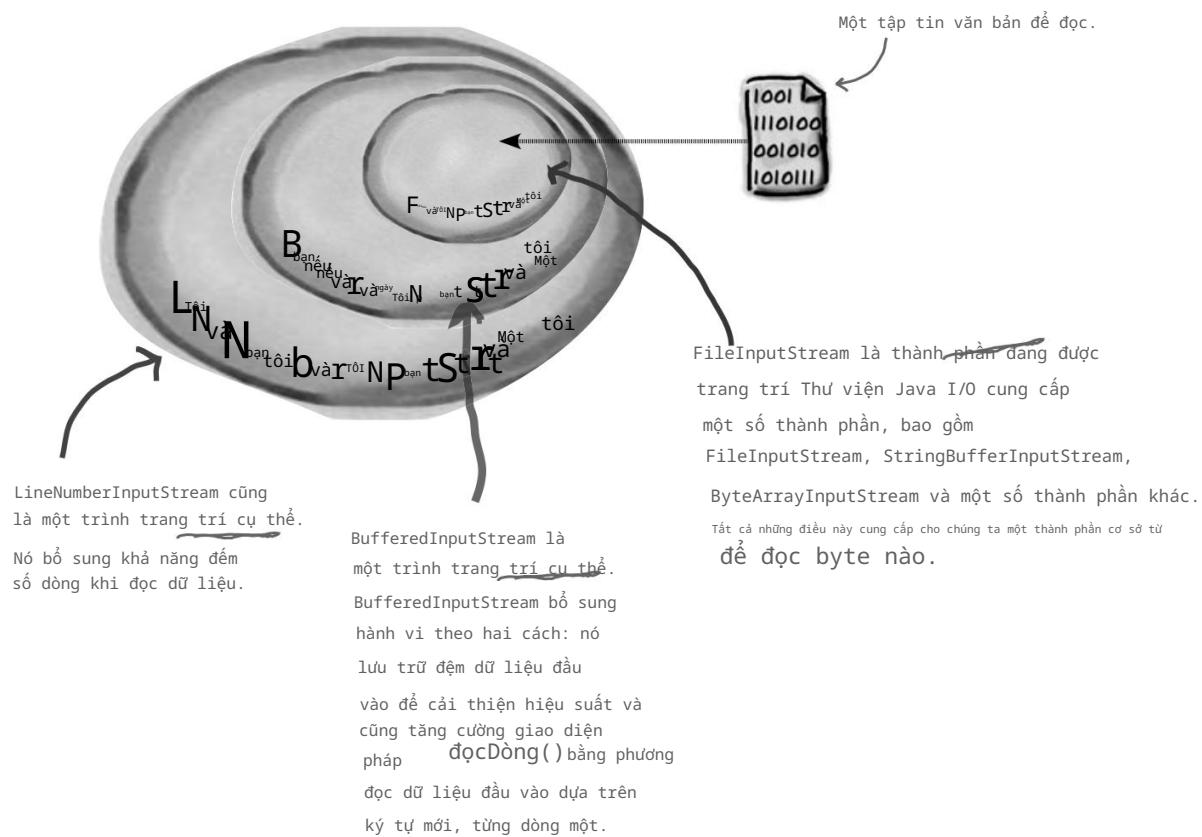
Những người bạn của chúng tôi tại Starbuzz đã giới thiệu các kích cỡ vào thực đơn của họ. Bây giờ bạn có thể gọi một tách cà phê theo kích cỡ tall, grande và venti (tức là nhỏ, vừa và lớn). Starbuzz coi đây là một phần nội tại của lớp cà phê, vì vậy họ đã thêm hai phương thức vào lớp Beverage: setSize() và getSize(). Họ cũng muốn các loại giá vị được tính phí theo kích cỡ, ví dụ, giá của Soy lần lượt là 10¢, 15¢ và 20¢ cho cà phê tall, grande và venti.

Bạn sẽ thay đổi các lớp trang trí như thế nào để xử lý sự thay đổi về yêu cầu này?

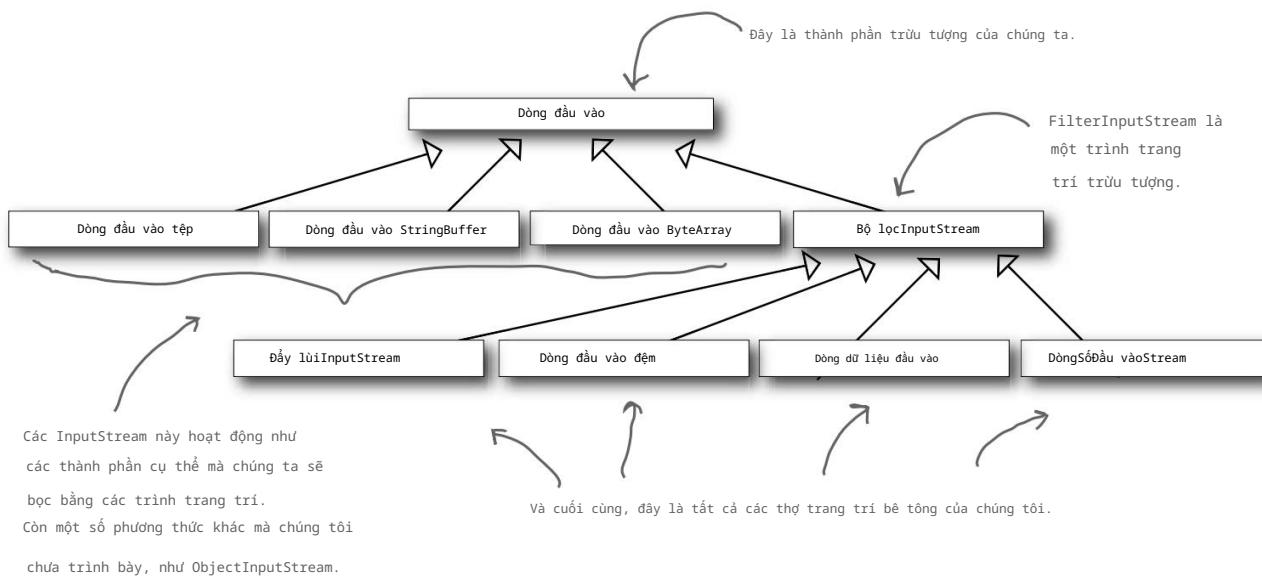
trang trí trong java i/o

Trình trang trí thế giới thực: Java I/O

Số lượng lớn các lớp trong gói `java.io` là... quá nhiều. Đừng cảm thấy cô đơn nếu bạn thốt lên "whoa" lần đầu tiên (và lần thứ hai và thứ ba) khi bạn nhìn vào API này. Nhưng bây giờ bạn đã biết về Mẫu trang trí, các lớp I/O sẽ có ý nghĩa hơn vì gói `java.io` phần lớn dựa trên Decorator. Sau đây là một tập hợp các đối tượng điển hình sử dụng decorator để thêm chức năng vào việc đọc dữ liệu từ một tệp:



Trang trí các lớp java.io



Bạn có thể thấy rằng điều này không khác nhiều so với thiết kế Starbuzz. Giờ bạn đã có thể xem qua tài liệu API java.io và soạn thảo các trình trang trí trên nhiều luồng đầu vào khác nhau.

Và bạn sẽ thấy rằng các luồng đầu ra có cùng thiết kế. Và bạn có thể thấy rằng các luồng Reader/Writer (dành cho dữ liệu dựa trên ký tự) phản ánh chặt chẽ thiết kế của các lớp luồng (với một vài điểm khác biệt và không nhất quán, nhưng đủ gần để tìm ra điều gì đang diễn ra).

Nhưng Java I/O cũng chỉ ra một trong những nhược điểm của Decorator Pattern: các thiết kế sử dụng mẫu này thường tạo ra một số lượng lớn các lớp nhỏ có thể gây choáng ngợp cho nhà phát triển khi cố gắng sử dụng API dựa trên Decorator. Nhưng giờ bạn đã biết Decorator hoạt động như thế nào, bạn có thể giữ mọi thứ trong tầm nhìn và khi bạn sử dụng API nặng về Decorator của người khác, bạn có thể tìm hiểu cách các lớp của họ được tổ chức để bạn có thể dễ dàng sử dụng wrap để có được hành vi bạn mong muốn.

viết trình trang trí i/o của riêng bạn

Viết Java I/O Decorator của riêng bạn

Được rồi, bạn biết về Decorator Pattern, bạn đã thấy sơ đồ lớp I/O. Bạn nên sẵn sàng viết decorator đầu vào của riêng mình.

Thέ này nhé: hãy viết một trình trang trí chuyển đổi tất cả các ký tự viết hoa thành chữ thường trong luồng đầu vào. Nói cách khác, nếu chúng ta đọc trong "Tôi biết Mẫu trang trí do đó TÔI QUYỀN!" thì trình trang trí của bạn sẽ chuyển đổi điều này thành "Tôi biết mẫu trang trí do đó tôi quy tắc!"

Đừng quên import

java.io... (không hiển thị)

Đầu tiên, hãy mở rộng FilterInputStream, trình trang trí trừu tượng cho tất cả InputStream.

```

lớp công khai LowerCaseInputStream mở rộng FilterInputStream {
    công khai LowerCaseInputStream(InputStream trong) {
        siêu(trong);
    }

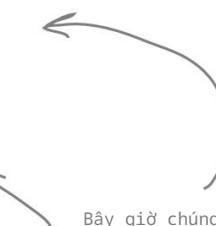
    public int read() ném IOException {
        int c = super.read();
        trả về (c == -1 ? c : Ký tự. thành chữ thường((char)c));
    }

    công khai int đọc (byte [] b, int offset, int len) ném IOException {
        int result = super.read(b, offset, len);
        đổi mới (int i = offset; i < offset+result; i++) {
            b[i] = (byte)Ký tự.thànhChữ thường((ký tự)b[i]);
        }
        trả về kết quả;
    }
}

```

NHỚ RẰNG: chúng tôi không cung cấp các câu lệnh import và package trong danh sách mã. Lấy mã nguồn đầy đủ từ trang web headfirstlabs. Bạn sẽ tìm thấy URL trên trang xxxiii trong phần Giới thiệu.

Không vấn đề gì. Tôi chỉ cần mở rộng lớp FilterInputStream và ghi đè phương thức read().



Bây giờ chúng ta cần triển khai hai phương thức đọc. Chúng lấy một byte (hoặc một mảng byte) và chuyển đổi từng byte (biểu diễn một ký tự) thành chữ thường nếu đó là ký tự viết hoa.

mẫu trang trí

Kiểm tra Java I/O Decorator mới của bạn

Viết một số mã nhanh để kiểm tra trình trang trí I/O:

```

lớp công khai InputTest {
    public static void main(String[] args) ném IOException {
        int c;
        thử {
            InputStream trong =
                new LowerCaseInputStream(
                    BufferedInputStream
                        mới( FileInputStream mới("test.txt")));
            trong khi((c = in.read()) >= 0) {
                System.out.print((char)c);
            }
            trong.close(); }
        catch (IOException e)
        { e.printStackTrace(); }
    }
}

```

Chỉ cần sử dụng luồng để đọc các ký tự cho đến khi kết thúc tệp và in ra khi thực hiện.

Thiết lập FileInputStream và trang trí nó, trước tiên bằng BufferedInputStream

và sau đó là bộ lọcLowerCaseInputStream hoàn toàn mới của chúng tôi.

Tôi biết mẫu Decorator nên tôi là người làm chủ!

tập tin test.txt

Bạn cần
tạo tập tin này.

Hãy thử xem:

```

Cửa sổ chỉnh sửa tệp Trợ giúp DecoratorsRule
% java InputTest tôi biết
mẫu trang trí do đó tôi cai trị! %

```

phóng vấn trang trí



Các mẫu được phơi bày

Cuộc phỏng vấn tuần này:

Lời thú nhận của một người trang trí

HeadFirst: Welcome Decorator Pattern. Chúng tôi nghe nói dạo này bạn hơi chán nản?

Người trang trí: Vâng, tôi biết thế giới coi tôi là một mẫu thiết kế quyền rũ, nhưng bạn biết đấy, tôi cũng có những vấn đề riêng giống như mọi người.

HeadFirst: Bạn có thể chia sẻ một số rắc rối của mình với chúng tôi không?

Decorator: Chắc chắn rồi. Vâng, bạn biết đấy, tôi có khả năng thêm tính linh hoạt vào các thiết kế, chắc chắn là vậy, nhưng tôi cũng có mặt tối. Bạn thấy đấy, đôi khi tôi có thể thêm nhiều lớp nhỏ vào một thiết kế và điều này đôi khi dẫn đến một thiết kế không dễ hiểu đối với người khác.

HeadFirst: Bạn có thể cho chúng tôi một ví dụ không?

Decorator: Lấy các thư viện Java I/O làm ví dụ. Thoạt đầu, chúng rất khó hiểu đối với mọi người.

Nhưng nếu họ chỉ xem các lớp như một tập hợp các wrapper xung quanh InputStream, cuộc sống sẽ dễ dàng hơn nhiều.

HeadFirst: Nghe có vẻ không tệ lắm. Bạn vẫn là một mô hình tuyệt vời, và cải thiện điều này chỉ là vấn đề giáo dục cộng đồng, đúng không?

Decorator: Tôi e rằng còn nhiều hơn thế nữa. Tôi gặp vấn đề về đánh máy: bạn thấy đấy, đôi khi mọi người lấy một đoạn mã của khách hàng dựa trên các kiểu cụ thể và giới thiệu các decorator mà không suy nghĩ thấu đáo mọi thứ. Bây giờ, một điều tuyệt vời về tôi là bạn thường có thể chèn các decorator một cách minh bạch và khách hàng không bao giờ phải biết rằng nó đang xử lý một decorator. Nhưng như tôi đã nói, một số mã phụ thuộc vào các kiểu cụ thể và khi bạn bắt đầu giới thiệu các decorator, bùm! Những điều tôi tệ xảy ra.

HeadFirst: Vâng, tôi nghĩ mọi người đều hiểu rằng bạn phải cẩn thận khi chèn trình trang trí, tôi không nghĩ đây là lý do để bạn quá thất vọng về bản thân.

Decorator: Tôi biết, tôi có gắng không như vậy. Tôi cũng gặp vấn đề là việc giới thiệu decorator có thể làm tăng độ phức tạp của mã cần thiết để khởi tạo thành phần. Khi bạn đã có decorator, bạn không chỉ phải khởi tạo thành phần mà còn phải bao bọc nó bằng không biết bao nhiêu decorator.

HeadFirst: Tôi sẽ phỏng vấn các mẫu Factory và Builder vào tuần tới - Tôi nghe nói chúng có thể rất hữu ích trong việc này?

Người trang trí: Đúng vậy; tôi nên nói chuyện với những người đó thường xuyên hơn.

HeadFirst: Vâng, tất cả chúng tôi đều nghĩ bạn là một hình mẫu tuyệt vời để tạo ra các thiết kế linh hoạt và tuân thủ Nguyên tắc Mở-Đóng, vì vậy hãy ngẩng cao đầu và suy nghĩ tích cực nhé!

Người trang trí: Tôi sẽ cố gắng hết sức, cảm ơn bạn.

mẫu trang trí



Công cụ cho hộp công cụ thiết kế của bạn

Bạn đã học thêm một chương nữa và có thêm một nguyên tắc và mô hình mới trong hộp công cụ.

Nguyên tắc 00

Bao gồm những gì thay đổi.

Ưu tiên thành phần hơn là thừa kế.

Chương trình hướng tới giao diện, không phải triển khai.

Có gắng thiết kế các đối tượng tương tác một cách lỏng lẻo.

Các lớp học nên được mở để gia hạn nhưng đóng cửa để sự sửa đổi.

Bây giờ chúng ta có Nguyên tắc Mở Đóng để hướng dẫn chúng ta. Chúng ta sẽ cố gắng thiết kế hệ thống của mình sao cho các phần đóng được tách biệt khỏi các phần mở rộng mới của chúng ta.

Mã Patterns

Chiến lược xác định một họ các thuật toán, được áp dụng cho một khung mà nó đang thay đổi trạng thái tất cả các thuộc tính của nó, nhằm chia nhỏ nó thành các khía cạnh khác nhau. Điều này là khía cạnh để sử dụng nó.

cung cấp một giải pháp thay thế linh hoạt cho việc phân lớp để mở rộng chức năng.

Và đây là mẫu *factory pattern* for creating

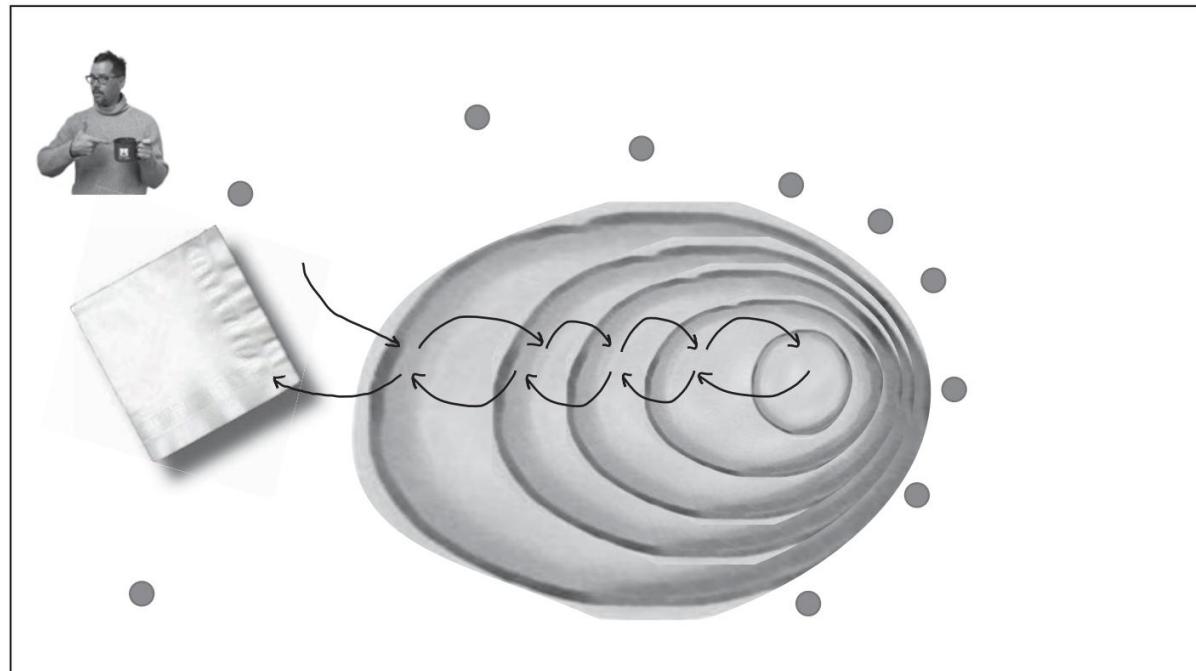
Để tạo một *factory pattern* Ngay khiếp Mở Đóng hay trao đổi. Mở Đóng.

là mẫu đầu tiên? Có mẫu nào khác mà chúng tôi đã sử dụng cũng tuân theo nguyên tắc này không?



ĐIỂM ĐẦU TIÊN

- β Di truyền là một hình thức mở rộng, nhưng không nhất thiết là cách tốt nhất để đạt được tính linh hoạt trong thiết kế của chúng ta.
- β Trong thiết kế, chúng ta nên cho phép mở rộng hành vi mà không cần phải sửa đổi mã hiện có.
- β Thành phần và phân quyền thường có thể được sử dụng để thêm các hành vi mới khi chạy.
- β Mẫu trang trí cung cấp một giải pháp thay thế cho việc phân lớp để mở rộng hành vi.
- β Mẫu trang trí bao gồm một tập hợp các lớp trang trí được sử dụng để bọc các thành phần cụ thể.
- β Các lớp trang trí phản ánh kiểu của các thành phần mà chúng trang trí. (Trên thực tế, chúng cùng kiểu với các thành phần mà chúng trang trí, thông qua kế thừa hoặc triển khai giao diện.)
- β Trình trang trí thay đổi hành vi của các thành phần của chúng bằng cách thêm chức năng mới trước và/hoặc sau (hoặc thậm chí thay thế) các lệnh gọi phương thức đến thành phần.
- β Bạn có thể bọc một thành phần bằng bất kỳ số lượng trình trang trí nào.
- β Trình trang trí thường trong suốt với máy khách của thành phần; nghĩa là, trừ khi máy khách đang dựa vào kiểu cụ thể của thành phần.
- β Trình trang trí có thể tạo ra nhiều đối tượng nhỏ trong thiết kế của chúng ta và việc sử dụng quá mức có thể trở nên phức tạp.



Giải pháp bài tập

Những người bạn của chúng tôi tại Starbuzz đã giới thiệu kích thước vào thực đơn của họ. Bây giờ bạn có thể gọi cà phê theo kích thước tall, grande và venti (đối với những người bình thường như chúng tôi: nhỏ, vừa và lớn). Starbuzz coi đây là một phần nội tại của lớp cà phê, vì vậy họ đã thêm hai phương thức vào lớp Beverage: setSize() và getSize(). Họ cũng muốn các loại gia vị được tính phí theo kích thước, ví dụ, Soy có giá lần lượt là 10¢, 15¢ và 20¢ cho cà phê tall, grande và venti.

Bạn sẽ thay đổi các lớp trang trí như thế nào để xử lý sự thay đổi về yêu cầu này?

```

lớp công khai Soy mở rộng CondimentDecorator {
    đồ uống giải khát;

    công cộng Đậu nành (Đồ uống giải khát) {
        this.beverage = đồ uống;
    }

    công khai int getSize() {
        trả về drink.getSize();
    }

    công khai String getDescription() {
        trả về beverage.getDescription() + ", Đậu nành";
    }

    công khai double cost() {
        chi phí gấp đôi = chi phí đồ uống();
        nếu (getSize() == Beverage.TALL) {
            chi phí += .10;
        } nếu không thì nếu (getSize() == Beverage.GRANDE) {
            chi phí += .15;
        } nếu không thì nếu (getSize() == Beverage.VENTI) {
            chi phí += .20;

        } chi phí trả lại;
    }
}

```

Bây giờ chúng ta cần truyền phương thức getSize() đến đồ uống được gói. Chúng ta cũng nên chuyển phương thức này đến lớp trùm tương tự nó được sử dụng trong tất cả các trình trang trí gia vị.

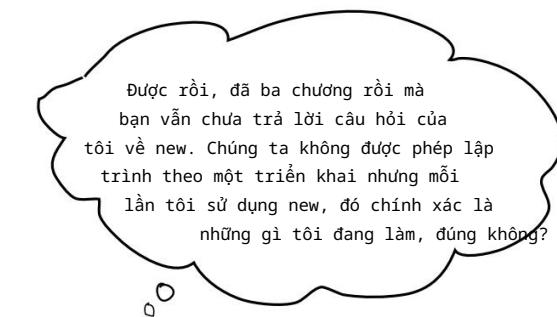
Ở đây chúng ta lấy kích thước (lần truyền đến toàn bộ đồ uống cụ thể) và sau đó thêm chi phí phù hợp.

4 Mẫu Nhà Máy h g Nướng với Ô Lòng tốt g



Hãy chuẩn bị để nướng một số thiết kế kết hợp lồng leo. Còn nhiều điều hơn thế nữa tạo ra các đối tượng hơn là chỉ sử dụng toán tử mới . Bạn sẽ học được rằng việc khởi tạo là một hoạt động không nên luôn luôn được thực hiện ở nơi công cộng và thường có thể dẫn đến các vấn đề về ghép đôi. Và bạn không muốn bạn có vây không? Tìm hiểu cách Factory Patterns có thể giúp bạn tránh khỏi những sự phụ thuộc đáng xấu hổ.

suy nghĩ về "mới"



Khi bạn nhìn thấy "mới", hãy nghĩ đến "cụ thể".

Vâng, khi bạn sử dụng new, bạn chắc chắn đang khởi tạo một lớp cụ thể, vì vậy
đó chắc chắn là một triển khai, không phải là một giao diện. Và đó là một câu hỏi
hay; bạn đã học được rằng việc liên kết mã của bạn với một lớp cụ thể có
thể khiến nó trở nên mạnh mẽ và linh hoạt hơn.

Vịt vịt = new MallardDuck();

Chúng tôi muốn sử dụng giao
diện để giữ cho mã linh hoạt.

Nhưng chúng ta phải tạo
một thể hiện của một lớp cụ thể!

Khi bạn có toàn bộ các lớp cụ thể có liên quan, bạn thường buộc phải viết mã như
thế này:

Vịt vịt;

```
néu (dã ngoại) {
    vịt = vịt Mallard mới();
} else if (sân bắn) {
    vịt = new DecoyDuck();
} néu không thì (trong Bồn tắm) {
    vịt = RubberDuck mới();
}
```

Chúng ta có một loạt các lớp
duck khác nhau và không biết
cần khởi tạo lớp nào cho đến khi
chạy chương trình.

Ở đây chúng ta có một số lớp cụ thể đang được khởi tạo và quyết định khởi tạo lớp
nào được đưa ra khi chạy tùy thuộc vào một số điều kiện nhất định.

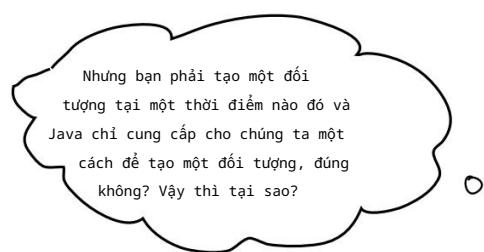
Khi bạn thấy mã như thế này, bạn biết rằng khi đến lúc cần thay đổi hoặc mở rộng,
bạn sẽ phải mở lại mã này và kiểm tra những gì cần thêm (hoặc xóa). Thường thì loại
mã này sẽ nằm ở nhiều phần của ứng dụng khiến việc bảo trì và cập nhật trở nên khó
khăn và dễ xảy ra lỗi hơn.

"Mới" có gì sai?

Về mặt kỹ thuật, không có gì sai với new, sau cùng, đó là một phần cơ bản của Java. Thủ phạm thực sự là người bạn cũ CHANGE của chúng ta và cách change tác động đến việc sử dụng new của chúng ta.

Bằng cách mã hóa vào một giao diện, bạn biết rằng bạn có thể tự lập trình khỏi nhiều thay đổi có thể xảy ra với một hệ thống trong tương lai. Tại sao? Nếu mã của bạn được viết vào một giao diện, thì nó sẽ hoạt động với bất kỳ lớp mới nào triển khai giao diện đó thông qua đơ hình. Tuy nhiên, khi bạn có mã sử dụng nhiều lớp cụ thể, bạn đang tìm kiếm rắc rối vì mã đó có thể phải được thay đổi khi các lớp cụ thể mới được thêm vào. Vì vậy, nói cách khác, mã của bạn sẽ không bị "đóng đẽ sửa đổi". Để mở rộng nó với các kiểu cụ thể mới, bạn sẽ phải mở lại nó.

Vậy bạn có thể làm gì? Vào những lúc như thế này, bạn có thể dựa vào Nguyên tắc thiết kế OO để tìm kiếm manh mối. Hãy nhớ rằng, nguyên tắc đầu tiên của chúng ta liên quan đến sự thay đổi và hướng dẫn chúng ta xác định các khía cạnh thay đổi và tách chúng ra khỏi những gì vẫn giữ nguyên.



Hãy nhớ rằng các thiết kế phải "mở để mở rộng nhưng đóng để sửa đổi" - xem

Chương 3 để biết thêm thông tin.

Sức náo mạnh

Làm thế nào bạn có thể lấy tất cả các phần của ứng dụng khởi tạo các lớp cụ thể và tách hoặc đóng gói chúng khỏi phần còn lại của ứng dụng?

xác định những gì thay đổi

Xác định các khía cạnh khác nhau

Giả sử bạn có một cửa hàng pizza và là chủ một cửa hàng pizza hiện đại ở Objectville, bạn có thể viết một số mã như thế này:

```
Đặt pizzaPizza() {
    Pizza pizza = Pizza mới();
    pizza.chuẩn bị();
    pizza.nướng();
    pizza.cắt();
    pizza.hộp();
    trả lại pizza;
}
```



Để linh hoạt hơn, chúng ta thực sự muốn đây là một lớp trừu tượng hoặc giao diện, nhưng chúng ta không thể trực tiếp khởi tạo bất kỳ lớp hoặc giao diện nào trong số đó.

Nhưng bạn cần nhiều hơn một loại pizza...

Sau đó, bạn sẽ thêm một số xác định loại pizza phù hợp và sau đó bắt đầu làm pizza:

```
Đặt hàng pizzaPizza(Loại chuỗi) {
    Pizza, pizza;
    nếu (type.equals("cheese")) {
        pizza = CheesePizza mới();
    } else if (type.equals("greek")) {
        pizza = pizza Hy Lạp mới();
    } else if (type.equals("pepperoni")) {
        pizza = pizza Pepperoni mới();
    }
    pizza.chuẩn bị();
    pizza.nướng();
    pizza.cắt();
    pizza.hộp();
    trả lại pizza;
}
```

Bây giờ chúng ta đang chuyển loại pizza sang đặt Pizza.

Dựa trên loại pizza, chúng tôi khởi tạo lớp cụ thể chính xác và gán nó cho biến thể hiện pizza. Lưu ý rằng mỗi pizza ở đây phải triển khai giao diện Pizza.

Khi đã có Pizza, chúng tôi sẽ chuẩn bị (bạn biết đấy, cắn bột, rưới nước sốt và thêm các loại topping & phô mai), sau đó chúng tôi nướng, cắt và đóng hộp!

Mỗi loại Pizza (CheesePizza, VeggiePizza, v.v.) đều có cách chế biến riêng.

Nhưng áp lực đang tăng lên để thêm nhiều loại pizza hơn

Bạn nhận ra rằng tất cả các đối thủ cạnh tranh của bạn đã thêm một vài loại pizza hợp thời trang vào thực đơn của họ: Clam Pizza và Veggie Pizza. Rõ ràng là bạn cần phải theo kịp đối thủ cạnh tranh, vì vậy bạn sẽ thêm những món này vào thực đơn của mình. Và gần đây bạn không bán nhiều Greek Pizza, vì vậy bạn quyết định loại bỏ món đó khỏi thực đơn:

Mã này KHÔNG được đóng để sửa đổi.
Nếu Pizza Shop thay đổi các loại pizza, chúng tôi phải nhập mã này và sửa đổi.

```
Đặt hàng pizzaPizza(Loại chuỗi) {
    Pizza, pizza;

    nếu (type.equals("cheese")) {
        pizza = CheesePizza mới();
    } else if (type.equals("greek")) {
        pizza = pizza Hy Lạp mới();
    } else if (type.equals("pepperoni")) {
        pizza = pizza Pepperoni mới();
    } else if (type.equals("clam")) {
        pizza = ClamPizza mới();
    } else if (type.equals("veggie")) {
        pizza = VeggiePizza mới();
    }

    pizza.chuẩn bị();
    pizza. nướng();
    pizza. cắt();
    pizza. hộp();
    trả lại pizza;
}
```

Đây chính là điều khác biệt.
Vì danh mục pizza thay đổi theo thời gian, bạn sẽ phải sửa đổi mã này nhiều lần.

Đây là những gì chúng tôi mong đợi sẽ giữ nguyên. Phần lớn, việc chuẩn bị, nấu và đóng gói pizza vẫn giữ nguyên trong nhiều năm. Vì vậy, chúng tôi không mong đợi mã này thay đổi, chỉ những chiếc pizza mà nó hoạt động.

Rõ ràng, việc xử lý lớp cụ thể nào được khởi tạo thực sự làm hỏng phương thức `orderPizza()` của chúng ta và ngăn không cho nó được đóng lại để sửa đổi. Nhưng bây giờ chúng ta đã biết những gì đang thay đổi và những gì không, có lẽ đã đến lúc đóng gói nó.

đóng gói việc tạo đối tượng

Đóng gói việc tạo đối tượng

Vậy bây giờ chúng ta biết rằng sẽ tốt hơn nếu di chuyển việc tạo đối tượng ra khỏi phương thức `orderPizza()`. Nhưng làm thế nào?

Vâng, những gì chúng ta sẽ làm là lấy mã tạo và di chuyển nó ra ngoài vào một đối tượng khác chỉ liên quan đến việc tạo pizza.

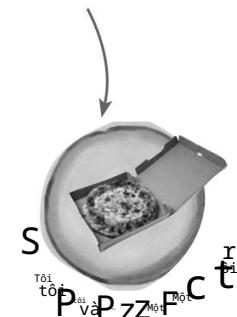
```
Đặt hàng pizzaPizza(Loại chuỗi) {
    Pizza, pizza;
    pizza.chuẩn bị();
    pizza.nướng();
    pizza.cắt();
    pizza.hộp();
    trả lại pizza;
}
```

Đầu tiên chúng ta kéo mã tạo đối tượng ra khỏi Phương thức `orderPizza`

Chuyện gì đang xảy ra để đi đến đây?

```
nếu (type.equals("cheese")) {
    pizza = CheesePizza mới();
} else if (type.equals("pepperoni")) {
    pizza = pizza Pepperoni mới();
} else if (type.equals("clam")) {
    pizza = ClamPizza mới();
} else if (type.equals("veggie")) {
    pizza = VeggiePizza mới();
}
```

Sau đó, chúng ta đặt mã đó vào một đối tượng chỉ lo về cách tạo pizza. Nếu bất kỳ đối tượng nào khác cần tạo pizza, thì đây là đối tượng cần đến.



Chúng tôi đặt tên cho đối tượng mới này: chúng tôi gọi nó là Nhà máy.

Các nhà máy xử lý các chi tiết của việc tạo đối tượng. Khi chúng ta có `SimplePizzaFactory`, phương thức `orderPizza()` của chúng ta chỉ trở thành một máy khách của đối tượng đó. Bất cứ khi nào nó cần một chiếc pizza, nó sẽ yêu cầu nhà máy pizza làm một chiếc. Đã qua rồi cái thời mà phương thức `orderPizza()` cần biết về pizza Hy Lạp so với pizza Clam. Bây giờ phương thức `orderPizza()` chỉ quan tâm đến việc nó nhận được một chiếc pizza, phương thức này triển khai giao diện `Pizza` để có thể gọi `prepare()`, `bake()`, `cut()` và `box()`.

Chúng ta vẫn còn một vài chi tiết cần điền vào đây; ví dụ, phương thức `orderPizza()` thay thế mã tạo của nó bằng gì? Hãy triển khai một nhà máy đơn giản cho cửa hàng pizza và tìm hiểu...

Xây dựng một nhà máy sản xuất pizza đơn giản

Chúng ta sẽ bắt đầu với chính nhà máy. Những gì chúng ta sẽ làm là định nghĩa một lớp đóng gói việc tạo đối tượng cho tất cả các loại pizza. Đây là nó...

Đây là lớp học mới của chúng tôi, SimplePizzaFactory. Lớp này có một nhiệm vụ duy nhất: tạo ra những chiếc pizza cho khách hàng.

```
lớp công khai SimplePizzaFactory {
    công khai Pizza createPizza(Kiểu chuỗi) {
        Pizza pizza = null;
```

```
        nếu (type.equals("cheese")) {
            pizza = CheesePizza mới();
        } else if (type.equals("pepperoni")) {
            pizza = pizza Pepperoni mới();
        } else if (type.equals("clam")) {
            pizza = ClamPizza mới();
        } else if (type.equals("veggie")) {
            pizza = VeggiePizza mới();
        }
        trả lại pizza;
    }
```

Đầu tiên chúng ta định nghĩa phương thức createPizza() trong factory. Đây là phương thức mà tất cả các máy khách sẽ sử dụng để khởi tạo các đối tượng mới.

Sau đây là đoạn mã chúng tôi lấy từ phương thức orderPizza().

Mã này vẫn được tham số hóa theo loại pizza, giống như phương thức orderPizza() ban đầu của chúng ta.

không có câu hỏi ngớ ngẩn nào

H: Ưu điểm của việc này là gì?
Có vẻ như chúng ta đang đẩy vấn đề sang đối tượng khác.

A: Một điều cần nhớ là SimplePizzaFactory có thể có nhiều khách hàng. Chúng ta chỉ thấy phương thức orderPizza(); tuy nhiên, có thể có một lớp PizzaShopMenu sử dụng factory để lấy pizza theo mô tả và giá hiện tại của chúng. Chúng ta cũng có thể có một lớp HomeDelivery xử lý pizza theo cách khác với

Lớp PizzaShop nhưng cũng là khách hàng của nhà máy.

Vì vậy, bằng cách đóng gói việc tạo pizza vào trong một lớp, giờ đây chúng ta chỉ có một nơi để thực hiện sửa đổi khi quá trình triển khai thay đổi. Đừng quên, chúng ta sắp xóa các phiên bản cũ thẻ khỏi mã máy khách!

H: Tôi đã thấy một thiết kế tương tự một nhà máy như thế này được định nghĩa là một phương pháp tĩnh. Sự khác biệt là gì?

A: Định nghĩa một nhà máy đơn giản là phương thức tĩnh là một kỹ thuật phổ biến và thường được gọi là nhà máy tĩnh. Tại sao lại sử dụng phương thức tĩnh? Bởi vì bạn không cần phải khởi tạo một đối tượng để sử dụng phương thức tạo. Nhưng hãy nhớ rằng nó cũng có nhược điểm là bạn không thể phân lớp và thay đổi hành vi của phương thức create.

nhà máy đơn giản

Làm lại lớp PizzaStore

Bây giờ là lúc sửa mã máy khách của chúng ta. Điều chúng ta muốn làm là dựa vào nhà máy để tạo ra pizza cho chúng ta. Sau đây là những thay đổi:

```

Bây giờ chúng ta cung cấp cho PizzaStore một
tham chiếu tới SimplePizzaFactory.

lớp công khai PizzaStore {
    Nhà máy SimplePizzaFactory;

    công khai PizzaStore(nhà máy SimplePizzaFactory) {
        this.factory = nhà máy;
    }

    công khai Pizza orderPizza(String type) {
        Pizza, pizza;

        pizza = factory.createPizza(kiểu);

        pizza.chuẩn bị();
        pizza. nướng();
        pizza. cắt();
        pizza. hộp();
        trả lại pizza;
    }

    // các phương pháp khác ở đây
}

```

PizzaStore sẽ được truyền nhà máy vào trong hàm tạo.

Và phương thức orderPizza() sử dụng nhà máy để tạo ra pizza bằng cách chỉ cần truyền vào loại đơn hàng.

Lưu ý rằng chúng tôi đã thay thế toán tử new bằng phương thức create trên đối tượng factory. Không còn bất kỳ khởi tạo cụ thể nào ở đây nữa!

não Apower

Chúng ta biết rằng thành phần đối tượng cho phép chúng ta thay đổi hành vi động tại thời điểm chạy (cùng với những thứ khác) vì chúng ta có thể hoán đổi các triển khai trong và ngoài. Làm thế nào chúng ta có thể sử dụng điều đó trong PizzaStore? Chúng ta có thể hoán đổi các triển khai nhà máy nào trong và ngoài?

cũng đừng quên New Haven nữa
Chúng tôi không biết về bạn, như

Nhà máy đơn giản được định nghĩa

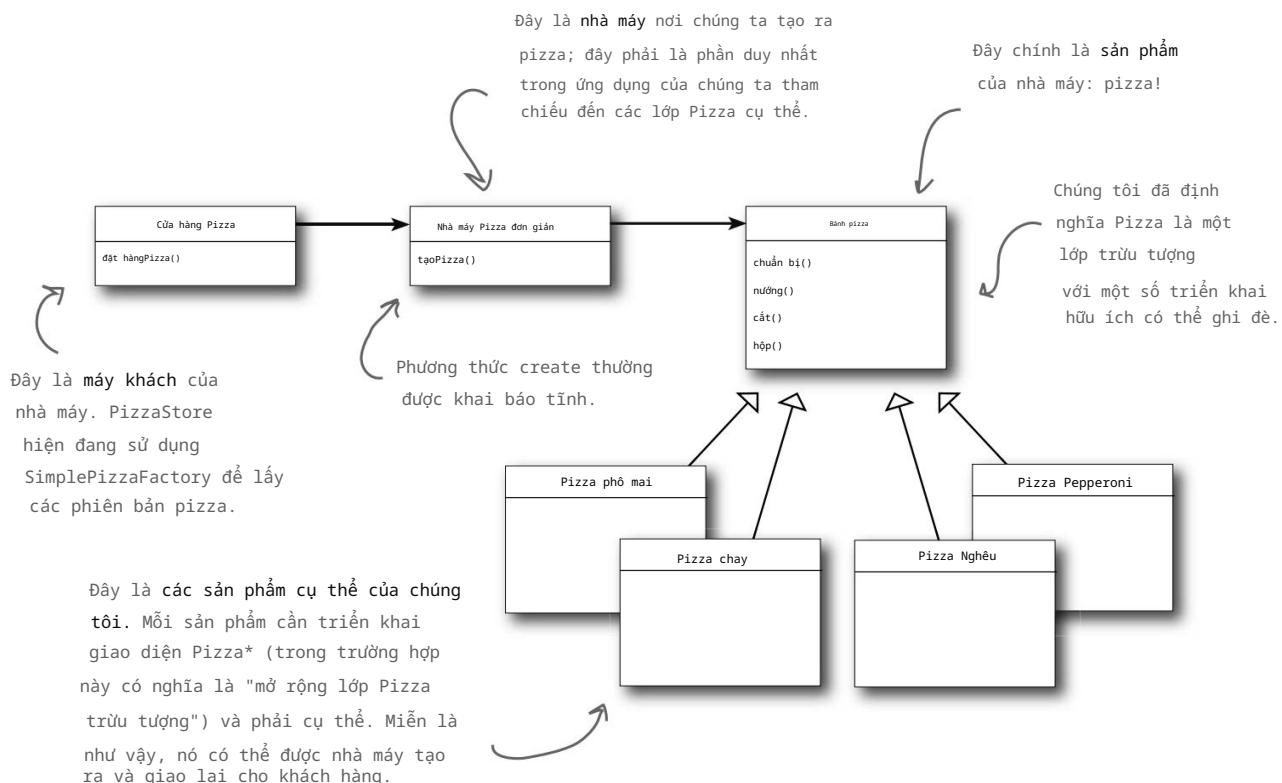
Simple Factory thực chất không phải là một Mẫu thiết kế; nó giống một thuật ngữ lập trình hơn

Nhưng nó được sử dụng rất phổ biến, vì vậy chúng tôi sẽ trao cho nó giải Khuyến khích cho Head First Pattern

Một số nhà phát triển nhằm lắn thành ngữ này với "Factory Pattern", vì vậy, lần sau khi có sự im lặng khó xử giữa bạn và một nhà phát triển khác, bạn sẽ có một chủ đề hay để phá vỡ sự im lặng đó.



Chỉ vì Simple Factory không phải là một mẫu THỰC SỰ không có nghĩa là chúng ta không nên kiểm tra cách nó được tạo ra. Hãy cùng xem sơ đồ lớp của Cửa hàng Pizza mới của chúng ta:



Hãy coi Simple Factory như một phần khởi động. Tiếp theo, chúng ta sẽ khám phá hai mẫu nặng đều là nhà máy. Nhưng đừng lo, vẫn còn nhiều pizza nữa!

*Chỉ là một lời nhắc nhở nữa: trong các mẫu thiết kế, cụm từ “triển khai giao diện” KHÔNG phải lúc nào cũng có nghĩa là “viết một lớp triển khai giao diện Java, bằng cách sử dụng từ khóa “triển khai” trong kiểu (có thể khai báo lớp”. Trong cụm từ này, một lớp cụ thể triển khai một phương thức từ một siêu lớp sử dụng chung HOẶC giao diện) vẫn được coi là “triển khai giao diện” của siêu kiểu đó.

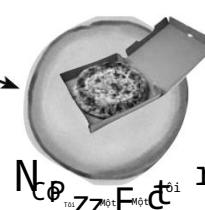
nhượng quyền pizza

Nhượng quyền cửa hàng pizza

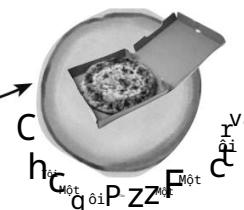
Objectville PizzaStore của bạn đã hoạt động rất tốt đến mức bạn đã đánh bại đối thủ cạnh tranh và giờ đây mọi người đều muốn có một PizzaStore trong khu phố của họ. Là bên nhượng quyền, bạn muốn đảm bảo chất lượng hoạt động nhượng quyền và do đó bạn muốn họ sử dụng mã đã được kiểm tra theo thời gian của bạn.

Nhưng còn sự khác biệt về khu vực thì sao? Mỗi cửa hàng nhượng quyền có thể muốn cung cấp các loại pizza khác nhau (ví dụ như New York, Chicago và California), tùy thuộc vào vị trí của cửa hàng nhượng quyền và khẩu vị của những người sành pizza địa phương.

Bạn muốn tất cả các cửa hàng pizza nhượng quyền tận dụng mã PizzaStore của bạn để các loại pizza được chế biến theo cùng một cách.



Một thương hiệu nhượng quyền muốn có một nhà máy làm pizza theo phong cách New York: đề mỏng, nước sốt ngọt ngon và chỉ một ít phô mai.



Một thương hiệu nhượng quyền khác muốn có một nhà máy sản xuất pizza theo phong cách Chicago; khách hàng của họ thích pizza có đề dày, nước sốt đậm đà và nhiều phô mai.

Chúng ta đã thấy một cách tiếp cận...

Nếu chúng ta loại bỏ SimplePizzaFactory và tạo ra ba nhà máy khác nhau, NYPizzaFactory, ChicagoPizzaFactory và CaliforniaPizzaFactory, thì chúng ta có thể chỉ cần tạo PizzaStore với nhà máy phù hợp và nhượng quyền thương mại là xong. Đó là một cách tiếp cận.

Hãy cùng xem nó trông như thế nào nhé...

```
NYPizzaFactory nyFactory = new NYPizzaFactory();
PizzaStore nyStore = new PizzaStore(nyFactory);
nyStore.order("Rau củ");
```

Ở đây chúng tôi tạo ra một nhà máy
để làm pizza theo phong cách New York.

Sau đó, chúng ta tạo một PizzaStore và truyền cho nó một tham
chiếu đến nhà máy NY.

...và khi chúng tôi làm pizza, chúng tôi sẽ
có pizza theo phong cách New York.

```
ChicagoPizzaFactory chicagoFactory = new ChicagoPizzaFactory();
PizzaStore chicagoStore = mới PizzaStore(chicagoFactory);
chicagoStore.order("Rau củ");
```

Tương tự như vậy đối với các cửa hàng pizza Chicago: chúng tôi tạo ra
một nhà máy sản xuất pizza Chicago và tạo ra một cửa hàng được tạo thành
từ một nhà máy Chicago. Khi chúng tôi làm pizza, chúng tôi có hương vị
Chicago
những cái

Nhưng bạn muốn kiểm soát chất lượng tốt hơn một chút...

Vì vậy, bạn đã thử nghiệm tiếp thị ý tưởng SimpleFactory và
phát hiện ra rằng các cửa hàng nhượng quyền đang sử dụng
nhà máy của bạn để làm pizza, nhưng bắt đầu áp dụng các quy
trình riêng của họ cho phần còn lại của quy trình: họ nướng
bánh theo cách hơi khác một chút, họ quên cắt pizza và sử
dụng hộp của bên thứ ba.

Xem xét lại vấn đề một chút, bạn thấy rằng điều bạn thực sự muốn làm là
tạo ra một khuôn khổ gắn kết cửa hàng và việc làm pizza lại với nhau,
nhưng vẫn đảm bảo tính linh hoạt.

Trong mã ban đầu của chúng tôi, trước SimplePizzaFactory, chúng
tôi có mã làm pizza được liên kết với PizzaStore, nhưng nó không
linh hoạt. Vậy, làm sao chúng tôi có thể vừa có pizza vừa ăn pizza?

Không phải là điều bạn muốn ở một
thương hiệu nhượng quyền tốt. Bạn KHÔNG muốn
biết anh ta cho gì vào pizza của mình.



bạn đang ở đây 4 119

để các lớp con quyết định

Một khuôn khổ cho cửa hàng pizza

Có một cách để bắn địa hào tất cả các hoạt động làm pizza cho lớp PizzaStore, nhưng vẫn tạo cho các cửa hàng nhượng quyền sự tự do trong việc duy trì phong cách riêng của khu vực.

Nhưng gì chúng ta sẽ làm là đưa phương thức createPizza() trở lại PizzaStore, nhưng lần này là một phương thức trừu tượng, sau đó tạo một lớp con PizzaStore cho mỗi khu vực.

Đầu tiên, chúng ta hãy xem những thay đổi của PizzaStore:

PizzaStore hiện đã mang tính trừu tượng (xem lý do bên dưới).



```
lớp trừu tượng công khai PizzaStore {
```

```
    công khai Pizza orderPizza(String type) {
```

```
        Pizza, pizza;
```

```
        pizza = createPizza(kiểu);
```

```
        pizza.chuẩn bị();
```

```
        pizza.nướng();
```

```
        pizza.cắt();
```

```
        pizza.hộp();
```

```
        trả lại pizza;
```

```
}
```

```
    trừu tượng Pizza createPizza(String type);
```

```
}
```

“Phương pháp nhà máy” của chúng tôi hiện đã được trừu tượng hóa trong PizzaStore.

Bây giờ createPizza lại quay trở lại là lệnh gọi phương thức trong PizzaStore thay vì trên một đối tượng nhà máy.

Tất cả trông đều giống nhau...

Bây giờ chúng ta đã di chuyển đối tượng nhà máy của mình đến phương thức này.

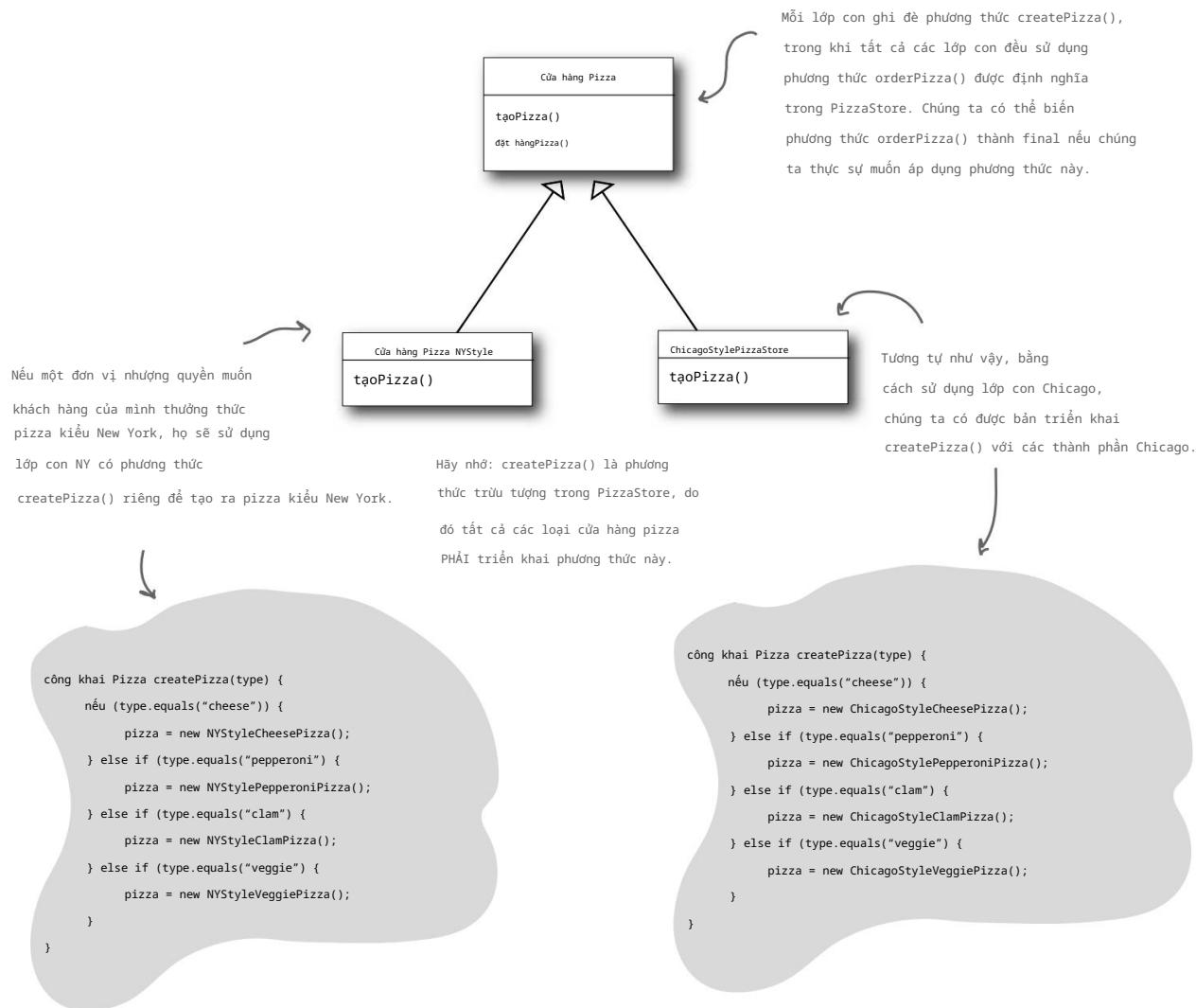
Bây giờ chúng ta có một cửa hàng đang chờ các lớp con; chúng ta sẽ có một lớp con cho mỗi loại khu vực (NYPizzaStore, ChicagoPizzaStore, CaliforniaPizzaStore) và mỗi lớp con sẽ đưa ra quyết định về những gì tạo nên một chiếc pizza. Hãy cùng xem cách thức hoạt động của nó.

Cho phép các lớp con quyết định

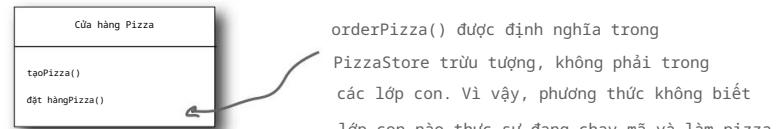
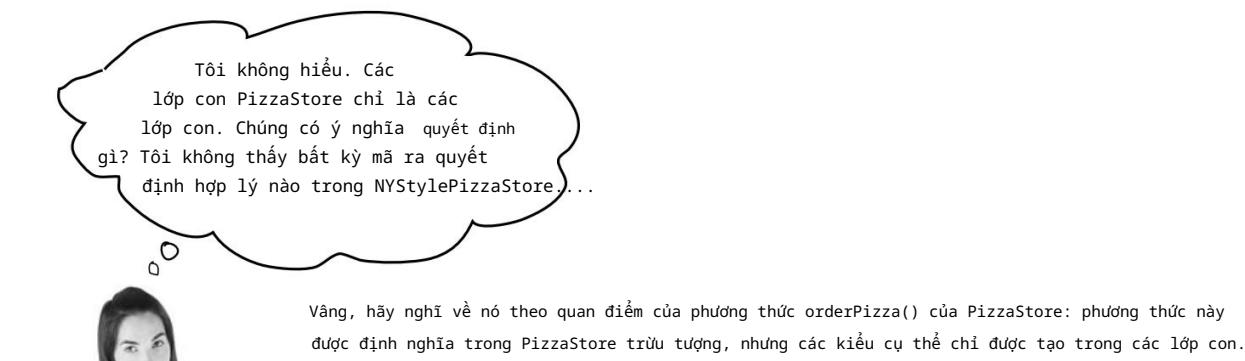
Hãy nhớ rằng, PizzaStore đã có một hệ thống đặt hàng được hoàn thiện trong phương thức `orderPizza()` và bạn muốn đảm bảo rằng nó thống nhất trên tất cả các cửa hàng nhượng quyền.

Điểm khác biệt giữa các PizzaStores theo khu vực là phong cách pizza mà họ làm - Pizza New York có đế mỏng, Pizza Chicago có đế dày, v.v. - và chúng ta sẽ đưa tất cả các biến thể này vào phương thức `createPizza()` và khiến nó chịu trách nhiệm tạo ra đúng loại pizza. Cách chúng ta thực hiện điều này là để mỗi lớp con của PizzaStore định nghĩa phương thức `createPizza()` trông như thế nào.

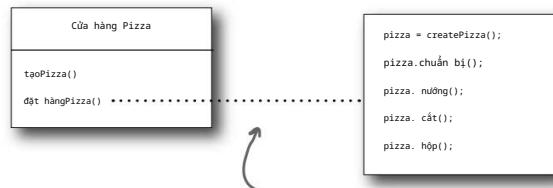
Vì vậy, chúng ta sẽ có một số lớp con cụ thể của PizzaStore, mỗi lớp có các biến thể pizza riêng, tất cả đều phù hợp trong khuôn khổ PizzaStore và vẫn sử dụng phương thức `orderPizza()` được điều chỉnh tốt.



các lớp con quyết định như thế nào ?

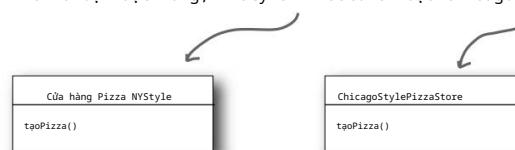


Bây giờ, để đi xa hơn một chút, phương thức `orderPizza()` thực hiện rất nhiều thứ với đối tượng Pizza (như chuẩn bị, nướng, cắt, đóng hộp), nhưng vì Pizza là trừu tượng, nên `orderPizza()` không biết những lớp cụ thể thực sự nào có liên quan. Nói cách khác, nó bị tách rời!



`orderPizza()` gọi `createPizza()` để thực sự lấy một đối tượng pizza. Nhưng nó sẽ lấy loại pizza nào? Phương thức `orderPizza()` không thể quyết định; nó không biết cách. Vậy ai quyết định?

Khi `orderPizza()` gọi `createPizza()`, một trong các lớp con của bạn sẽ được gọi vào hành động để tạo ra một chiếc pizza. Loại pizza nào sẽ được làm? Vâng, điều đó được quyết định bởi sự lựa chọn cửa hàng pizza mà bạn đặt hàng, `NYStylePizzaStore` hoặc `ChicagoStylePizzaStore`.



Vậy, có quyết định thời gian thực nào mà các lớp con đưa ra không? Không, nhưng theo quan điểm của `orderPizza()`, nếu bạn chọn `NYStylePizzaStore`, lớp con đó sẽ xác định loại pizza nào được làm. Vì vậy, các lớp con không thực sự "quyết định" - chính bạn là người quyết định bằng cách chọn cửa hàng bạn muốn - nhưng chúng xác định loại pizza nào được làm.

Hãy cùng làm một PizzaStore

Trở thành một đơn vị nhượng quyền có những lợi ích riêng. Bạn nhận được tất cả các chức năng của PizzaStore miễn phí. Tất cả những gì các cửa hàng khu vực cần làm là phân lớp PizzaStore và cung cấp phương thức `createPizza()` để triển khai kiểu Pizza của họ. Chúng tôi sẽ xử lý ba kiểu pizza lớn cho bên nhượng quyền.

Sau đây là phong cách vùng New York:

`createPizza()` trả về một Pizza và lớp con chịu trách nhiệm hoàn toàn về Pizza cụ thể mà nó khởi tạo

NYPizzaStore mở rộng PizzaStore, do đó nó kế thừa phương thức `orderPizza()` (cùng với các phương thức khác).

```
lớp công khai NYPizzaStore mở rộng PizzaStore {
    Pizza createPizza(Chuỗi mục) {
        nếu (item.equals("cheese")) {
            trả về NYStyleCheesePizza() mới;
        } else if (item.equals("veggie")) {
            trả về NYStyleVeggiePizza() mới;
        } else if (item.equals("clam")) {
            trả về NYStyleClamPizza() mới;
        } else if (item.equals("pepperoni")) {
            trả về NYStylePepperoniPizza() mới;
        } nếu không thì trả về null;
    }
}
```

← Chúng ta phải triển khai `createPizza()` vì nó là trừu tượng trong PizzaStore.

← Đây là nơi chúng ta tạo các lớp cụ thể. Đối với mỗi loại Pizza, chúng ta tạo ra kiểu NY.

* Lưu ý rằng phương thức `orderPizza()` trong lớp siêu lớp không biết chúng ta đang tạo loại Pizza nào; nó chỉ biết rằng nó có thể chuẩn bị, nướng, cắt và đóng hộp loại Pizza đó!

Sau khi chúng ta xây dựng xong các lớp con PizzaStore, đã đến lúc xem xét việc đặt một hoặc hai chiếc pizza. Nhưng trước khi làm điều đó, tại sao bạn không thử xây dựng các cửa hàng pizza Chicago Style và California Style ở trang tiếp theo.

phương pháp nhà máy



Chuốt bút chì của bạn

Chúng tôi đã khai trương NYPizzaStore, chỉ cần mở thêm hai cửa hàng nữa là chúng tôi sẽ sẵn sàng nhượng quyền!

Viết các triển khai PizzaStore tại Chicago và California ở đây:

Khai báo phương thức nhà máy

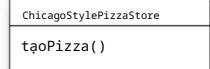
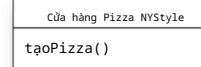
Chỉ với một vài lần chuyển đổi đối với PizzaStore, chúng ta đã chuyển từ việc có một đối tượng xử lý việc khởi tạo các lớp cụ thể của mình sang một tập hợp các lớp con hiện đang đảm nhiệm trách nhiệm đó. Hãy cùng xem kỹ hơn:

```

lớp truu tuong công khai PizzaStore {
    công khai Pizza orderPizza(String type) {
        Pizza, pizza;
        pizza = createPizza(kieu);
        pizza.chuan bi();
        pizza.nuong();
        pizza.cat();
        pizza.hop();
        trả lại pizza;
    }
    được bảo vệ truu tuong Pizza createPizza(String type);
    // các phương pháp khác ở đây
}

```

Các lớp con của PizzaStore xử lý việc khởi tạo đối tượng cho chúng ta trong phương thức createPizza().



Toàn bộ trách nhiệm khởi tạo Pizza đã được chuyển vào một phương thức hoạt động như một nhà máy.



Mã Gắn

Phương thức nhà máy xử lý việc tạo đối tượng và đóng gói nó trong một lớp con. Điều này tách mã máy khách trong lớp cha khỏi mã tạo đối tượng trong lớp con.

truu tuong Sản phẩm factoryMethod(Kieu chuỗi)

Phương thức nhà máy là truu tuong nên các lớp con được sử dụng để xử lý việc tạo đối tượng.

Phương thức nhà máy trả về một Sản phẩm thường được sử dụng trong các phương thức được xác định trong lớp siêu lớp.

Phương pháp nhà máy có thể được tham số hóa (hoặc không) để lựa chọn giữa nhiều biến thể của một sản phẩm.

Phương thức nhà máy cô lập máy khách (mã trong siêu lớp, như orderPizza()) khỏi việc biết loại Sản phẩm cụ thể nào thực sự được tạo ra.

đặt một chiếc bánh pizza

Hãy cùng xem cách thức hoạt động của nó: đặt pizza
bằng phương pháp pizza factory



Vậy họ sắp xếp thẻ nào?

- 1 Đầu tiên, Joel và Ethan cần một thể hiện của PizzaStore. Joel cần khởi tạo một ChicagoPizzaStore và Ethan cần một NYPizzaStore.
- 2 Với PizzaStore trong tay, cả Ethan và Joel đều gọi phương thức orderPizza() và truyền vào loại pizza họ muốn (phô mai, rau củ, v.v.).
- 3 Để tạo pizza, phương thức createPizza() được gọi, được định nghĩa trong hai lớp con NYPizzaStore và ChicagoPizzaStore. Theo như chúng tôi định nghĩa, NYPizzaStore khởi tạo một pizza kiểu NY và ChicagoPizzaStore khởi tạo pizza kiểu Chicago. Trong cả hai trường hợp, Pizza được trả về phương thức orderPizza().
- 4 Phương thức orderPizza() không biết loại pizza nào đã được tạo ra, nhưng nó biết đó là pizza và chuẩn bị, nướng, cắt và đóng hộp pizza cho Ethan và Joel.

mẫu nhà máy

Hãy cùng xem những chiếc pizza này thực sự được làm theo yêu cầu như thế nào nhé...

Hậu trường

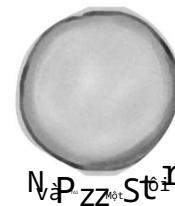


1

Chúng ta hãy làm theo thứ tự của Ethan: đầu tiên chúng ta cần một NY PizzaStore:

```
PizzaStore nyPizzaStore = new NYPizzaStore();
```

Tạo một phiên bản của NYPizzaStore.



2

Bây giờ chúng ta đã có cửa hàng, chúng ta có thể nhận đơn hàng:

```
nyPizzaStore.orderPizza("phô mai");
```

Phương thức orderPizza() được gọi trên thẻ hiện nyPizzaStore (phương thức được định nghĩa bên trong PizzaStore chạy).



3

Sau đó phương thức orderPizza() gọi phương thức createPizza():

```
Pizza pizza = createPizza("phô mát");
```

Hãy nhớ rằng, createPizza(), phương thức nhà máy, được triển khai trong lớp con. Trong trường hợp này, nó trả về NY Cheese Pizza.



4

Đối cùng, chúng ta có chiếc pizza chưa chế biến trong tay và phương thức orderPizza() hoàn tất việc chế biến nó:

```
pizza.chuẩn bị();
pizza. nướng();
pizza. cắt();
pizza. hộp();
```

Tất cả các phương thức này đều được định nghĩa trong pizza cụ thể được trả về từ phương thức createPizza() của nhà máy, được định nghĩa trong NYPizzaStore.

Phương thức orderPizza() trả về một Pizza mà không cần biết chính xác đó là lớp cụ thể nào.

các lớp học làm pizza

Chúng ta chỉ còn thiếu một thứ: PIZZA!

PizzaStore của chúng tôi sẽ không thể nổi tiếng nếu không có một số loại pizza, vậy hãy cùng thực hiện chúng nhé:



Chúng ta sẽ bắt đầu với lớp Pizza
trừu tượng và tất cả các loại pizza
cụ thể sẽ bắt nguồn từ lớp này.

```
lớp trừu tượng công khai Pizza {
    Tên chuỗi;
    Bột nhào;
    Nước sốt sợi;
    Phần mở đầu của ArrayList = new ArrayList();

    void chuẩn bị() {
        System.out.println("Đang chuẩn bị           + tên");
        " System.out.println("Đang nhào bột...");
        System.out.println("Đang thêm nước sốt...");
        System.out.println("Đang thêm lớp phủ: ");
        đối với (int i = 0; i < toppings.size(); i++) {
            System.out.println(" " + toppings.get(i));
        }
    }

    void nướng() {
        System.out.println("Nướng trong 25 phút ở nhiệt độ 350");
    }

    void cắt() {
        System.out.println("Cắt pizza thành các lát chéo");
    }

    hộp trống() {
        System.out.println("Đặt pizza vào hộp PizzaStore chính thức");
    }

    công khai String getName() {
        trả về tên;
    }
}
```

Mỗi chiếc Pizza đều có tên, loại bột,
loại nước sốt và một bộ đồ ăn kèm.

Lớp trừu tượng cung cấp
một số mặc định cơ bản để
nướng, cắt và đóng hộp.

Việc chuẩn bị bao
gồm một số bước
theo trình tự cụ thể.

NHỚ RẰNG: chúng tôi không cung cấp các câu lệnh import và package
trong danh sách mã. Lấy mã nguồn đầy đủ từ trang web headfirstlabs.
Bạn sẽ tìm thấy URL trên trang xxxiii trong phần Giới thiệu.

mẫu nhà máy

Bây giờ chúng ta chỉ cần một số lớp con cụ thể... thế còn việc định nghĩa pizza phô mai kiểu New York và Chicago thì sao?

```
lớp công khai NYStyleCheesePizza mở rộng Pizza {
    public NYStyleCheesePizza() { name = "Pizza
        phô mai và nước sốt kiểu NY";
        bột = "Bột vỏ mỏng";
        nước sốt = "Nước sốt Marinara";
        toppings.add("Phô mai Reggiano bào");
    }
}
```

Pizza NY có nước sốt marinara riêng và lớp vỏ mỏng.

Và một lớp phủ nữa là phô mai reggiano!

```
lớp công khai ChicagoStyleCheesePizza mở rộng Pizza {
    công khai ChicagoStyleCheesePizza() {
        tên = "Pizza phô mai đế dày kiểu Chicago";
        bột = "Bột vỏ siêu dày";
        sauce = "Sốt cà chua mặn";
        toppings.add("Phô mai Mozzarella bào sợi");
    }
}

void cắt() {
    System.out.println("Cắt pizza thành từng miếng vuông");
}
```

Pizza Chicago sử dụng cà chua mặn làm nước sốt cùng với lớp vỏ dày hơn.

 Pizza đế dày kiểu Chicago có rất nhiều phô mai mozzarella!

Pizza kiểu Chicago cũng ghi đè phương thức cut() để cắt các miếng pizza thành hình vuông.

làm một số chiếc bánh pizza

Bạn đã đợi đủ lâu rồi, đến giờ ăn pizza rồi!

```

lớp công khai PizzaTestDrive {

    public static void main(String[] args) {
        PizzaStore nyStore = new NYPizzaStore();
        PizzaStore chicagoStore = new ChicagoPizzaStore();

        Pizza pizza = nyStore.orderPizza("phô mai");
        System.out.println("Ethan đã đặt hàng một " + pizza.getName() + "\n");

        pizza = chicagoStore.orderPizza("phô mai");
        System.out.println("Joel đã đặt hàng một " + pizza.getName() + "\n");
    }
}

```

Đầu tiên chúng ta tạo
hai cửa hàng khác nhau.
Sau đó sử dụng một cửa hàng
để thực hiện đơn hàng của Ethan.
Và cái còn lại dành cho Joel.

Cửa sổ chính sửa tệp Trợ giúp Bạn muốn MootzOnThatPizza không?

%java PizzaTestDrive

```

Chuẩn bị Pizza sốt và phô mai kiểu New York
Trộn bột...
Thêm nước sốt...
Thêm lớp phủ: Phô mai
    Regiano bào
Nướng trong 25 phút ở nhiệt độ 350
Cắt pizza thành từng lát chéo
Đặt pizza vào hộp PizzaStore chính thức
Ethan đã gọi một chiếc Pizza sốt và phô mai kiểu New York

Chuẩn bị Pizza phô mai dẻ dày kiểu Chicago
Trộn bột...
Thêm nước sốt...
Thêm topping: Phô mai
    Mozzarella bào sợi
Nướng trong 25 phút ở nhiệt độ 350
Cắt pizza thành từng miếng vuông
Đặt pizza vào hộp PizzaStore chính thức
Joel đã gọi một chiếc bánh pizza phô mai dẻ dày kiểu Chicago

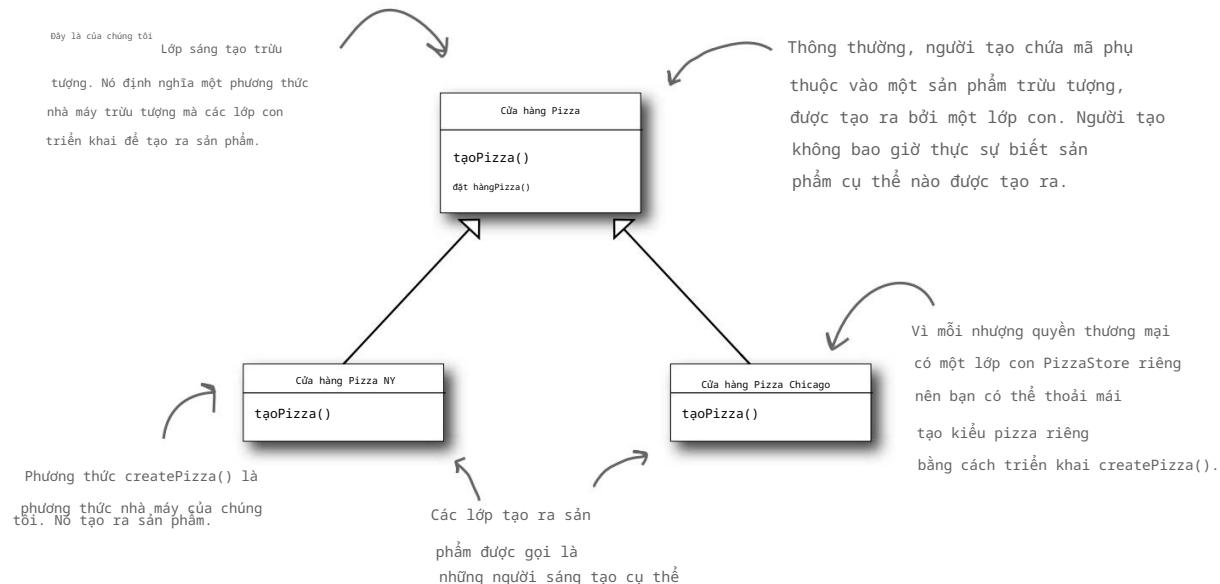
```

Cả hai loại pizza đều được
chuẩn bị, thêm lớp phủ, nướng,
cắt và đóng hộp.
Lớp cha của chúng ta không bao
giờ cần biết chi tiết, lớp con xử
 lý tất cả những điều đó chỉ
 bằng cách tạo ra đúng chiếc pizza.

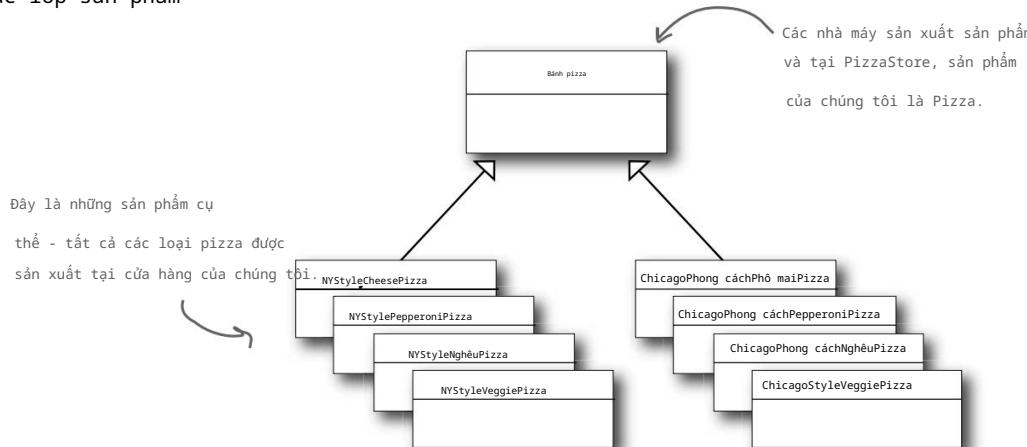
Cuối cùng đã đến lúc gấp Factory Method Pattern

Tất cả các mẫu nhà máy đều đóng gói việc tạo đối tượng. Mẫu phương thức nhà máy đóng gói việc tạo đối tượng bằng cách để các lớp con quyết định đối tượng nào sẽ tạo. Hãy cùng xem các sơ đồ lớp này để xem ai là người chơi trong mẫu này:

Các lớp học của Creator



Các lớp sản phẩm

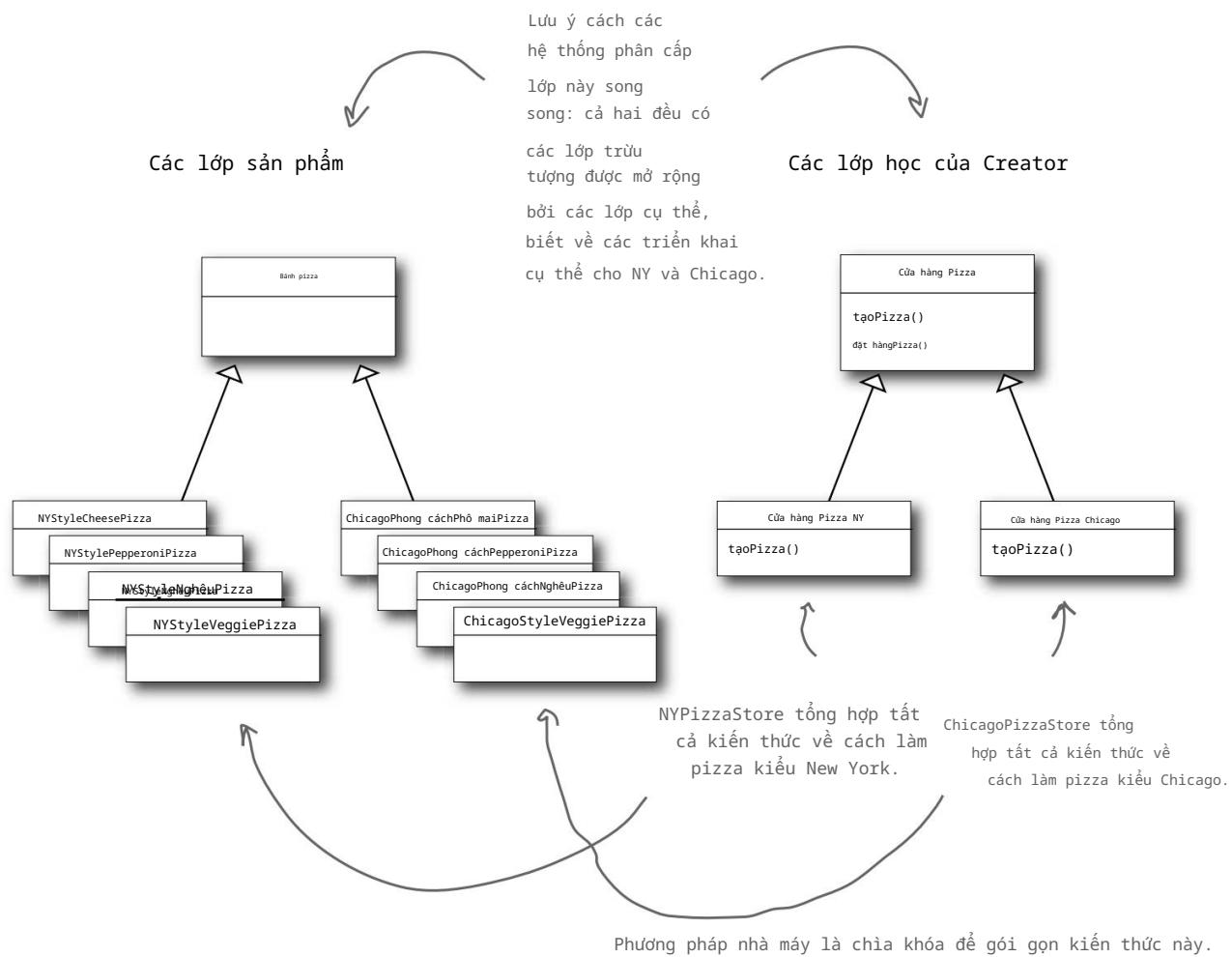


người sáng tạo và sản phẩm

Một góc nhìn khác: hệ thống phân cấp lớp song song

Chúng ta đã thấy rằng phương thức factory cung cấp một khuôn khổ bằng cách cung cấp phương thức `orderPizza()` được kết hợp với phương thức factory. Một cách khác để xem mô hình này như một khuôn khổ là cách nó đóng gói kiến thức sản phẩm vào từng người sáng tạo.

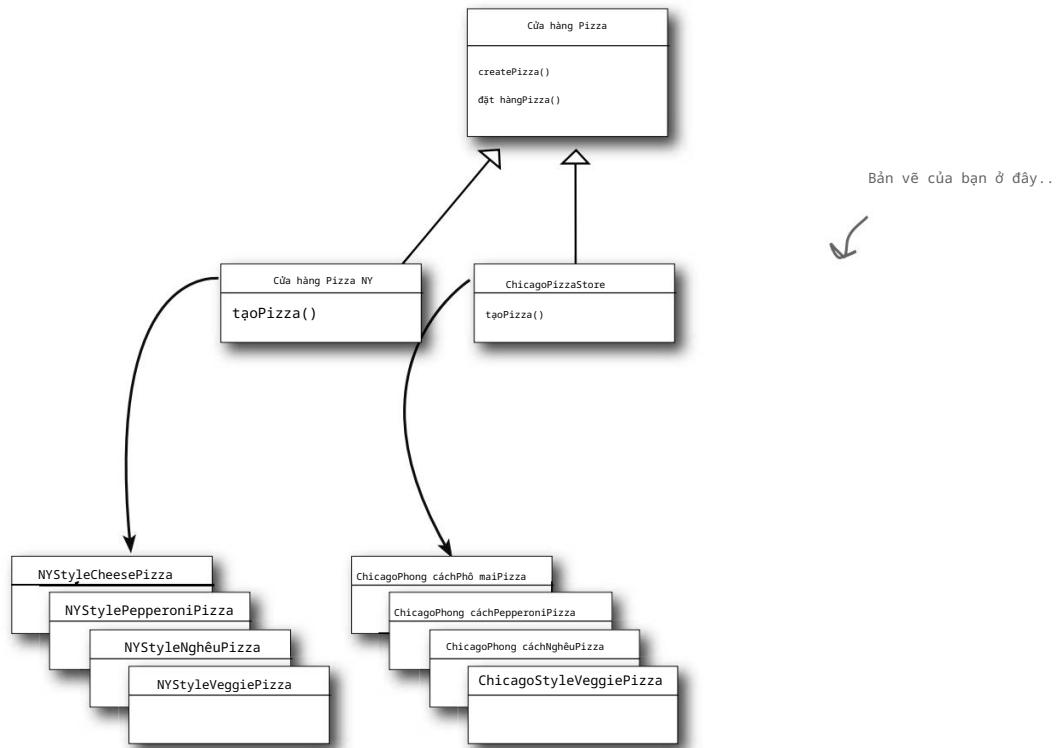
Hãy cùng xem xét hai hệ thống phân cấp lớp song song và xem chúng liên quan như thế nào





Thiết kế câu đố

Chúng ta cần một loại pizza khác dành cho những người California điên rồ (điên theo nghĩa tốt). Vẽ một tập hợp các lớp song song khác mà bạn cần để thêm một vùng California mới vào PizzaStore của chúng ta.



Bản vẽ của bạn ở đây...

Được rồi, bây giờ hãy viết ra năm thứ lợ nhất mà bạn có thể nghĩ ra để cho vào bánh pizza. Sau đó, bạn sẽ sẵn sàng bắt tay vào kinh doanh làm pizza ở California!

bạn đang ở đây 4 133

phương pháp nhà máy được định nghĩa

Mẫu phương pháp nhà máy được xác định

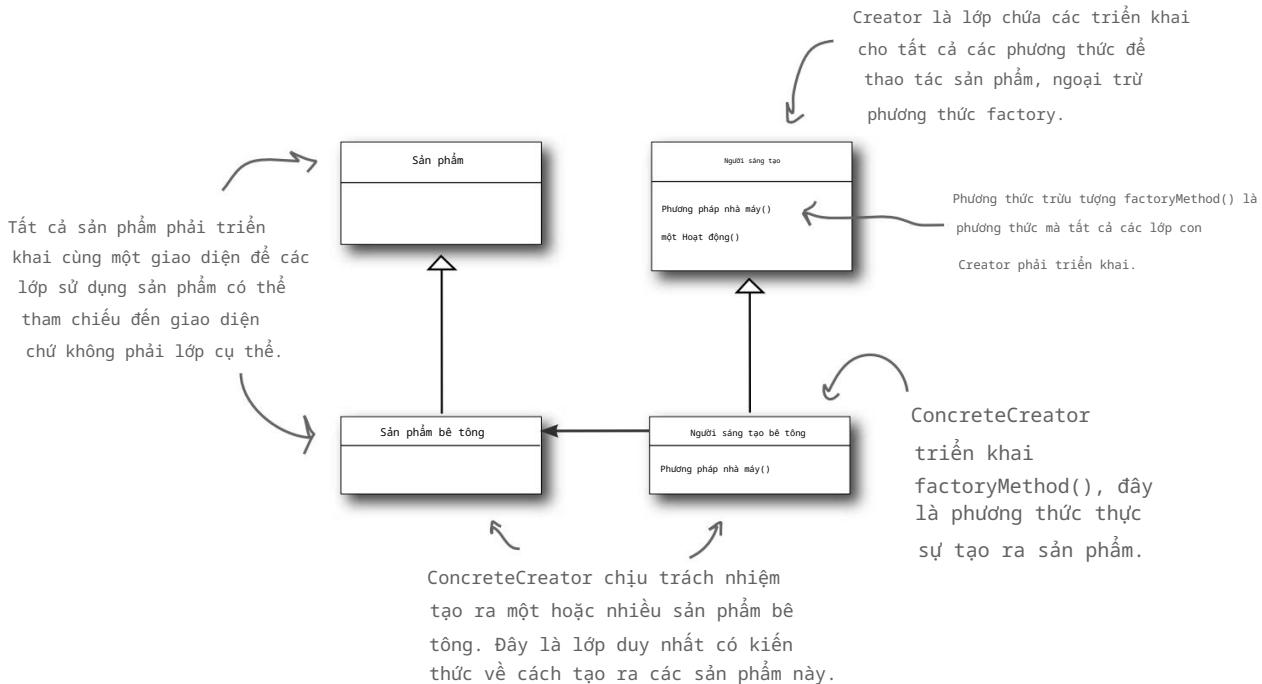
Đã đến lúc đưa ra định nghĩa chính thức về Mô hình phương pháp nhà máy:

Mẫu Factory Method định nghĩa một giao diện để tạo đối tượng, nhưng cho phép các lớp con quyết định lớp nào sẽ khởi tạo. Factory Method cho phép một lớp hoàn việc khởi tạo cho các lớp con.

Giống như mọi nhà máy, Factory Method Pattern cung cấp cho chúng ta một cách để đóng gói các thể hiện của các kiểu cụ thể. Khi xem sơ đồ lớp bên dưới, bạn có thể thấy rằng Creator trùu tượng cung cấp cho bạn một giao diện với một phương thức để tạo đối tượng, còn được gọi là "phương thức factory". Bất kỳ phương thức nào khác được triển khai trong Creator trùu tượng đều được viết để vận hành trên các sản phẩm do phương thức factory tạo ra. Chỉ có các lớp con thực sự triển khai phương thức factory và tạo sản phẩm.

Như trong định nghĩa chính thức, bạn thường nghe các nhà phát triển nói rằng Factory Method cho phép các lớp con quyết định lớp nào sẽ khởi tạo. Họ nói "quyết định" không phải vì mẫu cho phép chính các lớp con quyết định tại thời điểm chạy, mà vì lớp người sáng tạo được viết mà không biết về các sản phẩm thực tế sẽ được tạo ra, mà được quyết định hoàn toàn bởi sự lựa chọn của lớp con được sử dụng.

Bạn có thể hỏi họ "quyết định" có nghĩa là gì, nhưng chúng tôi cá là bây giờ bạn hiểu điều này rõ hơn họ!



không có câu hỏi ngớ ngẩn nào

Q: Ưu điểm của phương pháp nhà máy là gì?

Mẫu khi bạn chỉ có một ConcreteCreator?

A: Mô hình Factory Method hữu ích nêu

bạn chỉ có một trình tạo cụ thể vì bạn đang tách việc triển khai sản phẩm khỏi việc sử dụng nó. Nếu bạn thêm các sản phẩm bổ sung hoặc thay đổi việc triển khai sản phẩm, điều đó sẽ không ảnh hưởng đến Trình tạo của bạn (ví Trình tạo không được liên kết chặt chẽ với bất kỳ ConcreteProduct nào).

H: Có đúng không khi nói rằng NY và

Các cửa hàng ở Chicago được triển khai bằng Simple Factory? Trong chúng giống hệt vậy.

A: Chúng tương tự nhau, nhưng được sử dụng theo những cách khác nhau. Ngay cả mặc dù việc triển khai mỗi kho lưu trữ cụ thể trong rất giống SimplePizzaFactory, hãy nhớ rằng các kho lưu trữ cụ thể đang mở rộng một lớp đã định nghĩa createPizza() là một phương thức trừu tượng. Mỗi kho lưu trữ phải tự định nghĩa hành vi của phương thức createPizza().

Trong Simple Factory, nhà máy là một đối tượng khác được tạo thành từ PizzaStore.

H: Phương pháp nhà máy và Đóng sáng tạo có phải là

luôn trúu tượng?

A: Không, bạn có thể xác định một phương thức mặc định của nhà máy để sản xuất một số sản phẩm cụ thể. Khi đó, bạn luôn có phương tiện để tạo ra sản phẩm ngay cả khi không có lớp con nào của Đóng sáng tạo.

H: Mỗi cửa hàng có thể sản xuất bốn loại khác nhau của các loại pizza dựa trên loại được đưa vào. Tất cả những người sáng tạo cụ thể có tạo ra nhiều sản phẩm không, hay đôi khi họ chỉ tạo ra một sản phẩm?

A: Chúng tôi đã thực hiện cái được gọi là

phương thức nhà máy tham số hóa. Nó có thể tạo ra nhiều hơn một đối tượng dựa trên một tham số được truyền vào, như bạn đã thấy. Tuy nhiên, thông thường, một nhà máy chỉ tạo ra một đối tượng và không được tham số hóa. Cả hai đều là dạng hợp lệ của mẫu.

H: Các kiểu tham số của bạn có vẻ không phải là "kiểu-an toàn." Tôi chỉ đang truyền một String! Nếu tôi yêu cầu một "CalmPizza" thì sao?

A: Bạn chắc chắn đúng và điều đó sẽ gây ra,

cái mà chúng ta gọi trong kinh doanh là "lỗi thời gian chạy". Có một số kỹ thuật tinh vi hơn khác có thể được sử dụng để làm cho các tham số "an toàn kiểu" hơn, hay nói cách khác, để đảm bảo các lỗi trong tham số có thể được phát hiện tại thời điểm biên dịch. Ví dụ, bạn có thể tạo các đối tượng biểu diễn các kiểu tham số, sử dụng hàng số tĩnh hoặc, trong Java 5, bạn có thể sử dụng enum.

H: Tôi vẫn còn hơi bối rối về sự khác biệt

giữa Simple Factory và Factory Method. Chúng trông rất giống nhau, ngoại trừ trong Factory Method, lớp trả về pizza là một lớp con. Bạn có thể giải thích không?

A: Bạn nói đúng là các lớp con trông rất giống nhau

giống như Simple Factory, tuy nhiên hãy nghĩ về Simple Factory như một thỏa thuận một lần, trong khi với Factory Method, bạn đang tạo một khuôn khổ cho phép các lớp con quyết định triển khai nào sẽ được sử dụng. Ví dụ, phương thức orderPizza() trong Factory Method cung cấp một khuôn khổ chung để tạo pizza dựa trên phương thức factory để thực sự tạo ra các lớp cụ thể đi vào việc làm pizza. Bằng cách phân lớp lớp PizzaStore, bạn quyết định những sản phẩm cụ thể nào sẽ đi vào việc làm pizza mà orderPizza() trả về.

So sánh với SimpleFactory, phương pháp này cung cấp cho bạn cách đóng gói việc tạo đối tượng, nhưng không cung cấp cho bạn linh hoạt của Factory Method vì không có cách nào để thay đổi các sản phẩm bạn đang tạo.

thầy và trò



Thầy và trò...

Sư phụ: Cháu cháu, hãy cho ta biết tiến độ luyện tập của con thế nào?

Học viên: Thưa thầy, con đã học xong bài "bao hàm những gì" thay đổi" hơn nữa.

Sư phụ: Tiếp tục đi...

Học viên: Em đã học được rằng người ta có thể đóng gói mã tạo ra các đối tượng.

Khi bạn có mã khởi tạo các lớp cụ thể, đây là một lĩnh vực thường xuyên thay đổi. Em đã học được một kỹ thuật gọi là "factory" cho phép bạn đóng gói hành vi khởi tạo này.

Sư phụ: Và những "nhà máy" này có lợi ích gì?

Sinh viên: Có nhiều. Bằng cách đặt tất cả mã tạo của tôi vào một đối tượng hoặc phương thức, tôi tránh được sự trùng lặp trong mã của mình và cung cấp một nơi để thực hiện bảo trì. Điều đó cũng có nghĩa là khách hàng chỉ phụ thuộc vào giao diện chứ không phải các lớp cụ thể cần thiết để khởi tạo đối tượng. Như tôi đã học trong quá trình học, điều này cho phép tôi lập trình theo giao diện chứ không phải triển khai và điều đó làm cho mã của tôi linh hoạt và có thể mở rộng hơn trong tương lai.

Master: Vâng Grasshopper, bản năng OO của bạn đang phát triển. Bạn có câu hỏi nào cho master của bạn ngày hôm nay không?

Học viên: Thưa thầy, em biết rằng bằng cách đóng gói việc tạo đối tượng, em đang mã hóa thành các lớp trừu tượng và tách mã máy khách của em khỏi các triển khai thực tế. Nhưng mã nhà máy của em vẫn phải sử dụng các lớp cụ thể để khởi tạo các đối tượng thực. Em không phải đang tự lừa mình sao?

Master: Grasshopper, việc tạo đối tượng là một thực tế của cuộc sống; chúng ta phải tạo đối tượng nếu không chúng ta sẽ không bao giờ tạo ra một chương trình Java nào. Nhưng, với kiến thức về thực tế này, chúng ta có thể thiết kế mã của mình sao cho chúng ta đã dồn được mã sáng tạo này lại như con cùu mà bạn muốn kéo lông che mắt mình. Một khi đã dồn được lại, chúng ta có thể bảo vệ và chăm sóc mã sáng tạo. Nếu chúng ta để mã sáng tạo của mình chạy lung tung, thì chúng ta sẽ không bao giờ thu thập được "lông" của nó.

Học viên: Thưa thầy, con thấy điều này là đúng.

Sư phụ: Như tôi biết ông sẽ làm. Vậy giờ, hãy đi và thiền về sự phụ thuộc của đối tượng.

Một PizzaStore rất phụ thuộc



Chuốt bút chì của bạn

Giả sử bạn chưa từng nghe đến nhà máy OO. Đây là phiên bản PizzaStore không sử dụng nhà máy; hãy đếm số lượng đối tượng pizza cụ thể mà lớp này phụ thuộc vào. Nếu bạn thêm pizza kiểu California vào PizzaStore này, thì nó sẽ phụ thuộc vào bao nhiêu đối tượng?

```

lớp công khai DependentPizzaStore {

    public Pizza createPizza(String style, String type) { Pizza pizza =
        null; if
            (style.equals("NY")) { if
                (type.equals("cheese")) { pizza =
                    new NYStyleCheesePizza();
                } else if (type.equals("veggie")) {
                    pizza = new NYStyleVeggiePizza();
                } else if (type.equals("clam")) {
                    pizza = new NYStyleClamPizza(); } else
                if (type.equals("pepperoni"))
                    pizza = new NYStylePepperoniPizza();
                }
            } else if (style.equals("Chicago")) { if
                (type.equals("cheese")) { pizza =
                    new ChicagoStyleCheesePizza();
                } else if (type.equals("veggie")) {
                    pizza = new ChicagoStyleVeggiePizza(); } else if
                (type.equals("clam")) {
                    pizza = new ChicagoStyleClamPizza();
                } else if (type.equals("pepperoni"))
                    pizza = new ChicagoStylePepperoniPizza();

            } } else
            { System.out.println("Lỗi: loại pizza không hợp lệ"); return null;

        } pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
}

```

Xử lý tất cả các loại
pizza kiểu NY

Xử lý tất cả các
Pizza kiểu
Chicago

Bạn có thể viết
câu trả lời của mình ở đây:

con só

số với California nữa

bạn đang ở đây 4 137

phụ thuộc đối tượng

Xem xét các phụ thuộc đối tượng

Khi bạn trực tiếp khởi tạo một đối tượng, bạn phụ thuộc vào lớp cụ thể của nó. Hãy xem PizzaStore rất phụ thuộc của chúng tôi ở một trang trước. Nó tạo ra tất cả các đối tượng pizza ngay trong lớp PizzaStore thay vì ủy quyền cho một nhà máy.

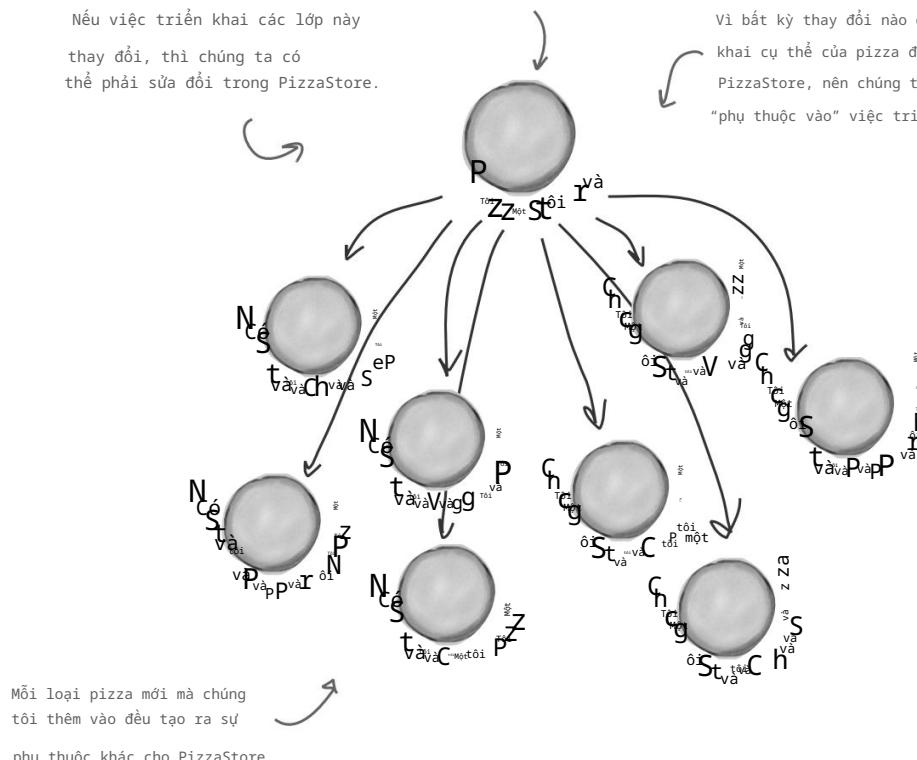
Nếu chúng ta vẽ sơ đồ thể hiện phiên bản PizzaStore đó và tất cả các đối tượng mà nó phụ thuộc, thì sơ đồ trông như thế này:

Phiên bản

PizzaStore này phụ thuộc
vào tất cả các đối tượng
pizza vì nó tạo ra chúng trực tiếp.

Nếu việc triển khai các lớp này
thay đổi, thì chúng ta có
thể phải sửa đổi trong PizzaStore.

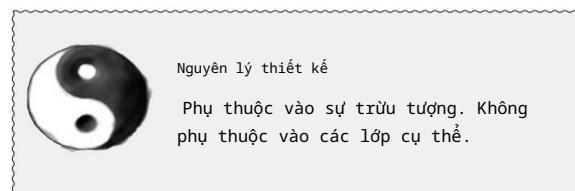
Vì bất kỳ thay đổi nào đối với việc triển
khai cụ thể của pizza đều ảnh hưởng đến
PizzaStore, nên chúng ta nói rằng PizzaStore
“phụ thuộc vào” việc triển khai pizza.



Nguyên lý đảo ngược sự phụ thuộc

Có thể thấy rõ rằng việc giảm sự phụ thuộc vào các lớp cụ thể trong mã của chúng ta là một "điều tốt". Trên thực tế, chúng ta có một nguyên tắc thiết kế hướng đối tượng chính thức hóa khái niệm này; nó thậm chí còn có một cái tên lớn và chính thức: Nguyên tắc đảo ngược sự phụ thuộc.

Sau đây là nguyên tắc chung:



Thoạt đầu, nguyên tắc này nghe có vẻ giống như "Lập trình cho một giao diện, không phải là một triển khai", đúng không? Nó tương tự; tuy nhiên, Nguyên tắc đảo ngược phụ thuộc đưa ra một tuyên bố thận chí còn mạnh mẽ hơn về sự trừu tượng. Nó gợi ý rằng các thành phần cấp cao của chúng ta không nên phụ thuộc vào các thành phần cấp thấp của chúng ta; thay vào đó, cả hai đều nên phụ thuộc vào sự trừu tượng.

Nhưng điều đó có nghĩa là gì?

Vâng, chúng ta hãy bắt đầu bằng cách xem lại sơ đồ cửa hàng pizza ở trang trước. PizzaStore là "thành phần cấp cao" của chúng ta và các triển khai pizza là "thành phần cấp thấp" của chúng ta, và rõ ràng PizzaStore phụ thuộc vào các lớp pizza cụ thể.

Bây giờ, nguyên tắc này cho chúng ta biết rằng chúng ta nên viết mã sao cho phụ thuộc vào các lớp trừu tượng chứ không phải các lớp cụ thể. Điều này áp dụng cho cả các mô-đun cấp cao và các mô-đun cấp thấp của chúng ta.

Nhưng chúng ta thực hiện điều này như thế nào? Hãy cùng suy nghĩ về cách áp dụng nguyên tắc này vào việc triển khai Very Dependent PizzaStore của chúng ta...

Một cụm từ khác mà bạn có thể sử dụng để gây ấn tượng với các giám đốc điều hành trong phòng! Khoản tăng lương của bạn sẽ bù đắp nhiều hơn chi phí của cuốn sách này và bạn sẽ nhận được sự ngưỡng mộ của các nhà phát triển đồng nghiệp.

Thành phần "cấp cao" là lớp có hành vi được xác định theo các thành phần "cấp thấp" khác.

Ví dụ, PizzaStore là một thành phần cấp cao vì hành vi của nó được định nghĩa theo thuật ngữ pizza - nó tạo ra tất cả các đối tượng pizza khác nhau, chuẩn bị, nướng, cắt và đóng hộp chúng, trong khi các loại pizza mà nó sử dụng là các thành phần cấp thấp.

nguyên lý đảo ngược phụ thuộc

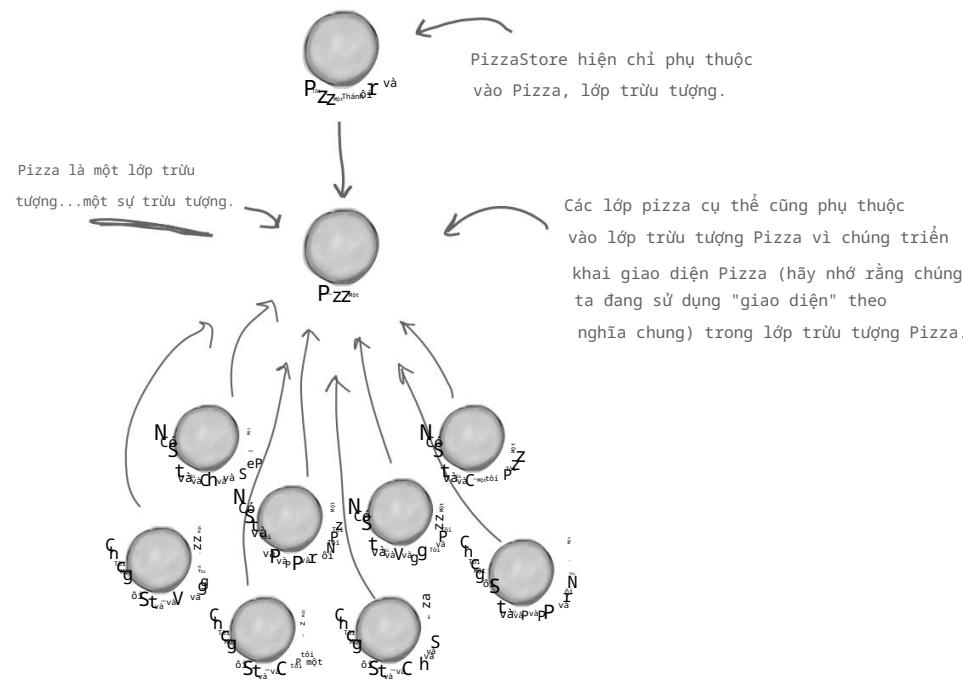
Áp dụng nguyên tắc

Bây giờ, vẫn đề chính với Very Dependent PizzaStore là nó phụ thuộc vào mọi loại pizza vì nó thực sự khởi tạo các kiểu cụ thể trong phương thức orderPizza() của nó.

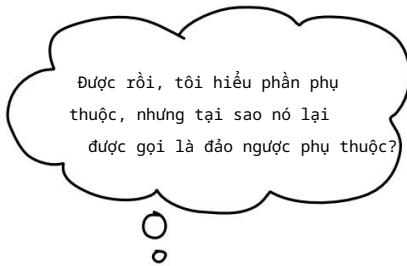
Mặc dù chúng ta đã tạo ra một lớp trừu tượng, Pizza, nhưng chúng ta vẫn đang tạo ra các loại Pizza cụ thể trong mã này, do đó chúng ta không tận dụng được nhiều lợi ích từ lớp trừu tượng này.

Làm sao chúng ta có thể lấy những thẻ hiện đó ra khỏi phương thức orderPizza()? Vâng, như chúng ta đã biết, Factory Method cho phép chúng ta làm điều đó.

Vì vậy, sau khi áp dụng Phương pháp Nhà máy, sơ đồ của chúng ta trông như thế này:



Sau khi áp dụng Factory Method, bạn sẽ nhận thấy rằng thành phần cấp cao của chúng ta, PizzaStore, và các thành phần cấp thấp của chúng ta, pizzas, đều phụ thuộc vào Pizza, sự trừu tượng. Factory Method không phải là kỹ thuật duy nhất để tuân thủ Nguyên tắc đảo ngược phụ thuộc, nhưng nó là một trong những kỹ thuật mạnh mẽ hơn.



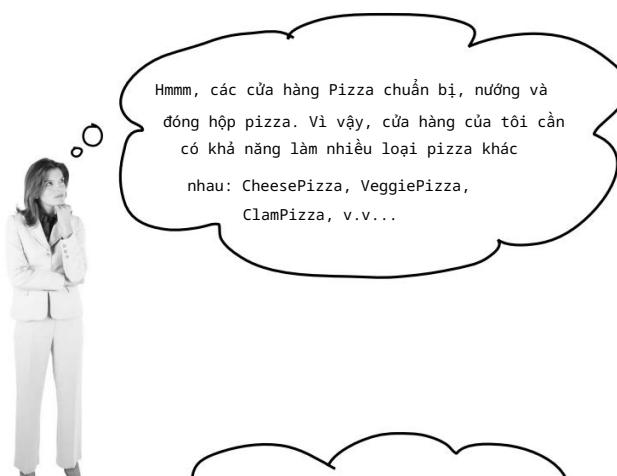
“Đảo ngược” trong Nguyên lý đảo ngược phụ thuộc ở
đâu?

“Đảo ngược” trong tên Nguyên tắc đảo ngược phụ
thuộc có ở đó vì nó đảo ngược cách bạn thường
nghĩ về thiết kế OO của mình. Hãy xem sơ đồ ở
trang trước, lưu ý rằng các thành phần cấp thấp
hiện phụ thuộc vào một mức trừu tượng cao hơn.
Tương tự như vậy, thành phần cấp cao cũng được
gắn với cùng một mức trừu tượng. Vì vậy, biểu
đồ phụ thuộc từ trên xuống dưới mà chúng ta đã vẽ
ở một vài trang trước đã tự đảo ngược, với cả các
mô-đun cấp cao và cấp thấp hiện phụ thuộc vào mức trừu tượng.

Chúng ta hãy cùng tìm hiểu suy nghĩ đằng sau quy trình thiết
kế thông thường và xem cách đưa ra nguyên tắc có thể đảo
ngược cách chúng ta suy nghĩ về thiết kế như thế nào...

đảo ngược suy nghĩ của bạn

Đảo ngược suy nghĩ của bạn...



Được rồi, vậy thì bạn cần triển khai một PizzaStore.

Ý nghĩ đầu tiên xuất hiện trong đầu bạn là gì?



Đúng vậy, bạn bắt đầu từ trên cùng và theo dõi mọi thứ xuống các lớp cụ thể. Nhưng, như bạn đã thấy, bạn không muốn cửa hàng của mình biết về các loại pizza cụ thể, vì khi đó nó sẽ phụ thuộc vào tất cả các lớp cụ thể đó!

Bây giờ, hãy "đảo ngược" suy nghĩ của bạn... thay vì bắt đầu từ trên xuống, hãy bắt đầu từ Pizza và nghĩ về những gì bạn có thể trừu tượng hóa.



Đúng rồi! Bạn đang nghĩ về Pizza trừu tượng. Vậy bây giờ, hãy quay lại và nghĩ về thiết kế của Cửa hàng Pizza một lần nữa.

Đóng. Nhưng để làm được điều đó, bạn sẽ phải dựa vào một nhà máy để lấy các lớp cụ thể đó ra khỏi Cửa hàng Pizza của bạn. Sau khi bạn đã làm điều đó, các loại pizza cụ thể khác nhau của bạn chỉ phụ thuộc vào một sự trừu tượng và cửa hàng của bạn cũng vậy. Chúng tôi đã lấy một thiết kế trong đó cửa hàng phụ thuộc vào các lớp cụ thể và đảo ngược các phụ thuộc đó (cùng với suy nghĩ của bạn).

Một số hướng dẫn giúp bạn tuân thủ Nguyên tắc...

Các hướng dẫn sau đây có thể giúp bạn tránh các thiết kế hướng đối tượng vi phạm Nguyên tắc đảo ngược phụ thuộc:

β Không có biến nào được phép giữ tham chiếu đến một lớp cụ thể.

Nếu bạn sử dụng `new`, bạn sẽ giữ một tham chiếu đến một lớp cụ thể.

Hãy sử dụng nhà máy để giải quyết vấn đề đó!

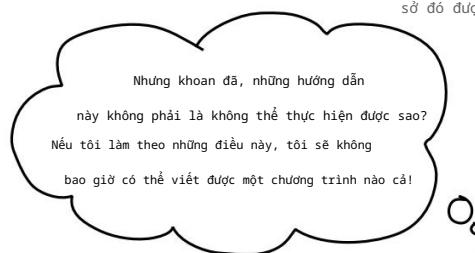
β Không có lớp nào được bắt nguồn từ một lớp cụ thể.

Nếu bạn bắt nguồn từ một lớp cụ thể, bạn sẽ phụ thuộc vào một lớp cụ thể. Xuất phát từ một sự trừu tượng, như một giao diện hoặc một lớp trừu tượng.

β Không có phương thức nào được ghi đè lên phương thức đã triển khai của bất kỳ lớp cơ sở nào của nó.

Nếu bạn ghi đè một phương thức đã triển khai, thì lớp cơ sở của bạn không thực sự là một lớp trừu tượng ngay từ đầu.

Các phương thức được triển khai trong lớp cơ sở đó được dùng chung cho tất cả các lớp con của bạn.



Bạn hoàn toàn đúng! Giống như nhiều nguyên tắc khác của chúng tôi, đây là hướng dẫn mà bạn nên phản ánh thực hiện, chứ không phải là quy tắc mà bạn phải tuân thủ mọi lúc.

Rõ ràng, mọi chương trình Java từng được viết đều vi phạm các nguyên tắc này!

Nhưng nếu bạn tiếp thu những nguyên tắc này và ghi nhớ chúng khi thiết kế, bạn sẽ biết khi nào mình vi phạm nguyên tắc và bạn sẽ có lý do chính đáng để làm như vậy. Ví dụ, nếu bạn có một lớp không có khả năng thay đổi và bạn biết điều đó, thì việc bạn khởi tạo một lớp cụ thể trong mã của mình không phải là tận thê. Hãy nghĩ về điều đó; chúng ta khởi tạo các đối tượng `String` mọi lúc mà không cần suy nghĩ nhiều.

Điều đó có vi phạm nguyên tắc không? Có. Có ổn không? Có. Tại sao? Bởi vì `String` rất khó có thể thay đổi.

Mặt khác, nếu lớp bạn viết có khả năng thay đổi, bạn có một số kỹ thuật tốt như Factory Method để đóng gói thay đổi đó.



các họ thành phần

Trong khi đó, quay lại PizzaStore...

Thiết kế của PizzaStore thực sự đang được hoàn thiện: có khuôn khổ linh hoạt và tuân thủ tốt các nguyên tắc thiết kế.

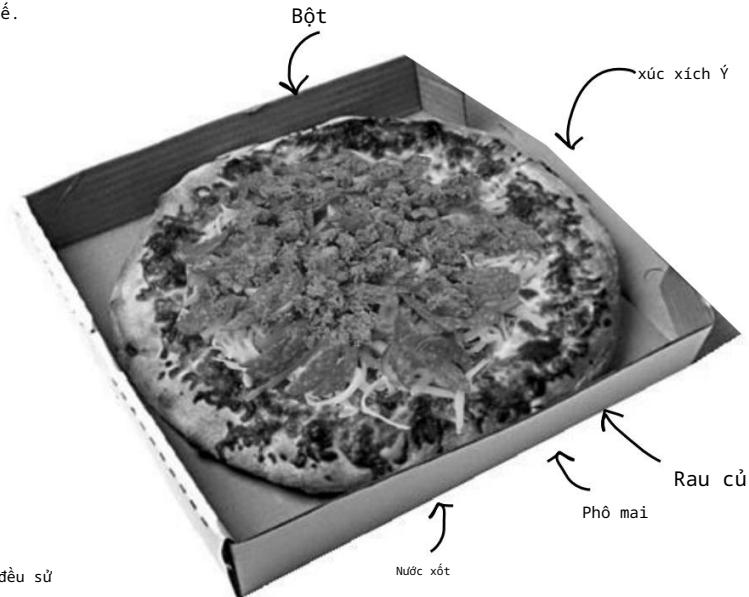
Bây giờ, chia khóa thành công của Objectville
Pizza luôn là nguyên liệu tươi, chất lượng,
và điều bạn phát hiện ra là với khuôn
khô mới, các cửa hàng nhượng quyền của bạn
đã tuân theo quy trình của bạn, nhưng một số
cửa hàng nhượng quyền đã thay thế các nguyên
liệu kém chất lượng trong bánh của họ để
giảm chi phí và tăng lợi nhuận. Bạn biết
mình phải làm gì đó, vì về lâu dài, điều này
sẽ gây tổn hại đến thương hiệu Objectville!

Đảm bảo tính nhất quán trong các
thành phần của bạn

Vậy làm sao bạn có thể đảm bảo mỗi cửa hàng nhượng quyền đều sử
dụng nguyên liệu chất lượng? Bạn sẽ xây dựng một nhà máy sản xuất
chung và vận chuyển đến các cửa hàng nhượng quyền của mình!

Hiện chỉ có một vấn đề với kế hoạch này: các cửa hàng nhượng quyền nằm ở các
khu vực khác nhau và món nước sốt đỏ ở New York không phải là món nước sốt đỏ ở Chicago.

Vì vậy, bạn có một bộ nguyên liệu cần được vận chuyển đến New York và một bộ khác
cần được vận chuyển đến Chicago. Chúng ta hãy xem xét kỹ hơn:





Chicago

Thực đơn Pizza

Pizza phô mai
Sốt cà chua mận, phô mai Mozzarella, Parmesan, Rau oregano

Pizza chay
Sốt cà chua mận, phô mai Mozzarella, Parmesan, Cà tím, Rau bina, Ô liu đen

Pizza Nghêu
Sốt cà chua mận, Mozzarella, Parmesan, Nghêu

Pizza xúc xích Ý
Sốt cà chua mận, phô mai Mozzarella, Parmesan, Cà tím, Rau bina, Ô liu đen, Pepperoni

Chúng tôi có
cùng một nhóm
sản phẩm (bột,
nước sốt, phô mai,
rau, thịt)
nhưng cách
thực hiện khác
nhau tùy theo khu vực.

New York

Thực đơn Pizza

Pizza phô mai
Sốt Marinara, Reggiano, Tỏi

Pizza chay
Sốt Marinara, Reggiano, Nấm, Hành tây, Ớt đỏ

Pizza Nghêu
Sốt Marinara, Reggiano, Nghêu tươi

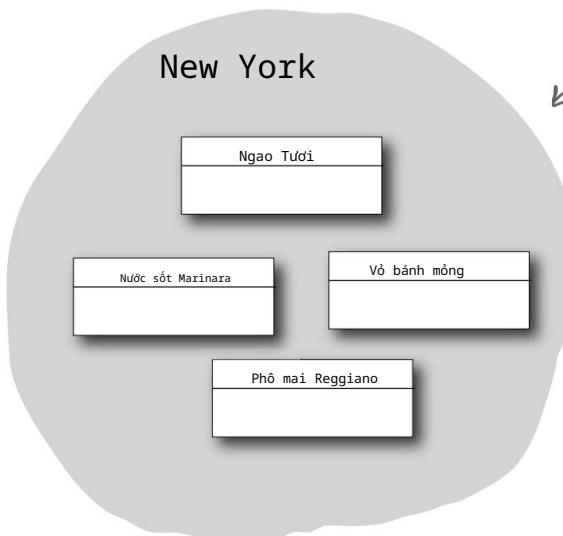
Pizza xúc xích Ý
Sốt Marinara, Reggiano, Nấm, Hành tây, Ớt đỏ, Pepperoni



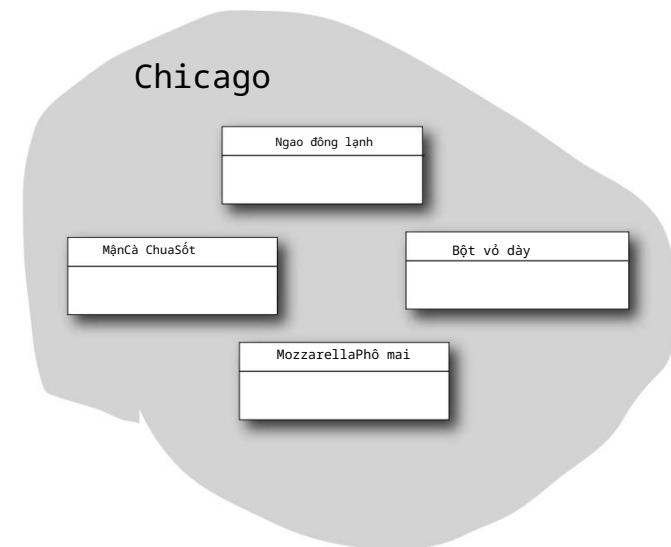
Các nhóm thành phần...

New York sử dụng một bộ nguyên liệu và Chicago sử dụng một bộ nguyên liệu khác. Với sự nổi tiếng của Objectville Pizza, sẽ không lâu nữa trước khi bạn cũng cần vận chuyển một loạt nguyên liệu địa phương khác đến California, và tiếp theo là gì? Seattle ư?

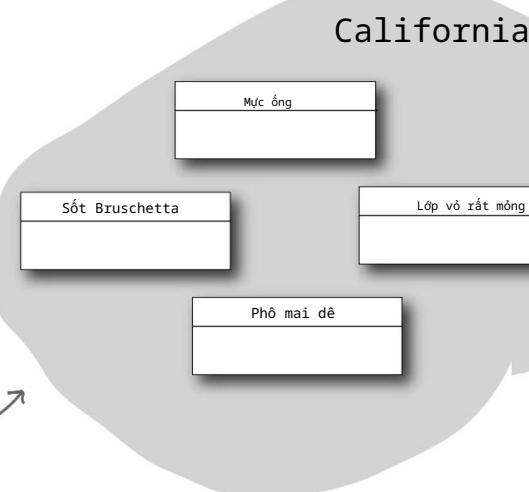
Để thực hiện được điều này, bạn sẽ phải tìm ra cách xử lý các nhóm nguyên liệu khác nhau.



Mỗi loại bao gồm một loại bột, một loại nước sốt, một loại phô mai và một loại hải sản phủ lên trên (cùng với một số loại khác mà chúng tôi chưa giới thiệu, như rau và gia vị).



Tất cả các loại pizza của Objectville đều được làm từ những thành phần giống nhau, nhưng mỗi vùng lại có cách thực hiện các thành phần đó khác nhau.



Tổng cộng, ba vùng này tạo nên các nhóm nguyên liệu, trong đó mỗi vùng sử dụng một nhóm nguyên liệu hoàn chỉnh.

nha máy nguyên liệu

Xây dựng nhà máy sản xuất nguyên liệu

Bây giờ chúng ta sẽ xây dựng một nhà máy để tạo ra các thành phần của mình; nhà máy sẽ chịu trách nhiệm tạo ra từng thành phần trong nhóm thành phần. Nói cách khác, nhà máy sẽ cần tạo ra bột, nước sốt, phô mai, v.v... Bạn sẽ thấy cách chúng ta xử lý các khác biệt theo vùng miền ngay sau đây.

Chúng ta hãy bắt đầu bằng cách xác định giao diện cho nhà máy sẽ tạo ra tất cả các thành phần của chúng ta:

```
giao diện công khai PizzaIngredientFactory {
```

```
    công khai Dough createDough();
    công khai Sauce createSauce();
    công khai Cheese createCheese();
    công khai Veggies[] createVeggies();
    công khai Pepperoni createPepperoni();
    công khai Clams createClam();
```

```
}
```

```
}
```

Có rất nhiều lớp học mới ở đây, mỗi lớp học dành cho một thành phần.



Đối với mỗi thành phần, chúng tôi định nghĩa một phương thức tạo trong giao diện của mình.

Nếu chúng ta có một số "máy móc" chung để triển khai trong mỗi trường hợp của nhà máy, chúng ta có thể biến nó thành một lớp trừu tượng...

Sau đây là những gì chúng ta sẽ làm:

- 1 Xây dựng một nhà máy cho mỗi vùng. Để thực hiện điều này, bạn sẽ tạo một lớp con của PizzaIngredientFactory triển khai từng phương thức tạo
- 2 Triển khai một tập hợp các lớp thành phần để sử dụng với nhà máy, như ReggianoCheese, RedPeppers và ThickCrustDough. Các lớp này có thể được chia sẻ giữa các vùng khi thích hợp.
- 3 Sau đó, chúng ta vẫn cần kết nối tất cả những thứ này bằng cách đưa nhà máy nguyên liệu mới vào mã PizzaStore cũ.

Xây dựng nhà máy nguyên liệu ở New York

Được rồi, đây là bản triển khai cho nhà máy nguyên liệu New York. Nhà máy này chuyên về sốt Marinara, phô mai Reggiano, nghêu tươi...

Nhà máy thành phần NY triển khai giao diện cho tất cả các thành phần nhà máy

```
lớp công khai NYPizzaIngredientFactory triển khai PizzaIngredientFactory {
```

```
    công khai Dough createDough() {
        trả về ThinCrustDough() mới;
    }
```

Đối với mỗi thành phần trong

```
    công khai Sauce createSauce() {
        trả về MarinaraSauce() mới;
    }
```

nhóm thành phần, chúng tôi

tạo ra phiên bản New York.

```
    công khai Cheese createCheese() {
        trả về ReggianoCheese mới();
    }
```



```
    công khai Veggies[] createVeggies() {
        Rau củ rau củ[] = { mới Tỏi(), mới Hành tây(), mới Nấm(), mới Ớt đỏ() };
        trả lại rau;
    }
```

Đối với rau, chúng ta trả về một mảng Veggies. Ở đây chúng ta đã mã hóa cùng các loại rau. Chúng ta có thể làm cho nó phức tạp hơn, nhưng điều đó không thực sự bổ sung bất cứ điều gì vào việc học mẫu máy, vì vậy chúng ta sẽ giữ cho nó đơn giản.

```
    công khai Pepperoni createPepperoni() {
        trả về SlicedPepperoni() mới();
    }
    công khai Clams createClam() {
        trả về FreshClams() mới();
    }
}
```



New York nằm trên bờ biển; nơi đây có ngao tươi. Chicago phải chấp nhận ngao đông lạnh.

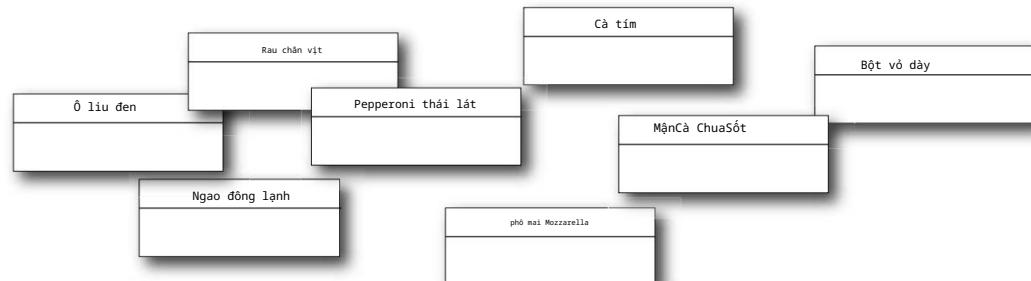
Pepperoni thái lát ngon nhất. Món này được chia sẻ giữa New York và Chicago. Hãy đảm bảo bạn sử dụng nó ở trang tiếp theo khi bạn tự mình thực hiện nhà máy Chicago

xây dựng một nhà máy



Chuốt bút chì của bạn

Viết ChicagoPizzaIngredientFactory. Bạn có thể tham
chiếu các lớp bên dưới trong quá trình triển khai của mình:



Làm lại bánh pizza...

Chúng tôi đã khởi động tất cả các nhà máy và sẵn sàng sản xuất các thành phần chất lượng; bây giờ chúng tôi chỉ cần làm lại Pizza để chúng chỉ sử dụng các thành phần do nhà máy sản xuất. Chúng tôi sẽ bắt đầu với lớp Pizza trừu tượng của mình:

```
lớp trừu tượng công khai Pizza {
    Tên chuỗi;
    Nhào bột;
    Nước sốt nước sốt;
    Rau củ rau củ[];
    Phô mai phô mai;
    Xúc xích Ý Pepperoni;
    Nghêu ngao;
    trừu tượng void chuẩn bị();
}

void nướng() {
    System.out.println("Nướng trong 25 phút ở nhiệt độ 350");
}

void cắt() {
    System.out.println("Cắt pizza thành các lát chéo");
}

hộp tiêng() {
    System.out.println("Đặt pizza vào hộp PizzaStore chính thức");
}

void setName(String name) { this.name =
    name;
}

Chuỗi getName() { trả về tên;
}

công khai String toString() {
    // mã để in pizza ở đây
}
}
```

Mỗi chiếc pizza đều có một số nguyên liệu được sử dụng để chế biến.

Bây giờ chúng ta đã làm cho phương thức chuẩn bị trở nên trừu tượng. Đây là nơi chúng ta sẽ thu thập các nguyên liệu cần thiết cho món pizza, tất nhiên sẽ lấy từ nhà máy nguyên liệu.

Các phương pháp khác của chúng tôi vẫn giữ nguyên, ngoại trừ phương pháp chuẩn bị.

thành phần tách rời

Làm lại bánh pizza, tiếp tục...

Bây giờ bạn đã có một chiếc Pizza trừu tượng để làm việc, đã đến lúc tạo ra những chiếc Pizza theo phong cách New York và Chicago - chỉ có điều lần này họ sẽ lấy nguyên liệu trực tiếp từ nhà máy. Những ngày tháng cắt giảm nguyên liệu của bên nhượng quyền đã qua rồi!

Khi chúng tôi viết mã Factory Method, chúng tôi có một lớp NYCheesePizza và một lớp ChicagoCheesePizza. Nếu bạn nhìn vào hai lớp, điểm khác biệt duy nhất là sử dụng nguyên liệu địa phương. Các loại pizza được làm giống nhau (bột + nước sốt + phô mai). Tương tự với các loại pizza khác: Veggie, Clam, v.v. Tất cả đều tuân theo các bước chuẩn bị giống nhau; chúng chỉ có các thành phần khác nhau.

Vì vậy, bạn sẽ thấy rằng chúng ta thực sự không cần hai lớp cho mỗi chiếc pizza; nhà máy nguyên liệu sẽ xử lý những khác biệt theo vùng cho chúng ta. Đây là Pizza phô mai:

```

lớp công khai CheesePizza mở rộng Pizza {
    PizzaIngredientFactory thành phầnFactory;
    công khai CheesePizza(PizzaIngredientFactory thành phầnFactory) {
        this.ingredientFactory = thành phầnFactory;
    }
    void chuẩn bị() {
        System.out.println("Đang chuẩn bị " + name);
        bột = thành phầnFactory.createDough();
        nước sốt = thành phầnFactory.createSauce();
        cheese = thành phầnFactory.createCheese();
    }
}

```



Để làm pizza, chúng ta cần một nhà máy cung cấp nguyên liệu. Vì vậy, mỗi lớp Pizza có một nhà máy được truyền vào hàm tạo của nó và được lưu trữ trong một biến thể hiện.

← Đây chính là nơi phép thuật xảy ra!



Phương thức prepare() thực hiện từng bước để tạo ra một chiếc bánh pizza phô mai và mỗi lần cần một nguyên liệu, nó sẽ yêu cầu nhà máy sản xuất nguyên liệu đó.



Mã Gần

Mã Pizza sử dụng nhà máy mà nó được biên soạn để sản xuất các thành phần được sử dụng trong pizza. Các thành phần được sản xuất phụ thuộc vào nhà máy mà chúng ta đang sử dụng. Lớp Pizza không quan tâm; nó biết cách làm pizza. Nay giờ, nó được tách khỏi sự khác biệt về thành phần theo vùng và có thể dễ dàng tái sử dụng khi có các nhà máy cho Rockies, Tây Bắc Thái Bình Dương và xa hơn nữa.

```
nước sốt = thành phầnFactory.createSauce();
```

Chúng tôi đang thiết lập biến thể hiện Pizza để tham chiếu đến loại nước sốt cụ thể được sử dụng trong chiếc pizza này.

Đây là nhà máy sản xuất nguyên liệu của chúng tôi. Pizza không quan tâm đến nhà máy nào được sử dụng, miễn là đó là nhà máy nguyên liệu.

Phương thức createSauce() trả về loại nước sốt được sử dụng trong vùng của nó. Nếu đây là nhà máy sản xuất nguyên liệu NY, thì chúng ta sẽ có nước sốt marinara.

Chúng ta hãy cùng xem qua ClamPizza nhé:

```
lớp công khai ClamPizza mở rộng Pizza {
    PizzaIngredientFactory thành phầnFactory;

    công khai ClamPizza(PizzaIngredientFactory thành phầnFactory) {
        this.ingredientFactory = thành phầnFactory;
    }

    void chuẩn bị() {
        System.out.println("Đang chuẩn bị " + name);
        bột = thành phầnFactory.createDough();
        nước sốt = thành phầnFactory.createSauce();
        cheese = thành phầnFactory.createCheese();
        clam = thành phầnFactory.createClam();
    }
}
```

ClamPizza cũng cần giàu một nhà máy sản xuất nguyên liệu.

Để làm pizza nghêu, phương pháp chế biến là thu thập các nguyên liệu phù hợp từ nhà máy địa phương.

Nếu là nhà máy ở New York, nghêu sẽ tươi; nếu là ở Chicago, nghêu sẽ được đông lạnh.

sử dụng đúng nhà máy nguyên liệu

Ghé thăm lại các cửa hàng pizza của chúng tôi

Chúng ta sắp hoàn thành rồi; chúng ta chỉ cần đến các cửa hàng nhuộm quyến của mình một chuyến để đảm bảo họ đang sử dụng đúng loại Pizza. Chúng ta cũng cần cung cấp cho họ thông tin tham khảo về các nhà máy nguyên liệu địa phương của họ:

```
lớp công khai NYPizzaStore mở rộng PizzaStore {

    được bảo vệ Pizza createPizza(String item) {
        Pizza pizza = null;
        PizzaIngredientFactory thành phầnFactory =
            mới NYPizzaIngredientFactory();

        nếu (item.equals("cheese")) {

            pizza = new CheesePizza(ingredientFactory);
            pizza.setName("Pizza phô mai kiểu New York");

        } else if (item.equals("veggie")) {

            pizza = VeggiePizza(ingredientFactory) mới;
            pizza.setName("Pizza chay kiểu New York");

        } else if (item.equals("clam")) {

            pizza = new ClamPizza(ingredientFactory);
            pizza.setName("Pizza nghêu kiểu New York");

        } else if (item.equals("pepperoni")) {
            pizza = new PepperoniPizza(ingredientFactory);
            pizza.setName("Pizza Pepperoni kiểu New York");

        } trả về pizza;
    }
}
```

Cửa hàng NY được tạo thành từ một nhà máy sản xuất nguyên liệu pizza NY. Nhà máy này sẽ được sử dụng để sản xuất các nguyên liệu dành cho tất cả các loại pizza kiểu New York.



Bây giờ chúng ta chuyển cho mỗi chiếc pizza một nhà máy nơi sẽ sản xuất ra các thành phần của nó.



Hãy xem lại một trang và đảm bảo rằng bạn hiểu được cách pizza và nhà máy hoạt động cùng nhau!



Đối với mỗi loại Pizza, chúng tôi tạo một loại Pizza mới và cung cấp cho nó nhà máy cần thiết để lấy nguyên liệu.

não Apower

So sánh phiên bản phương thức createPizza() này với phiên bản trong phần triển khai Phương thức Factory ở đầu chương.

Chúng ta đã làm gì?

Đó là một loạt các thay đổi mã; chính xác thì chúng tôi đã làm gì?

Chúng tôi đã cung cấp một phương tiện để tạo ra một nhóm nguyên liệu làm pizza bằng cách giới thiệu một loại nhà máy mới có tên là Nhà máy triều tượng.

Abstract Factory cung cấp cho chúng ta một giao diện để tạo ra một nhóm sản phẩm. Bằng cách viết mã sử dụng giao diện này, chúng ta tách mã của mình khỏi nhà máy thực tế tạo ra các sản phẩm.

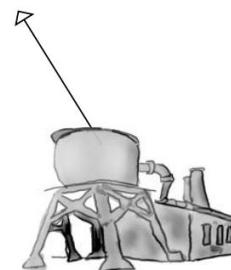
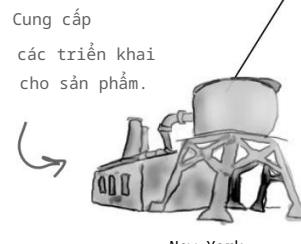
Điều đó cho phép chúng tôi triển khai nhiều nhà máy sản xuất các sản phẩm phù hợp với nhiều bối cảnh khác nhau - chẳng hạn như các khu vực khác nhau, các hệ điều hành khác nhau hoặc giao diện khác nhau.

Vì mã của chúng tôi được tách biệt khỏi các sản phẩm thực tế nên chúng tôi có thể thay thế các nhà máy khác nhau để có được các hành vi khác nhau (ví dụ như lấy nước sốt marinara thay vì cà chua mận).

Abstract Factory cung cấp giao diện cho một nhóm sản phẩm. Nhóm sản phẩm là gì? Trong trường hợp của chúng tôi, đó là tất cả những thứ chúng tôi cần để làm một chiếc pizza: bột, nước sốt, phô mai, thịt và rau.

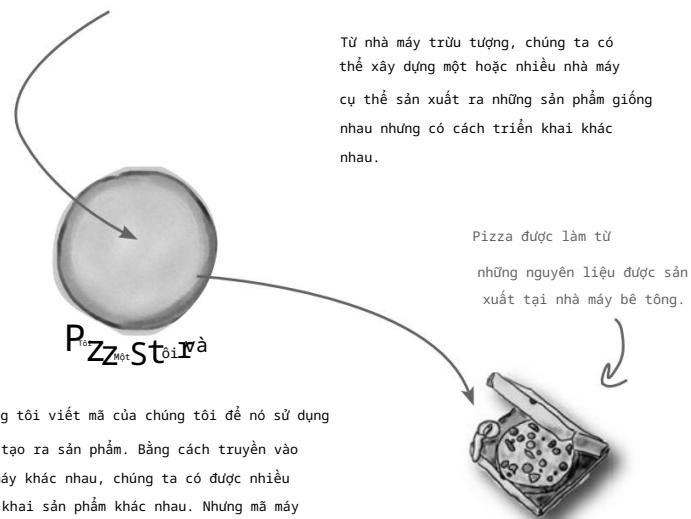


ObjectvilleAbstract IngredientFactory



New York

Chicago



Từ nhà máy triều tượng, chúng ta có thể xây dựng một hoặc nhiều nhà máy cụ thể sản xuất ra những sản phẩm giống nhau nhưng có cách triển khai khác nhau.

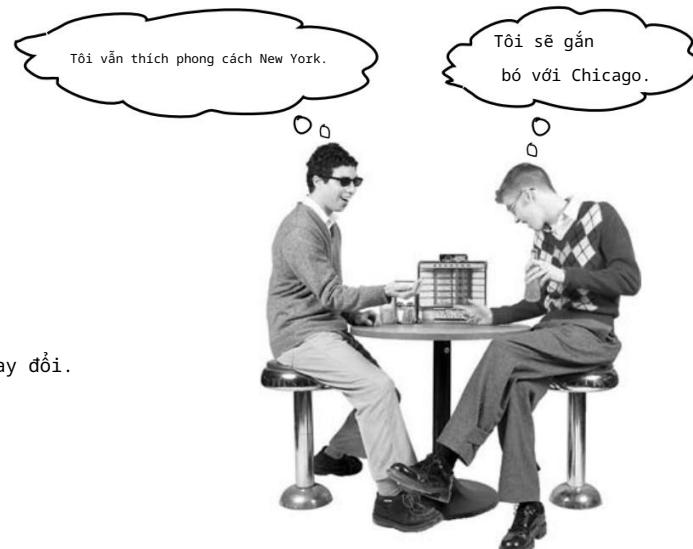
Pizza được làm từ những nguyên liệu được sản xuất tại nhà máy bê tông.

gọi thêm một ít pizza nữa

Thêm pizza cho Ethan và Joel...

Ethan và Joel không thể có đủ Objectville Pizza! Điều họ không biết là bây giờ đơn hàng của họ đang sử dụng các nhà máy nguyên liệu mới. Vì vậy, bây giờ khi họ đặt hàng...

Hậu
trường



Phần đầu tiên của quy trình đặt hàng không hề thay đổi.

Chúng ta hãy theo dõi lại đơn hàng của Ethan:

- Đầu tiên chúng ta cần một NY PizzaStore:

```
PizzaStore nyPizzaStore = new NYPizzaStore();
```

Tạo một phiên bản
của NYPizzaStore.

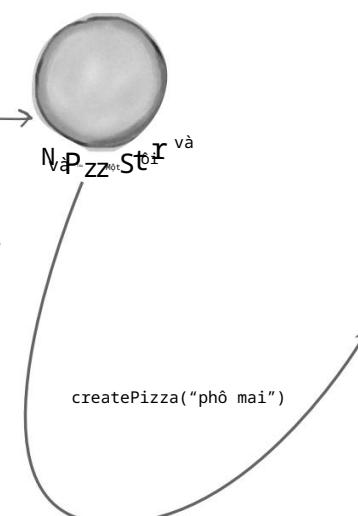
- Bây giờ chúng ta đã có cửa hàng, chúng ta có thể nhận đơn hàng:

```
nyPizzaStore.orderPizza("phô mai");
```

phương thức orderPizza() được gọi
trên thẻ hiện nyPizzaStore.

- Phương thức orderPizza() đầu tiên gọi phương thức createPizza():

```
Pizza pizza = createPizza("phô mát");
```



mẫu nhà máy

Từ đây mọi thứ thay đổi, vì chúng tôi
đang sử dụng một nhà máy nguyên liệu

Hậu
trường

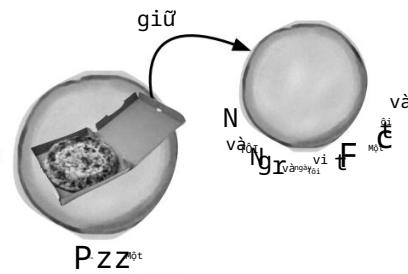


- 4 Khi phương thức `createPizza()` được gọi, đó là lúc nhà máy
thành phần của chúng ta tham gia vào:

Nhà máy thành phần được chọn và
khởi tạo trong `PizzaStore`, sau đó
được truyền vào hàm tạo của mỗi chiếc pizza.

`Pizza pizza = mới CheesePizza(nyIngredientFactory);`

Tạo một phiên bản
Pizza được tạo
thành từ nhà máy
nguyên liệu New York.



- 5 Tiếp theo chúng ta cần chuẩn bị pizza. Khi phương thức
`prepare()` được gọi, nhà máy được yêu cầu chuẩn bị nguyên
liệu:

```
void chuẩn bị() {
    bột = nhà máy.createDough();
    sauce = factory.createSauce();
    phô mai = nhà máy.createCheese();
}
```

Đối với pizza của Ethan, nhà máy nguyên
liệu ở New York được sử dụng, vì vậy chúng ta
có được nguyên liệu của New York.

Vỏ mỏng
Nước sốt Marinara
Reggiano

bột

- 6 Cuối cùng, chúng ta đã có chiếc pizza đã hoàn thành trong tay và
phương thức `orderPizza()` sẽ nướng, cắt và đóng hộp pizza.

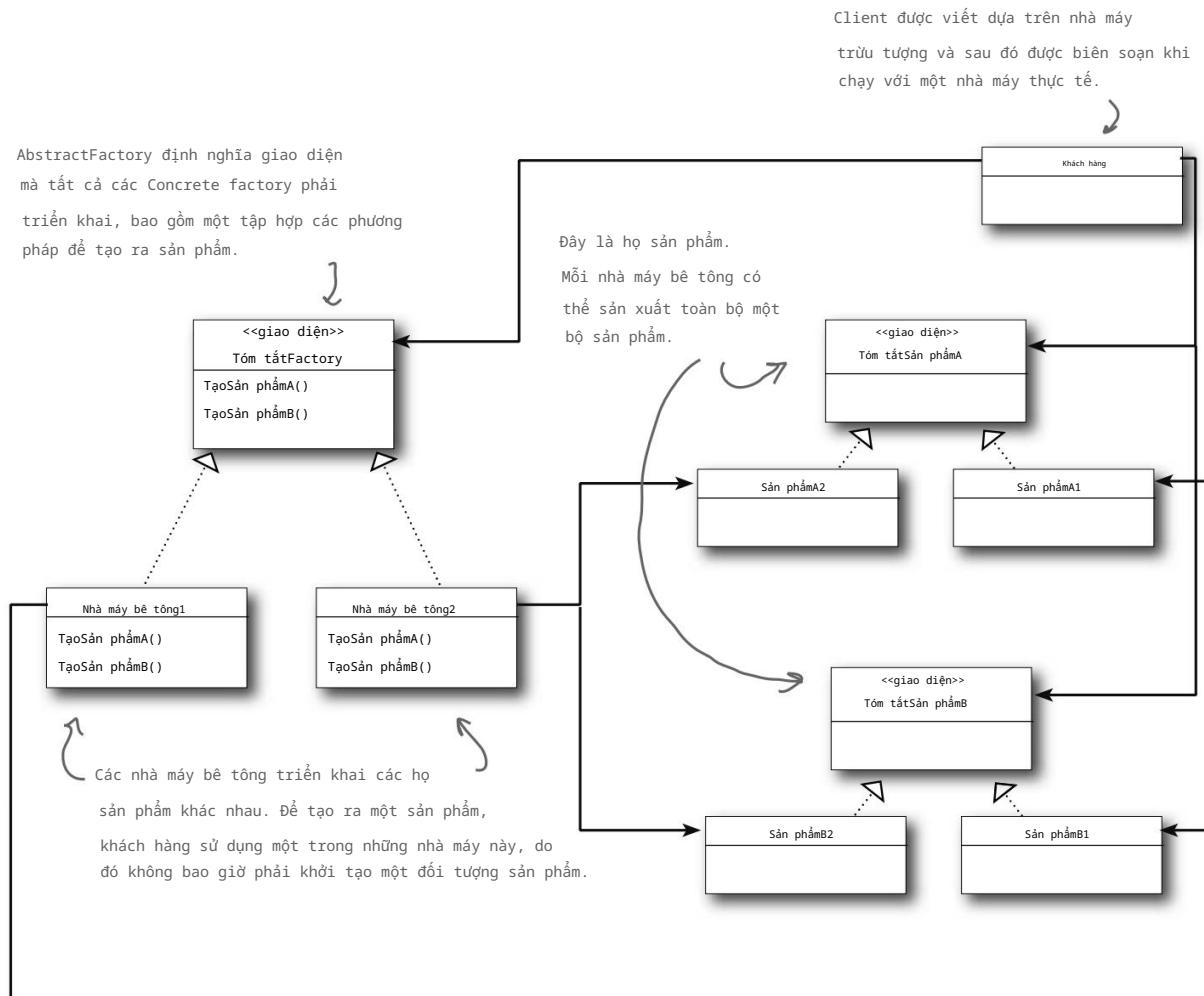
nhà máy trừu tượng được định nghĩa

Mẫu Abstract Factory được định nghĩa

Chúng tôi đang thêm một mẫu nhà máy nữa vào họ mẫu của chúng tôi, mẫu này cho phép chúng tôi tạo ra các họ sản phẩm. Hãy cùng xem định nghĩa chính thức cho mẫu này:

Mẫu Abstract Factory cung cấp giao diện để tạo ra các họ đối tượng có liên quan hoặc phụ thuộc mà không cần chỉ định các lớp cụ thể của chúng.

Chúng ta chắc chắn đã thấy Abstract Factory cho phép một khách hàng sử dụng một giao diện trừu tượng để tạo ra một tập hợp các sản phẩm có liên quan mà không cần biết (hoặc quan tâm) đến các sản phẩm cụ thể thực sự được sản xuất. Theo cách này, khách hàng được tách khỏi bất kỳ thông số cụ thể nào của các sản phẩm cụ thể. Hãy xem sơ đồ lớp để xem tất cả những điều này gắn kết với nhau như thế nào:

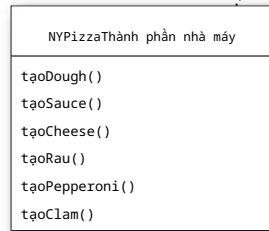
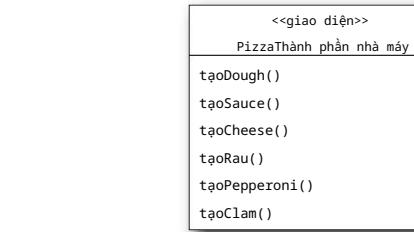


mẫu nhà máy

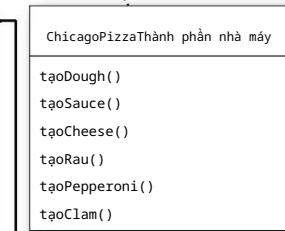
Đây là một sơ đồ lớp khá phức tạp; chúng ta hãy xem xét nó theo góc nhìn của PizzaStore:

PizzaIngredientFactory trừu tượng là giao diện xác định cách tạo ra một nhóm sản phẩm liên quan - mọi thứ chúng ta cần để làm một chiếc pizza.

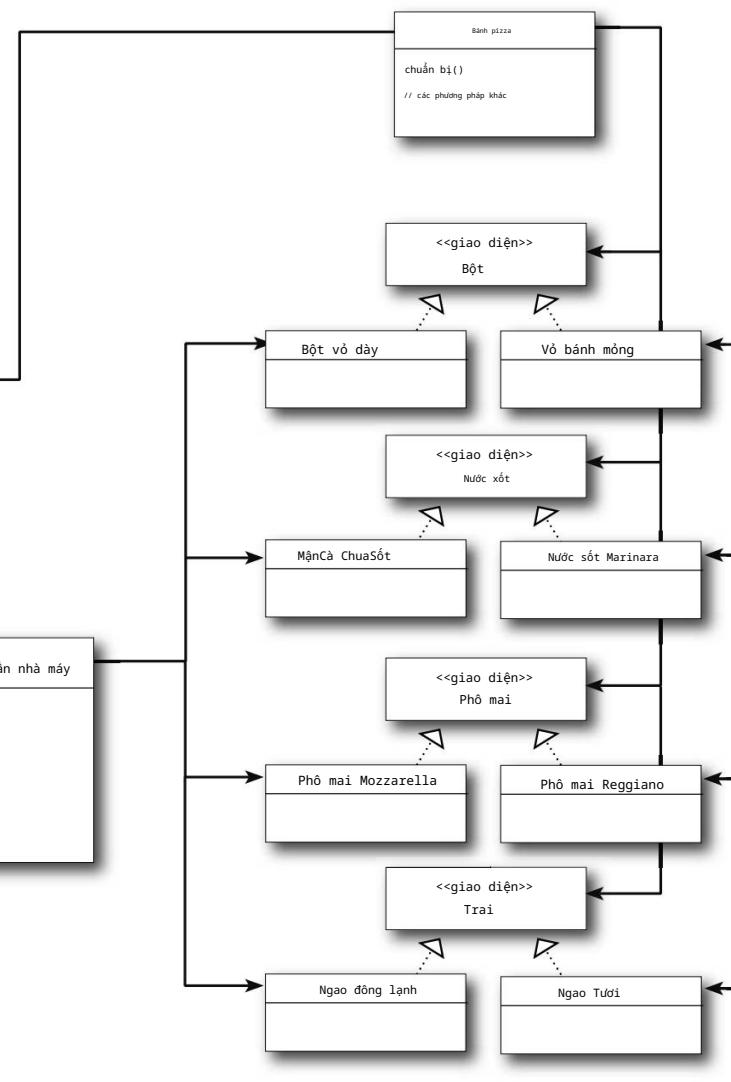
Các máy khách của Abstract Factory là những thể hiện cụ thể của lớp trừu tượng Pizza.



Công việc của các nhà máy sản xuất pizza bằng bê tông là sản xuất nguyên liệu làm pizza. Mỗi nhà máy đều biết cách tạo ra những sản phẩm phù hợp với khu vực của mình.



Mỗi nhà máy sản xuất một giải pháp khác nhau cho dòng sản phẩm.



phỏng vấn với các mẫu nhà máy



Tôi nhận thấy rằng mỗi phương thức
trong Abstract Factory thực sự trông giống như một
Phương thức Factory (createDough(), createSauce(), v.v.).
Mỗi phương thức được khai báo là trừu tượng và
các lớp con ghi đè lên nó để tạo ra một số
đối tượng. Đó không phải là Factory Method sao?

Đó có phải là một Phương pháp Nhà máy ẩn nấp bên trong
Nhà máy Trừu tượng không?
Đúng rồi! Đúng vậy, thường thì các phương thức của Abstract Factory được triển
khai như các phương thức của factory. Có lý, đúng không? Nhiệm vụ của Abstract Factory là
định nghĩa một giao diện để tạo ra một tập hợp các sản phẩm.
Mỗi phương thức trong giao diện đó chịu trách nhiệm tạo ra một sản phẩm cụ thể và
chúng tôi triển khai một lớp con của Abstract Factory để cung cấp các triển khai
đó. Vì vậy, các phương thức factory là một cách tự nhiên để triển khai các phương thức
product của bạn trong các abstract factory của bạn.



Các mẫu được phơi bày

Cuộc phỏng vấn tuần này:

Phương pháp Nhà máy và Nhà máy Trừu tượng, trên nhau

HeadFirst: Wow, một cuộc phỏng vấn với hai mẫu cùng một lúc! Đây là lần đầu tiên chúng tôi làm như vậy.

Factory Method: Vâng, tôi không chắc mình có thích bị xếp chung với Abstract Factory không, bạn biết đấy. Chỉ
vì cả hai chúng ta đều là mẫu nhà máy không có nghĩa là chúng ta không nên có cuộc phỏng vấn riêng.

HeadFirst: Đừng buồn, chúng tôi muốn phỏng vấn bạn cùng nhau để có thể giúp làm sáng tỏ mọi sự nhầm lẫn về việc ai là
ai đối với độc giả. Hai bạn có điểm tương đồng, và tôi nghe nói rằng đôi khi mọi người nhầm lẫn giữa hai bạn.

Abstract Factory: Đúng vậy, có những lúc tôi bị nhầm là Factory Method, và tôi biết bạn cũng gặp vấn đề tương tự, Factory
Method. Cả hai chúng ta đều rất giỏi trong việc tách ứng dụng khỏi các triển khai cụ thể; chúng ta chỉ làm theo những
cách khác nhau. Vì vậy, tôi có thể hiểu tại sao đôi khi mọi người có thể nhầm lẫn chúng ta.

Factory Method: Vâng, nó vẫn làm tôi khó chịu. Sau cùng, tôi sử dụng các lớp để tạo và bạn sử dụng các đối tượng;
hoàn toàn khác nhau!

HeadFirst: Bạn có thể giải thích thêm về điều đó không, Factory Method?

Factory Method: Chắc chắn rồi. Cá Abstract Factory và tôi đều tạo ra các đối tượng - đó là công việc của chúng tôi. Nhưng tôi thực hiện thông qua kế thừa...

Abstract Factory: ...và tôi thực hiện điều đó thông qua việc sáng tác đối tượng.

Factory Method: Đúng vậy. Điều đó có nghĩa là để tạo đối tượng bằng Factory Method, bạn cần mở rộng một lớp và ghi đè một factory method.

HeadFirst: Và phương pháp nhà máy đó thực hiện những gì?

Factory Method: Tất nhiên là nó tạo ra các đối tượng! Ý tôi là, toàn bộ mục đích của Factory Method Pattern là bạn đang sử dụng một lớp con để thực hiện việc sáng tạo cho bạn. Theo cách đó, khách hàng chỉ cần biết loại trừu tượng mà họ đang sử dụng, lớp con lo lắng về loại cụ thể.

Nói cách khác, tôi tách biệt khách hàng khỏi những kiểu người cụ thể.

Abstract Factory: Và tôi cũng vậy, chỉ là tôi làm theo cách khác.

HeadFirst: Thời nào, Abstract Factory... anh có nói gì đó về thành phần đối tượng phải không?

Abstract Factory: Tôi cung cấp một loại trừu tượng để tạo ra một họ sản phẩm. Các lớp con của loại này xác định cách sản xuất các sản phẩm đó. Để sử dụng factory, bạn khởi tạo một factory và truyền nó vào một số mã được viết theo loại trừu tượng. Vì vậy, giống như Factory Method, khách hàng của tôi được tách khỏi các sản phẩm cụ thể thực tế mà họ sử dụng.

HeadFirst: Ô, tôi hiểu rồi, vậy thì một lợi thế nữa là bạn có thể nhóm các sản phẩm có liên quan lại với nhau.

Abstract Factory: Đúng vậy.

HeadFirst: Điều gì xảy ra nếu bạn cần mở rộng tập hợp các sản phẩm liên quan đó, chẳng hạn như thêm một sản phẩm khác? Điều đó không đòi hỏi phải thay đổi giao diện của bạn sao?

Abstract Factory: Đúng vậy; giao diện của tôi phải thay đổi nếu thêm sản phẩm mới, mà tôi biết mọi người không thích làm vậy....

Phương pháp nhà máy: <cười khúc khích>

Abstract Factory: Bạn đang cười nhạo điều gì vậy, Factory Method?

Phương pháp nhà máy: Ô, thời nào, đó là vấn đề lớn mà!

Thay đổi giao diện của bạn có nghĩa là bạn phải vào và thay đổi giao diện của mọi lớp con! Nghe có vẻ rất nhiều việc.

Abstract Factory: Vâng, nhưng tôi cần một giao diện lớn vì tôi thường tạo ra toàn bộ các dòng sản phẩm.

Bạn chỉ tạo ra một sản phẩm, vì vậy bạn không thực sự cần một giao diện lớn, bạn chỉ cần một phương pháp.

HeadFirst: Abstract Factory, tôi nghe nói anh thường sử dụng phương pháp nhà máy để triển khai các nhà máy bê tông của mình phải không?

Abstract Factory: Vâng, tôi thừa nhận, các nhà máy bê tông của tôi thường áp dụng phương pháp nhà máy để tạo ra sản phẩm của họ. Trong trường hợp của tôi, chúng chỉ được sử dụng để tạo ra sản phẩm...

Phương pháp nhà máy: ...trong trường hợp của tôi, tôi thường triển khai mã trong trình tạo trừu tượng sử dụng các kiểu cụ thể mà các lớp con tạo ra.

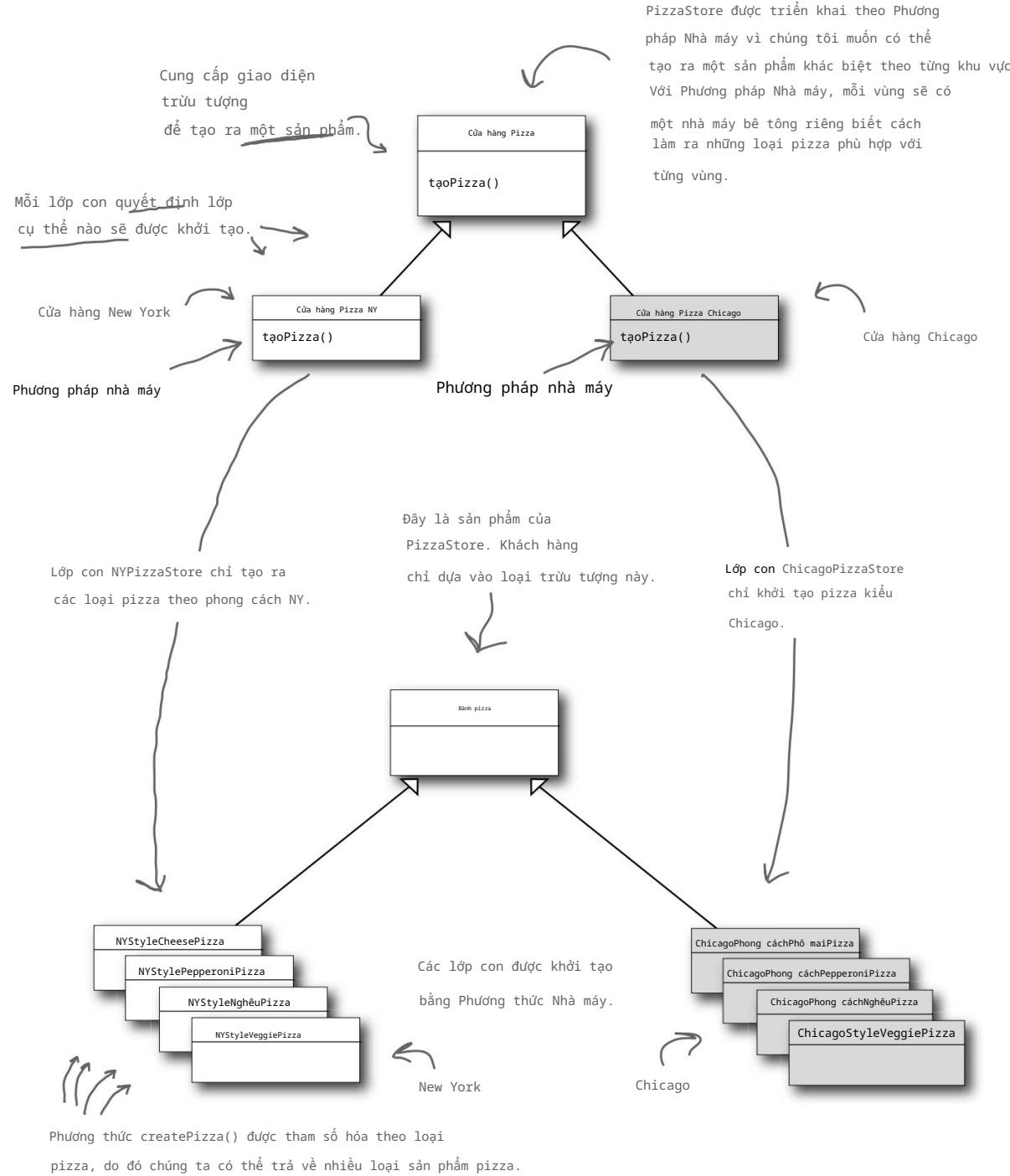
HeadFirst: Nghe có vẻ như cả hai bạn đều giỏi trong lĩnh vực của mình. Tôi chắc rằng mọi người thích có sự lựa chọn; sau cùng, các nhà máy rất hữu ích, họ sẽ muốn sử dụng chúng trong mọi tình huống khác nhau. Cả hai bạn đều đóng gói việc tạo đối tượng để giữ cho các ứng dụng được kết nối lỏng lẻo và ít phụ thuộc vào các triển khai, điều này thực sự tuyệt vời, cho dù bạn đang sử dụng Factory Method hay Abstract Factory. Tôi có thể cho phép mỗi người nói một lời chia tay không?

Abstract Factory: Cảm ơn. Hãy nhớ đến tôi, Abstract Factory, và sử dụng tôi bất cứ khi nào bạn có nhóm sản phẩm cần tạo và bạn muốn đảm bảo rằng khách hàng của mình tạo ra các sản phẩm phù hợp với nhau.

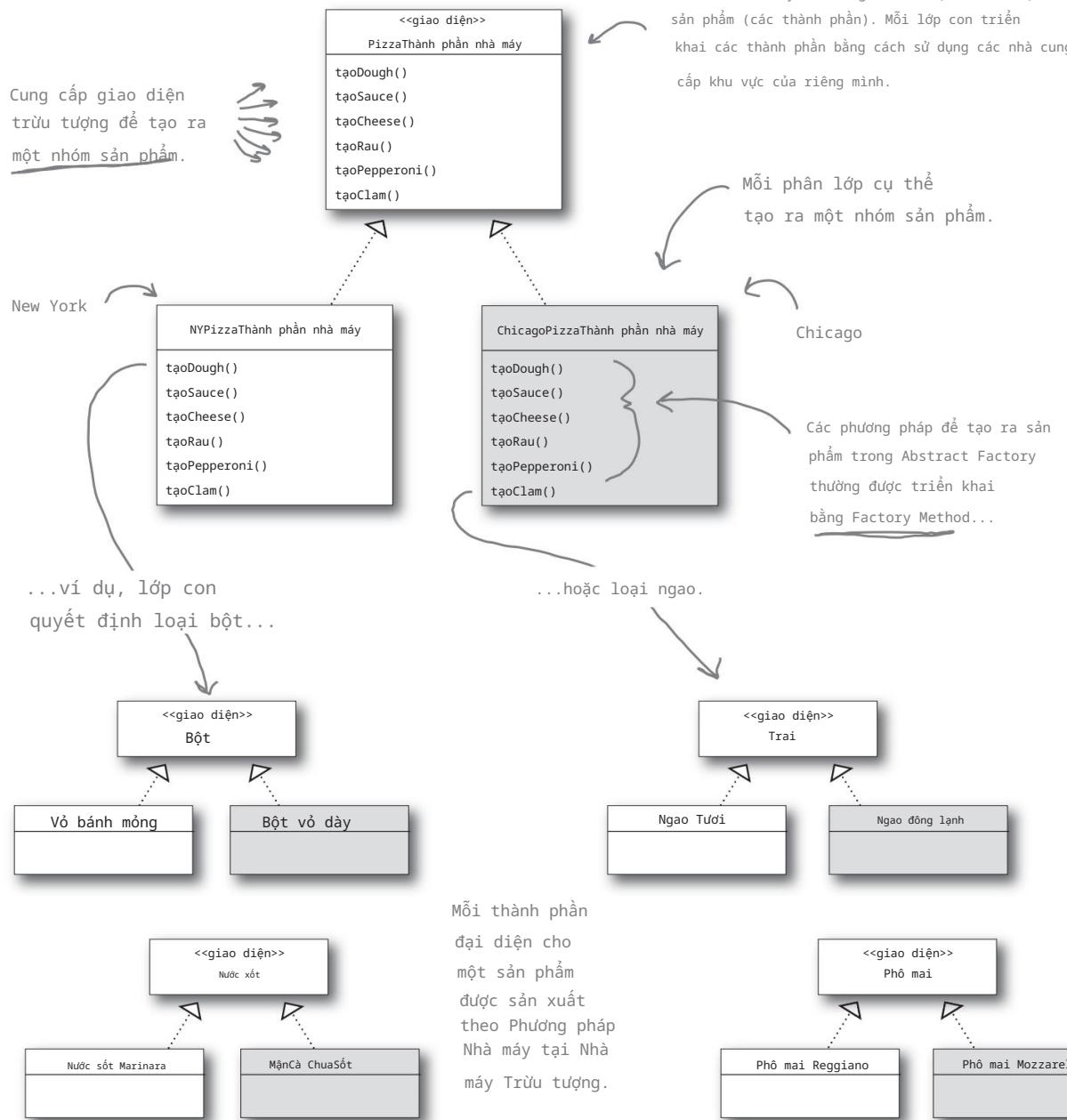
Factory Method: Và tôi là Factory Method; hãy sử dụng tôi để tách mã khách của bạn khỏi các lớp cụ thể mà bạn cần khởi tạo hoặc nếu bạn không biết trước tất cả các lớp cụ thể mà bạn sẽ cần. Để sử dụng tôi, chỉ cần phân lớp me và triển khai phương thức factory của tôi!

các mẫu so sánh

So sánh Factory Method và Abstract Factory



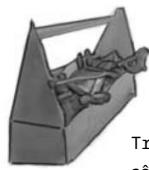
mẫu nhà máy



Các phân lớp sản phẩm tạo ra các tập hợp song song các họ sản phẩm.

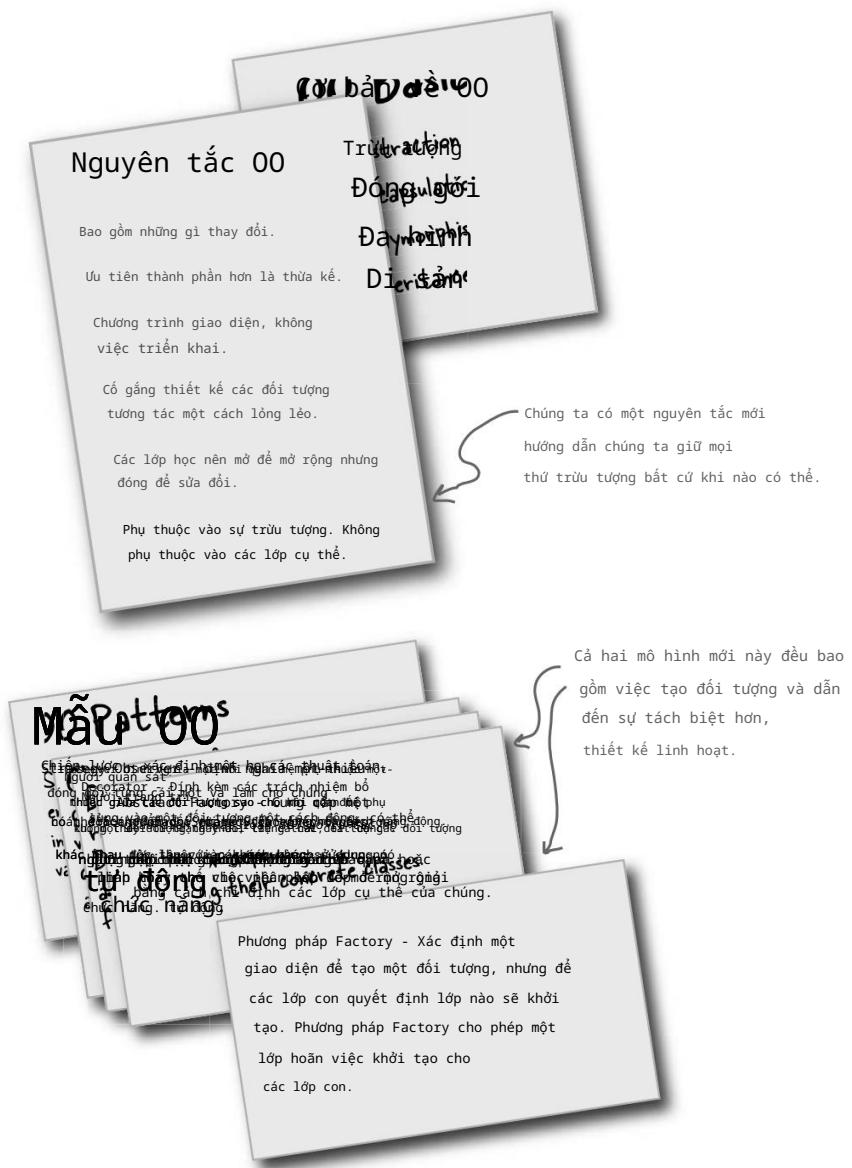
Ở đây chúng ta có một họ nguyên liệu New York và một họ nguyên liệu Chicago.

hộp công cụ thiết kế của bạn



Công cụ cho hộp công cụ thiết kế của bạn

Trong chương này, chúng tôi đã thêm hai công cụ nữa vào hộp công cụ của bạn: Factory Method và Abstract Factory. Cả hai mẫu đều đóng gói việc tạo đối tượng và cho phép bạn tách mảng của mình khỏi các kiểu cụ thể.



ĐIỂM ĐẦU TIÊN

Þ Tất cả các nhà máy đều đóng gói việc tạo đối tượng.

Þ Simple Factory tuy không phải là một mẫu thiết kế thực sự, nhưng là một cách đơn giản để tách khách hàng của bạn khỏi các lớp cụ thể.

Phương pháp nhà máy dựa vào kế thừa: việc tạo đối tượng được chuyển giao cho các lớp con triển khai phương thức nhà máy để tạo đối tượng.

Þ Abstract Factory dựa vào thành phần đối tượng: việc tạo đối tượng được triển khai theo các phương thức được trình bày trong giao diện nhà máy.

Þ Tất cả các mẫu nhà máy đều được thúc đẩy liên kết lồng léo bằng cách giảm sự phụ thuộc của ứng dụng của bạn vào các lớp cụ thể.

Mục đích của Phương thức Factory là cho phép một lớp hoãn việc khởi tạo sang các lớp con của nó.

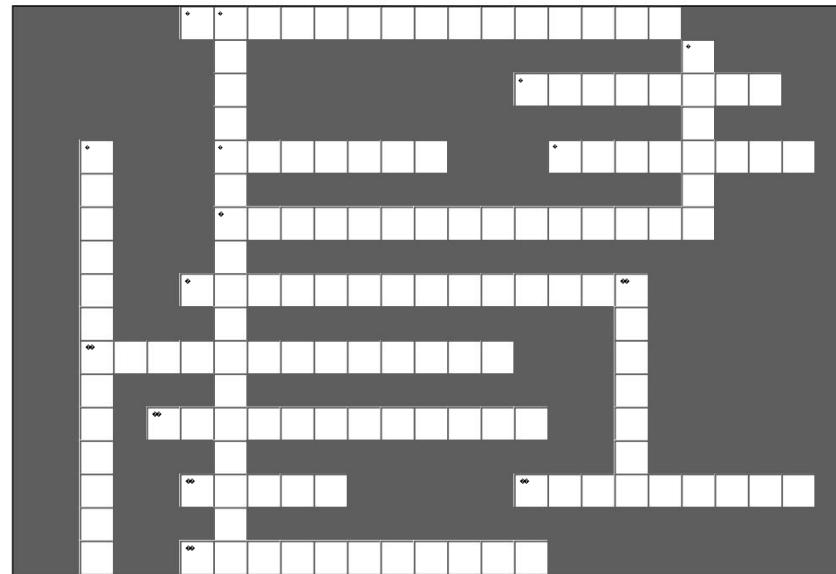
Mục đích của Abstract Factory là tạo ra các họ đối tượng có liên quan mà không cần phụ thuộc vào các lớp cụ thể của chúng.

Nguyên lý đảo ngược sự phụ thuộc hướng dẫn chúng ta tránh sự phụ thuộc vào các kiểu cụ thể và hướng tới sự trừu tượng.

Þ Các nhà máy là một kỹ thuật mạnh mẽ để mã hóa các trừu tượng, không phải các lớp cụ thể



Đây là một chương dài. Hãy lấy một miếng Pizza và thử giãn khi giải ô chữ này; tất cả các từ giải đều nằm trong chương này.



giải bài tập



Giải pháp bài tập



Chuốt bút chì của bạn

Chúng tôi đã khai trương NYPizzaStore; chỉ cần mở thêm hai cửa hàng nữa là chúng tôi sẽ sẵn sàng như ơng quyền! Viết các triển khai PizzaStore tại Chicago và California ở đây:

← Cả hai cửa hàng này đều gần giống hệt cửa hàng ở New York... họ chỉ tạo ra các loại pizza khác nhau

```
lớp công khai ChicagoPizzaStore mở rộng PizzaStore { được bảo vệ Pizza
    createPizza(String item) { nếu (item.equals("cheese")) {

        trả về ChicagoStyleCheesePizza();
    } else if (item.equals("veggie")) {
        trả về ChicagoStyleVeggiePizza();
    } else if (item.equals("clam")) {
        trả về ChicagoStyleClamPizza();
    } else if (item.equals("pepperoni")) {
        trả về new ChicagoStylePepperoniPizza(); } else trả về null;

    }
}
```

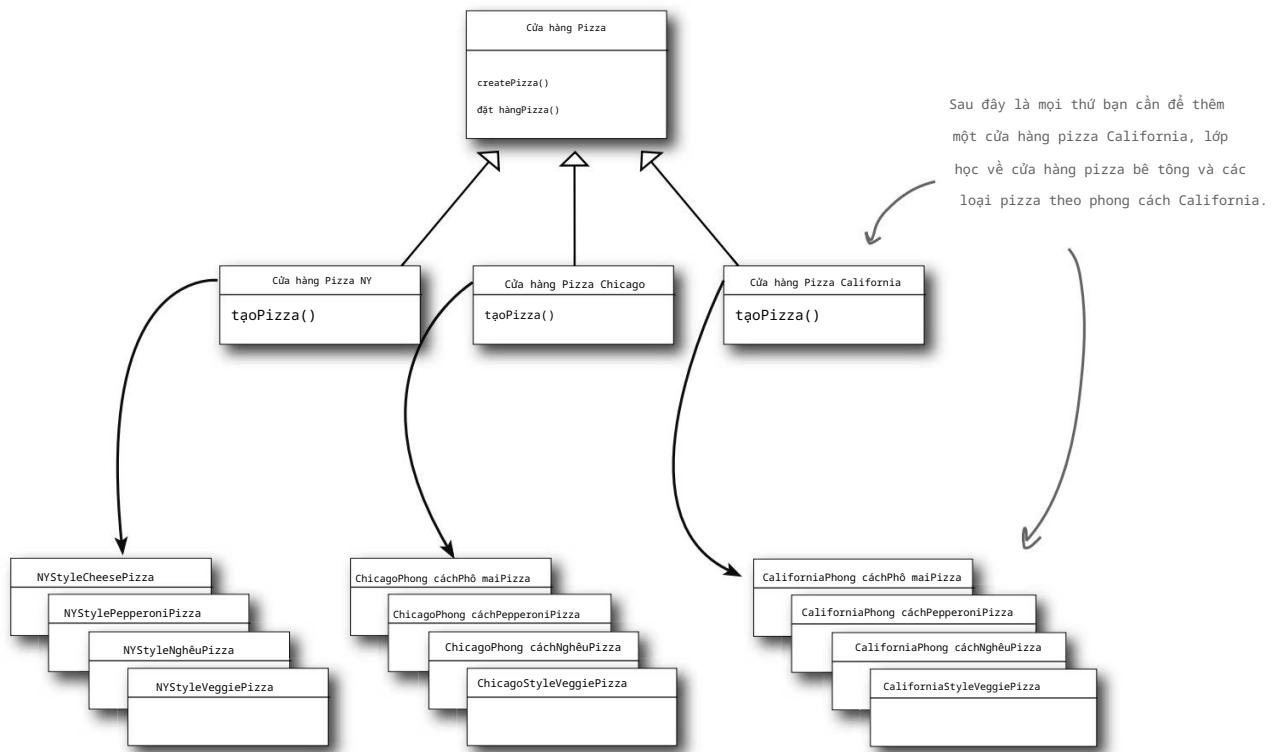
Đối với cửa hàng pizza ở Chicago,
chúng tôi chỉ cần đảm bảo tạo
ra những chiếc pizza theo
phong cách Chicago...

```
lớp công khai CaliforniaPizzaStore mở rộng PizzaStore {
    được bảo vệ Pizza createPizza(String item) { nếu
        (item.equals("cheese")) { trả về new
            CaliforniaStyleCheesePizza();
        } else if (item.equals("veggie")) {
            trả về CaliforniaStyleVeggiePizza() mới; } else if
        (item.equals("clam")) { trả về
            CaliforniaStyleClamPizza() mới;
        } else if (item.equals("pepperoni")) {
            trả về CaliforniaStylePepperoniPizza();
        } nếu không thì trả về null;
    }
}
```

và đối với cửa hàng pizza
California, chúng tôi tạo ra
những chiếc pizza theo phong cách California.

Giải pháp thiết kế câu đố

Chúng ta cần một loại pizza khác dành cho những người California điên rồ (điên theo nghĩa TỐT). Vẽ một tập hợp các lớp song song khác mà bạn cần để thêm một vùng California mới vào PizzaStore của chúng ta.



Để rồi, bây giờ hãy viết ra năm điều ngớ ngẩn nhất mà bạn có thể nghĩ ra để đặt lên pizza. Sau đó, bạn sẽ sẵn sàng kinh doanh pizza ở California!

Sau đây là những
gợi ý của chúng tôi...

Khoai tây nghiền với tỏi rang

Núi sốt BBQ

Trái tim atisô

M&M của

Đậu phộng

bạn đang ở đây 4 165

giải bài tập

Một PizzaStore rất phụ thuộc



Chuốt bút chì của bạn

Giả sử bạn chưa từng nghe đến nhà máy OO. Đây là phiên bản PizzaStore không sử dụng nhà máy; hãy để số lượng đối tượng pizza cụ thể mà lớp này phụ thuộc vào. Nếu bạn thêm pizza kiểu California vào PizzaStore này, thì nó sẽ phụ thuộc vào bao nhiêu đối tượng?

```

lớp công khai DependentPizzaStore {

    public Pizza createPizza(String style, String type) { Pizza pizza = null;
        if (style.equals("NY"))
            { if (type.equals("cheese"))
                { pizza = new NYStyleCheesePizza();

                    } else if (type.equals("veggie"))
                        pizza = new NYStyleVeggiePizza();
                    } else if (type.equals("clam"))
                        pizza = new NYStyleClamPizza(); } else if
                    (type.equals("pepperoni"))
                        pizza = new NYStylePepperoniPizza();
                    }

        } else if (style.equals("Chicago"))
            { if
                (type.equals("cheese"))
                    { pizza = new
                        ChicagoStyleCheesePizza();
                    } else if (type.equals("veggie"))
                        pizza = new ChicagoStyleVeggiePizza(); } else if
                    (type.equals("clam"))
                        pizza = new ChicagoStyleClamPizza();
                    } else if (type.equals("pepperoni"))
                        pizza = new ChicagoStylePepperoniPizza();

        } } else
            { System.out.println("Lỗi: loại pizza không hợp lệ"); return null;

    } pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
}
}

```

Xử lý tất cả các loại
pizza kiểu NY

Xử lý tất cả các
Pizza kiểu
Chicago

Bạn có thể viết
câu trả lời của mình ở đây:



Chuốt bút chì của bạn

Hãy tiếp tục và viết `ChicagoPizzaIngredientFactory`; bạn có thể tham chiếu các lớp bên dưới trong quá trình triển khai của mình:

```

lớp công khai ChicagoPizzaIngredientFactory triển khai
    PizzaIngredientFactory
{
    công khai Dough createDough() {
        trả về ThickCrustDough() mới();
    }

    công khai Sauce createSauce() {
        trả về PlumTomatoSauce mới();
    }

    công khai Cheese createCheese() {
        trả về MozzarellaCheese mới();
    }

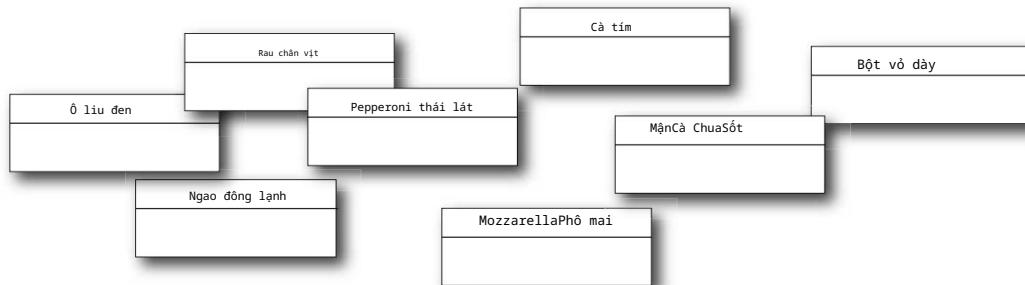
    công khai Veggies[] createVeggies() {
        Rau củ rau [] = { new BlackOlives(), new Spinach(), new
                           Eggplant() };

        trả lại rau;
    }

    công khai Pepperoni createPepperoni() { trả về new
        SlicedPepperoni();
    }

    công khai Clams createClam() {
        trả về FrozenClams() mới();
    }
}

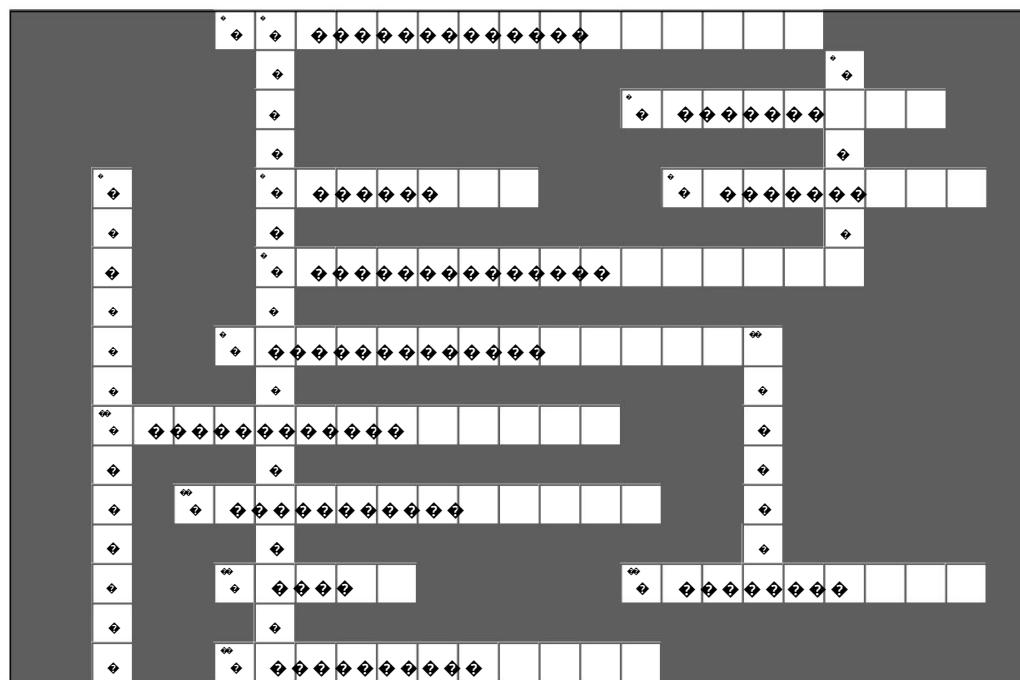
```



giải ô chữ



Giải pháp câu đố



???????

????????????????????????????????
????????????????????????????????
????????????????????????????????
????? ? ? ?????????? ??????????
????????????????????????????????
????????? ??????????
?? ?? ? ? ? ? ? ? ? ? ? ? ? ?
????????????????????????????????
????????? ?? ?? ?????????? ?????????
????????????????????????????????
????????? ?? ?? ?????????? ?????????
????????????????????????????????

?????

?? ?? ? ? ? ? — ?? ??????
????????????????????????????????
????????????????????????????????
?? ?? ?????? ?????? ?????? ?????? ? —
????????????????????????????????
????????????????????????????????
????????????????????????????????
????????????????????????????????

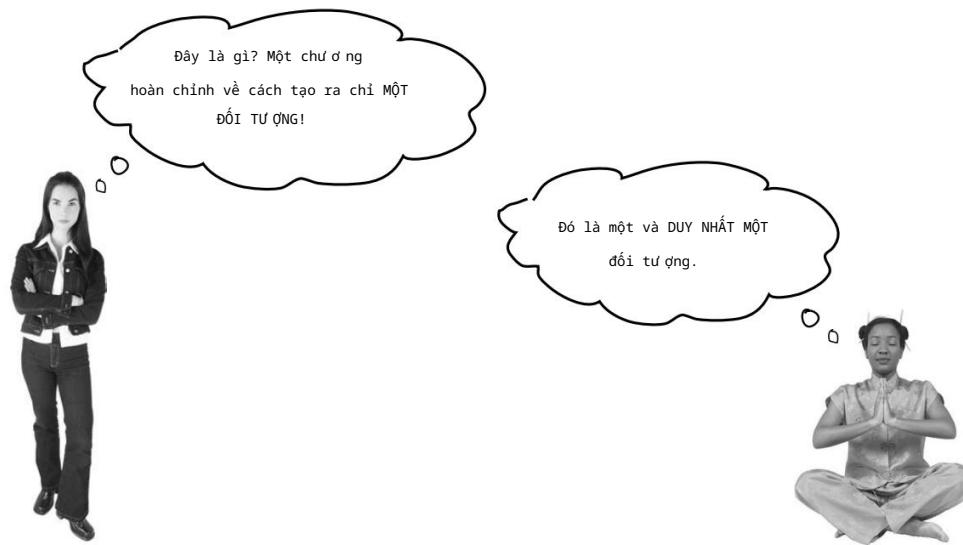
5 Mẫu Singleton

g Một trong một Loại Đối tư ợng g h



Điểm dừng tiếp theo của chúng ta là Singleton Pattern, tắm vé để chúng ta tạo ra những đối tượng độc nhất vô nhị mà chỉ có một thể hiện. Bạn có thể vui mừng khi biết rằng trong tất cả các mẫu, Singleton là mẫu đơn giản nhất về mặt sơ đồ lớp; trên thực tế, sơ đồ chỉ chứa một lớp duy nhất! Như ng dung quá thoải mái; mặc dù đơn giản theo quan điểm thiết kế lớp, chúng ta sẽ gặp khá nhiều trở ngại và lỗ hổng trong quá trình triển khai. Vì vậy, hãy thắt dây an toàn.

một và chỉ một



Nhà phát triển: Điều đó có tác dụng gì?

Guru: Có nhiều đối tượng mà chúng ta chỉ cần một trong số đó: nhóm luồng, bộ nhớ đệm, hộp thoại, đối tượng xử lý tùy chọn và cài đặt số đăng ký, đối tượng được sử dụng để ghi nhật ký và đối tượng hoạt động như trình điều khiển thiết bị cho các thiết bị như máy in và card đồ họa. Trên thực tế, đối với nhiều loại đối tượng này, nếu chúng ta khởi tạo nhiều hơn một đối tượng, chúng ta sẽ gặp phải đủ loại vấn đề như hành vi chương trình không chính xác, sử dụng quá nhiều tài nguyên hoặc kết quả không nhất quán.

Nhà phát triển: Đư ợc rồi, có thể có những lớp chỉ nên được khởi tạo một lần, nhưng tôi có cần cả một chương cho việc này không? Tôi không thể chỉ làm điều này theo quy ước hoặc theo biến toàn cục sao? Bạn biết đấy, giống như trong Java, tôi có thể làm điều đó bằng một biến tĩnh.

Guru: Theo nhiều cách, Singleton Pattern là một quy ước để đảm bảo chỉ một và chỉ một đối tượng được khởi tạo cho một lớp nhất định. Nếu bạn có một quy ước tốt hơn, thế giới sẽ muốn nghe về nó; nhưng hãy nhớ rằng, giống như tất cả các quy ước, Singleton Pattern là một phương pháp đã được kiểm tra theo thời gian để đảm bảo chỉ có một đối tượng được tạo ra. Singleton Pattern cũng cung cấp cho chúng ta một điểm truy cập toàn cục, giống như một biến toàn cục, nhưng không có như ợc điểm.

Nhà phát triển: Như ợc điểm là gì?

Guru: Vâng, đây là một ví dụ: nếu bạn gán một đối tượng cho một biến toàn cục, thì đối tượng đó có thể được tạo khi ứng dụng của bạn bắt đầu. Đúng không? Nếu đối tượng này tồn tại nguyên và ứng dụng của bạn không bao giờ sử dụng nó thì sao? Như bạn sẽ thấy, với Singleton Pattern, chúng ta chỉ có thể tạo đối tượng khi cần thiết.

Nhà phát triển: Việc này có vẻ không quá khó khăn.

Thầy: Nếu bạn nắm rõ các biến và phương thức lớp tĩnh cũng như các trình sửa đổi truy cập thì nó không phải là vấn đề. Như ng, trong cả hai trường hợp, thật thú vị khi thấy Singleton hoạt động như thế nào, và, nghe có vẻ đơn giản, mà Singleton rất khó để làm đúng. Hãy tự hỏi: làm thế nào để ngăn không cho nhiều hơn một đối tượng được tạo? Điều đó không quá rõ ràng, phải không?

Cô gái đọc thân nhỏ bé

Một bài tập nhỏ theo phong cách Socratic của The Little Lisper

Bạn sẽ tạo ra một đối tượng duy nhất như thế nào?

MyObject mới ();

Và, nếu một đối tượng khác muốn tạo MyObject thì sao? Nó có thể gọi new trên MyObject một lần nữa không?

Vâng tất nhiên.

Vậy thì miễn là chúng ta có một lớp, chúng ta có thể luôn khởi tạo nó một hoặc nhiều lần không?

Có. Vâng, chỉ khi đó là lớp học công cộng thôi.

Còn nếu không thì sao?

Vâng, nếu đó không phải là lớp công khai, chỉ các lớp trong cùng một gói mới có thể khởi tạo nó. Nhưng chúng vẫn có thể khởi tạo nó nhiều hơn một lần.

Ồ, thú vị đây.

Không, tôi chưa bao giờ nghĩ đến điều đó, nhưng tôi đoán nó có lý vì đó là một định nghĩa pháp lý.

Bạn có biết mình có thể làm được điều này không?

```
công khai MyClass {  
    riêng tư MyClass() {}  
}
```

Điều này có nghĩa là gì?

Tôi cho rằng đây là một lớp không thể khởi tạo được vì nó có hàm tạo riêng.

Vậy, có đối tượng nào có thể sử dụng hàm tạo riêng tư không?

Hmm, tôi nghĩ mã trong MyClass là mã duy nhất có thể gọi nó. Nhưng điều đó không có nhiều ý nghĩa.

tạo một singleton

Tại sao không?

Bởi vì tôi phải có một thể hiện của lớp để gọi nó, nhưng tôi không thể có một thể hiện vì không có lớp nào khác có thể khởi tạo nó. Đây là vấn đề con gà và quả trứng: Tôi có thể sử dụng hàm tạo từ một đối tượng có kiểu MyClass, nhưng tôi không bao giờ có thể khởi tạo đối tượng đó vì không có đối tượng nào khác có thể sử dụng "new MyClass()".

Đư ợc thôi. Đó chỉ là một suy nghĩ thôi.

Điều này có nghĩa là gì?

MyClass là một lớp có phương thức tĩnh. Chúng ta có thể gọi phương thức tĩnh như thế này:

```
MyClass.getInstance();
```

```
công khai MyClass {  
  
    công khai tĩnh MyClass getInstance() {  
        }  
}
```

Tại sao bạn lại sử dụng MyClass thay vì tên đối tượng nào đó?

Vâng, getInstance() là một phương thức tĩnh; nói cách khác, nó là một phương thức CLASS. Bạn cần sử dụng tên lớp để tham chiếu đến một phương thức tĩnh.

Thật thú vị. Nếu chúng ta ghép mọi thứ lại với nhau thì sao?

Ồ, chắc chắn là có thể.

Bây giờ tôi có thể khởi tạo MyClass không?

```
công khai MyClass {  
  
    riêng tư MyClass() {}  
  
    công khai tĩnh MyClass getInstance() {  
        trả về MyClass mới();  
    }  
}
```

Vậy bây giờ bạn có thể nghĩ ra cách thứ hai để khởi tạo một đối tượng không?

```
MyClass.getInstance();
```

Bạn có thể hoàn thiện mã để chỉ tạo MỘT phiên bản MyClass không?

Vâng, tôi nghĩ vậy...

(Bạn sẽ tìm thấy mã ở trang tiếp theo.)

mẫu đơ n

Phân tích Singleton cổ điển

Triển khai mẫu

```

lớp công khai Singleton {
    Singleton tĩnh riêng tư uniqueInstance;

    // các biến thể hữu ích khác ở đây

    Singleton riêng tư () {}

    công khai tĩnh Singleton getInstance() {
        nếu (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        trả về uniqueInstance;
    }

    // các phương pháp hữu ích khác ở đây
}

```

Hãy đổi tên MyClass thành Singleton.

Chúng ta có một biến tĩnh để lưu trữ một thể hiện của lớp Singleton.

Hàm tạo của chúng ta được khai báo là riêng tư; chỉ Singleton mới có thể khởi tạo lớp này!

Phương thức getInstance() cung cấp cho chúng ta cách khởi tạo lớp và trả về một thể hiện của lớp đó.

Tất nhiên, Singleton là một lớp bình thường; nó có các biến thể hiện và phương thức hữu ích khác.



Mã Gắn

```

uniqueInstance lưu trữ MỘT thể hiện của chúng ta; hãy nhớ rằng,
đây là một biến tĩnh.

nếu (uniqueInstance == null) {
    uniqueInstance = new MyClass();
}

trả về uniqueInstance;

```

Nếu uniqueInstance là null thì chúng ta vẫn chưa tạo ra thể hiện đó...

...và nếu nó không tồn tại, chúng ta khởi tạo Singleton thông qua hàm tạo riêng của nó và gán nó cho uniqueInstance. Lưu ý rằng nếu chúng ta không bao giờ cần đến thể hiện đó, nó sẽ không bao giờ được tạo; đây là khởi tạo lười biếng.

Khi chúng ta chạy đoạn mã này, chúng ta đã có một thể hiện và trả về thể hiện đó.

Nếu uniqueInstance không phải là null thì có nghĩa là nó đã được tạo trước đó. Chúng ta chỉ cần chuyển sang câu lệnh return.

phỏng vấn với singleton



Các mẫu đú ợc phò i bày

Cuộc phỏng vấn tuần này:

Lời thú tội của một người độc thân

HeadFirst: Hôm nay chúng tôi rất vui được mang đến cho bạn cuộc phỏng vấn với một đối tượng Singleton. Tại sao bạn không bắt đầu bằng cách kể cho chúng tôi đôi chút về bản thân bạn?

Singleton: Vâng, tôi hoàn toàn độc đáo; tôi chỉ có một mình!

HeadFirst: Một?

Singleton: Vâng, một. Tôi dựa trên Mẫu Singleton, đảm bảo rằng tại bất kỳ thời điểm nào cũng chỉ có một trường hợp duy nhất của tôi.

HeadFirst: Như vậy không phải là lãng phí sao? Ai đó đã dành thời gian để phát triển một lớp hoàn chỉnh và bây giờ tất cả những gì chúng ta có thể nhận được chỉ là một đối tượng từ nó?

Singleton: Không hề! Có sức mạnh trong ONE. Giả sử bạn có một đối tượng chứa các thiết lập số đăng ký. Bạn không muốn nhiều bản sao của đối tượng đó và các giá trị của nó chạy xung quanh - điều đó sẽ dẫn đến hỗn loạn. Bằng cách sử dụng một đối tượng như tôi, bạn có thể đảm bảo rằng mọi đối tượng trong ứng dụng của bạn đều sử dụng cùng một tài nguyên toàn cục.

HeadFirst: Hãy cho chúng tôi biết thêm.

Singleton: Ô, tôi giỏi dù thử. Độc thân đôi khi cũng có lợi thế, bạn biết đấy. Tôi thường được dùng để quản lý nhóm tài nguyên, như nhóm kết nối hoặc nhóm luồng.

HeadFirst: Nhưng chỉ có một người cùng loại với anh thôi sao? Nghe có vẻ cô đơn nhỉ.

Singleton: Vì chỉ có một mình tôi nên tôi luôn bận rộn, nhưng sẽ thật tuyệt nếu nhiều nhà phát triển biết đến tôi hơn - nhiều nhà phát triển gặp phải lỗi vì họ có nhiều bản sao của các đối tượng trôi nổi xung quanh mà họ thậm chí không biết.

HeadFirst: Vậy, nếu chúng tôi có thể hỏi, làm sao bạn biết chỉ có một người trong số các bạn? Không phải bất kỳ ai có một nhà điều hành mới đều có thể tạo ra một "bạn mới" sao?

Singleton: Không! Tôi thực sự độc đáo.

HeadFirst: Vậy các nhà phát triển có tuyên thệ sẽ không khởi tạo bạn nhiều hơn một lần không?

Singleton: Tất nhiên là không. Sự thật là, ừm, điều này hơi riêng tư như. Tôi không có hàm tạo công khai nào cả.

HeadFirst: KHÔNG CÓ CÔNG TRÌNH XÂY DỰNG CÔNG CỘNG! Ô, xin lỗi, không có công trình xây dựng công cộng nào sao?

Singleton: Đúng vậy. Hàm tạo của tôi được khai báo là private.

HeadFirst: Nó hoạt động như thế nào? Làm thế nào để BẠN CÓ THỂ được khởi tạo?

Singleton: Bạn thấy đấy, để nắm giữ một đối tượng Singleton, bạn không khởi tạo một đối tượng, bạn chỉ cần yêu cầu một thể hiện. Vì vậy, lớp của tôi có một phương thức tĩnh được gọi là getInstance(). Gọi phương thức đó và tôi sẽ xuất hiện ngay lập tức, sẵn sàng làm việc. Trên thực tế, tôi có thể đã giúp các đối tượng khác khi bạn yêu cầu tôi.

HeadFirst: Vâng, thưa ông Singleton, có vẻ như có rất nhiều điều ẩn giấu dưới vỏ bọc của ông để mọi việc này có thể thành công.

Cảm ơn bạn đã chia sẻ thông tin về bản thân và chúng tôi hy vọng sớm được trò chuyện với bạn lần nữa!

mẫu đơ n

Nhà máy Sôcôla

Mọi người đều biết rằng tất cả các nhà máy sô cô la hiện đại đều có lò hơi sô cô la được điều khiển bằng máy tính. Nhiệm vụ của lò hơi là đưa sô cô la và sữa vào, đun sôi chúng, sau đó chuyển sang giai đoạn tiếp theo là làm thanh sô cô la.

Đây là lớp điều khiển cho nồi nấu sô cô la công nghiệp của Choc-O-Holic, Inc. Hãy xem mã; bạn sẽ thấy họ đã cố gắng rất cẩn thận để đảm bảo rằng những điều tồi tệ không xảy ra, chẳng hạn như xả 500 gallon hỗn hợp chưa đun sôi, hoặc đổ đầy nồi hơi khi nó đã đầy, hoặc đun sôi một nồi hơi rỗng!

```

lớp công khai ChocolateBoiler {
    boolean riêng tư rỗng;
    boolean riêng tư đư ợc luộc;

    riêng công ChocolateBoiler() {
        rỗng = đúng;
        luộc = sai;
    }

    công khai void fill() {
        nếu (isEmpty()) {
            rỗng = sai;
            luộc = sai;
            // đổ đầy hỗn hợp sữa/sô cô la vào nồi hơi
        }
    }

    công khai void drain() {
        nếu (!isEmpty() && isBoiled()) {
            // đổ hết sữa đun sôi và sô-cô-la
            rỗng = đúng;
        }
    }

    công khai void đun sôi() {
        nếu (!isEmpty() && !isBoiled()) {
            // đun sôi các chất trong nồi
            luộc = đúng;
        }
    }

    boolean công khai isEmpty() {
        trả về kết quả rỗng;
    }

    công khai boolean isBoiled() {
        đun sôi trở lại;
    }
}

```



Mã này chỉ đư ợc chạy khi nồi hơi trống!

Để làm đầy nồi hơi, nồi phải rỗng và khi nồi đầy, chúng ta sẽ đặt cờ rỗng và cờ sôi.

Để xả nồi hơi, nồi hơi phải đầy (không rỗng) và cũng phải sôi. Sau khi xả xong, chúng ta đặt rỗng trở lại đúng.

Để đun sôi hỗn hợp, nồi hơi phải đầy và chưa sôi. Sau khi đun sôi, chúng ta đặt cờ đun sôi thành đúng.

nồi hơi sô cô la singleton

não Apower

Choc-O-Holic đã làm tốt việc đảm bảo những điều tồi tệ không xảy ra, bạn có nghĩ vậy không? Mặt khác, bạn có thể nghĩ rằng nếu hai trường hợp ChocolateBoiler bị lồng, một số điều rất tồi tệ có thể xảy ra.

Điều gì có thể xảy ra sai sót nếu có nhiều hơn một phiên bản ChocolateBoiler được tạo trong một ứng dụng?



Chuốt bút chì của bạn

Bạn có thể giúp Choc-O-Holic cải thiện lớp ChocolateBoiler bằng cách biến nó thành lớp đơn lẻ không?

```
lớp công khai ChocolateBoiler { boolean riêng
    tư rỗng; boolean riêng tư đã đun
    sôi;
```

```
private ChocolateBoiler() { empty = true;
    chosen = false;
```

```
}
```

```
public void fill() { if
    (isEmpty()) { empty =
        false; chosen =
        false; // đổ đầy nồi hơi
        bằng hỗn hợp sữa/sô cô la
    }
}
// phần còn lại của mã ChocolateBoiler...
}
```

176 Chương 5

Tải xuống tại WoweBook.Com

mẫu đơ n

Mẫu Singleton đư ợc định nghĩa

Bây giờ bạn đã hiểu cách triển khai Singleton cổ điển trong đầu, đã đến lúc ngồi lại, thư ờng thức một thanh sô cô la và khám phá những điểm tinh tế của Mẫu Singleton.

Chúng ta hãy bắt đầu với định nghĩa ngắn gọn về mẫu:

Mẫu Singleton đảm bảo mỗi lớp chỉ có một thể hiện và cung cấp điểm truy cập toàn cục vào thể hiện đó.

Không có gì ngạc nhiên lớn ở đây. Nhưng chúng ta hãy phân tích thêm một chút:

Β Thực sự thi điều gì đang diễn ra ở đây? Chúng ta đang lấy một lớp và để nó quản lý một thể hiện duy nhất của chính nó. Chúng ta cũng đang ngăn chặn bất kỳ lớp nào khác tạo ra một thể hiện mới của riêng nó. Để có được một thể hiện, bạn phải đi qua chính lớp đó.

Β Chúng tôi cũng cung cấp một điểm truy cập toàn cục đến thể hiện: bắt cứ khi nào bạn cần một thể hiện, chỉ cần truy vấn lớp và nó sẽ trả lại cho bạn thể hiện duy nhất. Như bạn đã thấy, chúng tôi có thể triển khai điều này để Singleton đư ợc tạo theo cách lười biếng, điều này đặc biệt quan trọng đối với các đối tượng sử dụng nhiều tài nguyên.

Đư ợc rồi, chúng ta hãy xem sơ đồ lớp:

Phương thức getInstance() là phương thức tĩnh, nghĩa là nó là phương thức lớp, do đó bạn có thể dễ dàng truy cập phương ng thức này từ bất kỳ đâu trong mã của mình bằng Singleton.getInstance(). Điều đó cũng dễ như truy cập một biến toàn cục, như chúng ta có đư ợc những lợi ích như khởi tạo lười biếng từ Singleton.

Độc thân
static uniqueInstance
// Dữ liệu Singleton hữu ích khác...
tính getInstance()
// Các phương thức Singleton hữu ích khác...

Biến lớp
uniqueInstance lưu trữ phiên bản Singleton duy nhất của chúng ta.

Một lớp triển khai Mô hình Singleton không chỉ là một Singleton; nó là một lớp có mục đích chung với tập hợp dữ liệu và phương thức riêng.

chủ đề là một vấn đề

Hershey, PA

Houston, chúng ta có vấn đề rồi...

Có vẻ như Chocolate Boiler đã làm chúng ta thất vọng; mặc dù chúng ta đã cải thiện mã bằng Classic Singleton, nhưng bằng cách nào đó, phuzzering thức `fill()` của ChocolateBoiler vẫn có thể bắt đầu đổ đầy boiler mặc dù một mẻ sữa và sôcôla đã sôi! Đó là 500 gallon sữa (và sôcôla) bị đổ! Chuyện gì đã xảy ra!?



Việc bổ sung các luồng có thể gây ra thế này? Không phải là trường hợp một khi chúng ta đã thiết lập biến `uniqueInstance` cho thể hiện duy nhất của ChocolateBoiler, tắt cả các lệnh gọi đến `getInstance()` đều phải trả về cùng một thể hiện? Đúng không?

mẫu đơ n

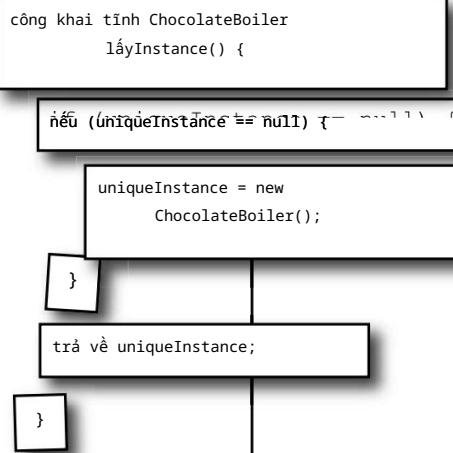
TRỞ THÀNH JVM



Chúng ta có hai luồng, mỗi luồng thực thi mã này. Nhiệm vụ của bạn là chạy JVM và xác định xem có truy cập hợp nào mà hai luồng có thể nắm giữ các đối tượng boiler khác nhau không. Gợi ý: bạn thực sự chỉ cần xem trình tự các hoạt động trong phư ơng thức getInstance() và giá trị của uniqueInstance để xem chúng có thể chồng chéo lên nhau như thế nào.

Sử dụng mã Magnets để giúp bạn nghiên cứu cách mã có thể xen kẽ để tạo ra hai đối tượng boiler.

```
Lò hơ i ChocolateBoiler =  
ChocolateBoiler.getInstance();  
đỗ đầy();  
đun sôi();  
làm khô hạn();
```



Chú đề
Một

Chú đề
Hai

Giá trị
của uniqueInstance

Hãy chắc chắn rằng bạn kiểm tra câu trả lời của mình ở trang 188 trước khi lật trang!

đa luồng và đơ n luồng

Xử lý đa luồng

Những rắc rối về đa luồng của chúng ta có thể được khắc phục dễ dàng bằng cách biến `getInstance()` thành một phương thức được đồng bộ hóa:

```
lớp công khai Singleton {
    Singleton tĩnh riêng tư uniqueInstance;

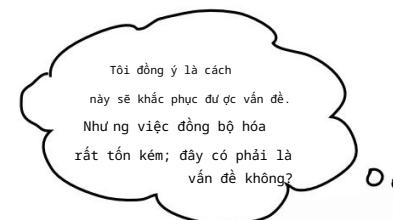
    // các biến thể hữu ích khác ở đây

    Singleton riêng tư () {}

    công khai tĩnh đồng bộ Singleton getInstance() {
        nếu (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        trả về uniqueInstance;
    }

    // các phương pháp hữu ích khác ở đây
}
```

Bằng cách thêm từ khóa `synchronized` vào `getInstance()`, chúng ta buộc mọi luồng phải đợi đến lượt trước khi có thể vào phương thức. Nghĩa là không có hai luồng nào có thể vào phương thức cùng một lúc.



Quan điểm hay, và thực ra nó tệ hơn một chút so với những gì bạn nói: thời điểm duy nhất đồng bộ hóa có liên quan là lần đầu tiên thông qua phương pháp này. Nói cách khác, sau khi chúng ta đặt biến `uniqueInstance` thành một thể hiện của `Singleton`, chúng ta không cần phải đồng bộ hóa phương pháp này nữa. Sau lần đầu tiên thông qua, đồng bộ hóa hoàn toàn không cần thiết!

Chúng ta có thể cải thiện đa luồng không?

Đối với hầu hết các ứng dụng Java, rõ ràng là chúng ta cần đảm bảo Singleton hoạt động khi có nhiều luồng.

Như ng việc đồng bộ hóa phu ơng thức getInstance() có vẻ khá tốn kém, vậy chúng ta phải làm gì?

Vâng, chúng ta có môt vài lựa chọn...

- Không làm gì nếu hiệu suất của getInstance() không quan trọng đối với ứng dụng của bạn

Đúng vậy; nếu việc gọi phu ơng thức getInstance() không gây ra chi phí đáng kể cho ứng dụng của bạn, hãy quên nó đi. Đồng bộ hóa getInstance() rất đơn giản và hiệu quả. Chỉ cần lưu ý rằng việc đồng bộ hóa một phu ơng thức có thể làm giảm hiệu suất xuống 100 lần, vì vậy nếu một phần lưu lư ợng truy cập cao trong mã của bạn bắt đầu sử dụng getInstance(), bạn có thể phải xem xét lại.

- Di chuyển đến một tru ờng hợp được tạo ra một cách hào hức thay vì một tru ờng hợp được tạo ra một cách lười biếng

Nếu ứng dụng của bạn luôn tạo và sử dụng một phiên bản của Singleton hoặc chỉ tạo và các khía cạnh thời gian chạy của Singleton không quá lớn, bạn có thể muốn tạo phiên bản của mình. Singleton hào hức như thế này:

```

lớp công khai Singleton {
    riêng tư tĩnh Singleton uniqueInstance = new Singleton();

    Singleton riêng tư () {}

    công khai tĩnh Singleton getInstance() {
        trả về uniqueInstance;
    }
}

```

Hãy tiếp tục và tạo một
thể hiện của Singleton
trong trình khởi tạo tĩnh.
Mã này được đảm bảo là an
tòn cho luồng!

Chúng ta đã có một thể
hiện rồi, vậy hãy trả về nó.

Sử dụng cách tiếp cận này, chúng ta dựa vào JVM để tạo ra phiên bản duy nhất của Singleton khi lớp được tải. JVM đảm bảo rằng phiên bản sẽ được tạo trước khi bất kỳ luồng nào truy cập vào biến static uniqueInstance.

khóa đã kiểm tra lại

3. Sử dụng “double-checked locking” để giảm việc sử dụng đồng bộ hóa trong getInstance()

Với khóa kiểm tra kép, trước tiên chúng ta kiểm tra xem một thê hiện có được tạo hay không, và nếu không, THÌ chúng ta đồng bộ hóa. Theo cách này, chúng ta chỉ đồng bộ hóa lần đầu tiên, chính xác những gì chúng ta muốn.

Hãy cùng kiểm tra mã:

```

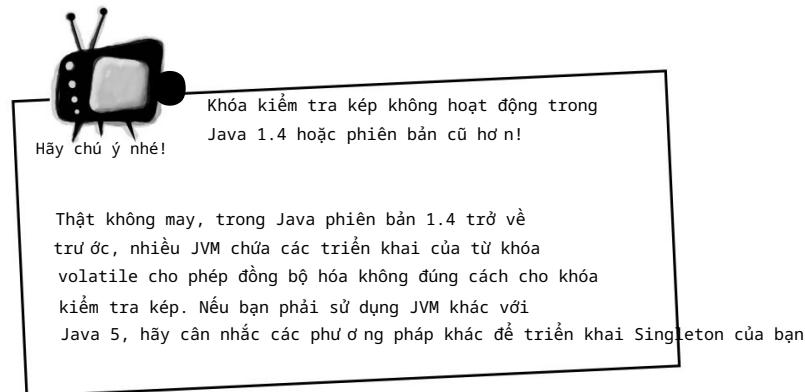
lớp công khai Singleton {
    riêng tư biến động tĩnh Singleton uniqueInstance;

    Singleton riêng tư () {}

    công khai tĩnh Singleton getInstance() {
        nếu (uniqueInstance == null) {
            đồng bộ (Singleton.class) {
                nếu (uniqueInstance == null) {
                    uniqueInstance = new Singleton();
                }
            }
        }
        trả về uniqueInstance;
    }
}

* Từ khóa volatile đảm bảo rằng nhiều luồng xử lý biến uniqueInstance một cách chính xác khi nó được khởi tạo thành phiên bản Singleton.
  
```

Nếu hiệu suất là vấn đề khi bạn sử dụng phương thức getInstance() thì phương pháp triển khai Singleton này có thể giảm đáng kể chi phí.



mẫu đơ n

Trong khi đó, quay lại Nhà máy Sôcôla...

Trong khi chúng tôi đang chẩn đoán các vấn đề đa luồng, nồi hơi sô cô la đã đư ợc dọn dẹp và sẵn sàng hoạt động. Như ng trư ớc tiên, chúng tôi phải sửa các vấn đề đa luồng. Chúng tôi có một số giải pháp trong tay, mỗi giải pháp có những đánh đổi khác nhau, vậy chúng tôi sẽ sử dụng giải pháp nào?



Chuốt bút chì của bạn

Đối với mỗi giải pháp, hãy mô tả khả năng áp dụng của nó vào vấn đề sửa chữa Chocolate. Mã nồi hơi:

Đóng bộ hóa phư ơng thức getInstance():

Sử dụng khởi tạo háo hức:

Kiểm tra khóa hai lần:

Chúc mừng!

Đến thời điểm này, Chocolate Factory là một khách hàng hài lòng và Choc-O-Holic rất vui khi có một số chuyên môn đư ợc áp dụng vào mã boiler của họ. Bất kề bạn áp dụng giải pháp đa luồng nào, boiler sẽ hoạt động tốt và không còn sự cố nữa. Xin chúc mừng. Bạn không chỉ thoát khỏi 500 pound sô cô la nóng trong chương này mà còn vượt qua đư ợc tất cả các vấn đề tiềm ẩn của Singleton.

bạn đang ở đây 4 183

hỏi đáp về singleton

không có câu hỏi ngớ ngẩn nào

Q: Đối với một mẫu đơn giản như vậy chỉ bao gồm một lớp, Singleton chắc chắn có một số vấn đề.

A: Vâng, chúng tôi đã cảnh báo bạn rồi! Như ng đừng để những vấn đề này làm bạn nản lòng; mặc dù việc triển khai Singleton đúng cách có thể rất khó khăn, nhưng sau khi đọc chương này, giờ đây bạn đã nắm rõ các kỹ thuật tạo Singleton và nên sử dụng chúng bất cứ khi nào bạn cần kiểm soát số lượng phiên bản bạn đang tạo.

H: Tôi không thể tạo một lớp học trong tất cả các phương thức và biến đều được định nghĩa là tĩnh? Điều đó có giống với Singleton không?

A: Có, nếu lớp của bạn tự học được chứa đựng và không phụ thuộc vào khởi tạo phức tạp. Tuy nhiên, do cách xử lý khởi tạo tĩnh trong Java, điều này có thể trở nên rất lỏn xộn, đặc biệt là nếu có nhiều lớp tham gia. Thông thường, kịch bản này có thể dẫn đến các lỗi tinh vi, khó tìm liên quan đến thứ tự khởi tạo.

Trừ khi có

nhu cầu cấp thiết phải triển khai "singleton" của bạn theo cách này, tốt hơn hết là nên ở lại thế giới đối tượng.

H: Còn trình tải lớp thì sao?
Tôi nghe nói có khả năng hai trình nạp lớp có thể tạo ra phiên bản Singleton riêng của chúng.

A: Đúng vậy, điều đó đúng vì mỗi lớp loader định nghĩa một không gian tên. Nếu bạn có hai hoặc nhiều classloader, bạn có thể tải cùng một lớp nhiều lần (mỗi lần trong một classloader). Bây giờ, nếu lớp đó là Singleton, thì vì chúng ta có nhiều hơn một phiên bản của lớp, chúng ta cũng có nhiều hơn một thể hiện của Singleton. Vì vậy, nếu bạn đang sử dụng nhiều classloader và Singleton, hãy cẩn thận. Một cách để giải quyết vấn đề này là tự chỉ định classloader.



Tin đồn về việc Singletons bị những người thu gom rác ăn thịt là bị thổi phồng quá mức

Trước Java 1.2, một lỗi trong trình thu gom rác cho phép Singleton được thu thập trước thời hạn nếu không có tham chiếu toàn cục nào đến chúng. Nói cách khác, bạn có thể tạo một Singleton và nếu tham chiếu duy nhất đến Singleton nằm trong chính Singleton đó, thì nó sẽ được trình thu gom rác thu thập và hủy. Điều này dẫn đến các lỗi gây nhầm lẫn vì sau khi Singleton được "thu thập", lệnh gọi tiếp theo tới getInstance() tạo ra một Singleton mới sáng bóng. Trong nhiều ứng dụng, điều này có thể gây ra hành vi gây nhầm lẫn khi trạng thái được đặt lại một cách bí ẩn về các giá trị ban đầu hoặc những thứ như kết nối mạng được đặt lại.

Kể từ Java 1.2, lỗi này đã được sửa và không còn cần tham chiếu toàn cục nữa. Nếu vì lý do nào đó, bạn vẫn sử dụng JVM trước Java 1.2, hãy lưu ý vấn đề này, nếu không, bạn có thể yên tâm vì biết rằng Singleton của bạn sẽ không bị thu thập trước thời hạn.

H: Tôi luôn được dạy rằng

một lớp chỉ nên làm một việc và chỉ một việc.
Một lớp làm hai việc được coi là thiết kế
OO kém.

Singleton không phải là đang vi phạm điều này sao?

A: Bạn sẽ đề cập đến

nguyên tắc "Một lớp, một trách nhiệm", và đúng, bạn nói đúng, Singleton không chỉ chịu trách nhiệm quản lý một thể hiện của nó (và cung cấp quyền truy cập toàn cục), mà còn chịu trách nhiệm cho bất kỳ vai trò chính nào của nó trong ứng dụng của bạn. Vì vậy, chắc chắn có thể lập luận rằng nó đang đảm nhiệm hai trách nhiệm. Tuy nhiên, không khó để thấy rằng có tiện ích trong một lớp quản lý thể hiện của riêng nó; nó chắc chắn làm cho thiết kế tổng thể đơn giản hơn. Ngoài ra, nhiều nhà phát triển quen thuộc với mô hình Singleton vì nó được sử dụng rộng rãi. Tuy nhiên, một số nhà phát triển cảm thấy cần phải trừu tượng hóa chức năng Singleton.

Q: Tôi muốn phân lớp của tôi

Mã Singleton, nhưng tôi gặp phải vấn đề. Có được phép phân lớp Singleton không?

H: Tôi vẫn chưa hiểu rõ lắm

tại sao biến toàn cục lại tệ hơn Singleton.

A: Một vấn đề với phân lớp

Singleton là constructor là private. Bạn không thể mở rộng một lớp với một constructor private. Vì vậy, điều đầu tiên bạn phải làm là thay đổi constructor của mình thành public hoặc protected. Nhưng sau đó, nó không thực sự không còn là Singleton nữa, vì các lớp khác có thể khởi tạo nó.

Nếu bạn thay đổi trình xây dựng của mình, sẽ có một vấn đề khác.

Việc triển khai Singleton dựa trên một biến tĩnh, vì vậy nếu bạn thực hiện một lớp con đơn giản, tất cả các lớp dẫn xuất của bạn sẽ chia sẻ cùng một biến thể hiện. Đây có lẽ không phải là điều bạn nghĩ đến. Vì vậy, để phân lớp con hoạt động, cần phải triển khai registry of types trong lớp cơ sở.

A: Trong Java, các biến toàn cục là

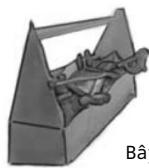
về cơ bản là tham chiếu tĩnh tới các đối tượng. Có một vài ưu điểm khi sử dụng biến toàn cục theo cách này. Chúng tôi đã đề cập đến một ưu điểm: vấn đề khởi tạo lười biếng so với háo hức. Nhưng chúng ta cần ghi nhớ mục đích của mẫu: đảm bảo chỉ có một thể hiện của một lớp tồn tại và cung cấp quyền truy cập toàn cục. Một biến toàn cục có thể cung cấp cái sau, nhưng không phải cái trước. Biến toàn cục cũng có xu hướng khuyến khích các nhà phát triển làm ô nhiễm không gian tên bằng nhiều tham chiếu toàn cục đến các đối tượng nhỏ.

Những người độc thân không khuyến khích điều này theo cùng một cách, nhưng vẫn có thể bị lạm dụng.

Trước khi triển khai một lược đồ như vậy, bạn nên tự hỏi bản thân xem bạn thực sự đạt được gì khi phân lớp Singleton. Giống như hầu hết các mẫu, Singleton không nhất thiết phải là một giải pháp có thể phù hợp với thư viện. Ngoài ra, mã Singleton rất dễ thêm vào bất kỳ lớp hiện có nào.

Cuối cùng, nếu bạn đang sử dụng một số lược đồ Singleton trong ứng dụng của mình, bạn nên xem xét kỹ lưỡng thiết kế của mình. Singleton được cho là nên sử dụng một cách tiết kiệm.

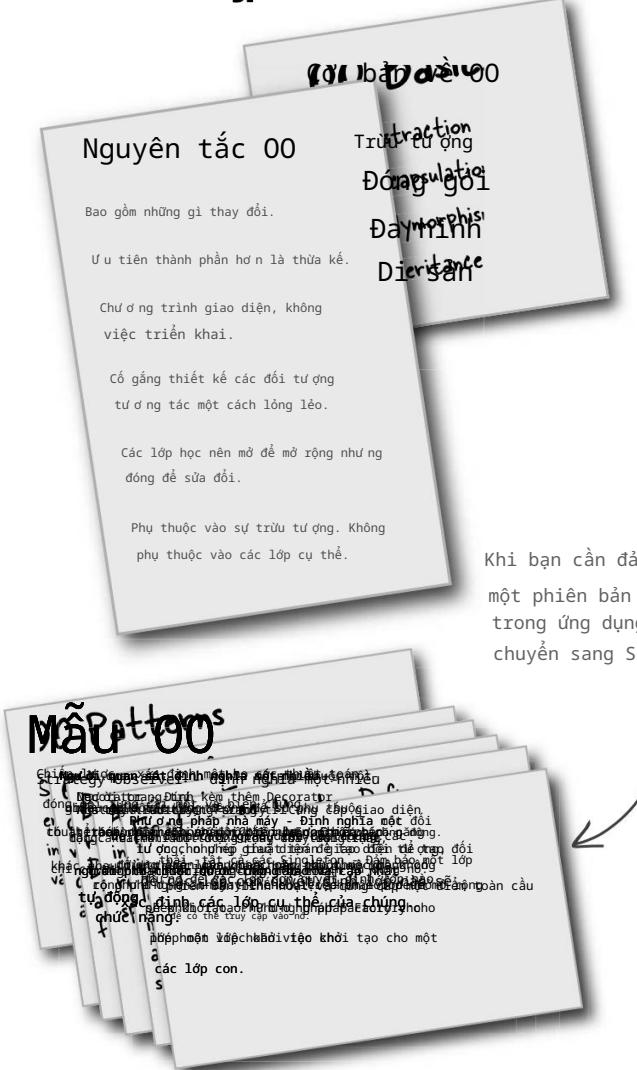
hộp công cụ thiết kế của bạn



Công cụ cho hộp công cụ thiết kế của bạn

Bây giờ bạn đã thêm một mẫu khác vào hộp công cụ của mình. Singleton cung cấp cho bạn một phương pháp khác để tạo đối tượng - trong trường hợp này là đối tượng duy nhất.

5



Như bạn đã thấy, mặc dù có vẻ đơn giản, nhưng có rất nhiều chi tiết liên quan đến việc triển khai Singleton. Tuy nhiên, sau khi đọc chương này, bạn đã sẵn sàng để sử dụng Singleton ngoài thực tế.

ĐIỂM ĐẦU TIÊN

Mẫu Singleton đảm bảo bạn chỉ có tối đa một phiên bản của một lớp trong ứng dụng của mình.

β Mẫu Singleton cũng
cung cấp điểm truy cập toàn cầu
tới trường hợp đó.

ß Việc triển khai Mẫu Singleton của Java sử dụng một hàm tạo riêng, một phương thức tĩnh kết hợp với một biến tĩnh.

B Kiểm tra hiệu suất của bạn và hạn chế về tài nguyên và cản thận lựa chọn triển khai Singleton phù hợp cho các ứng dụng đa luồng (và chúng ta nên coi tắt cả các ứng dụng đều là đa luồng!).

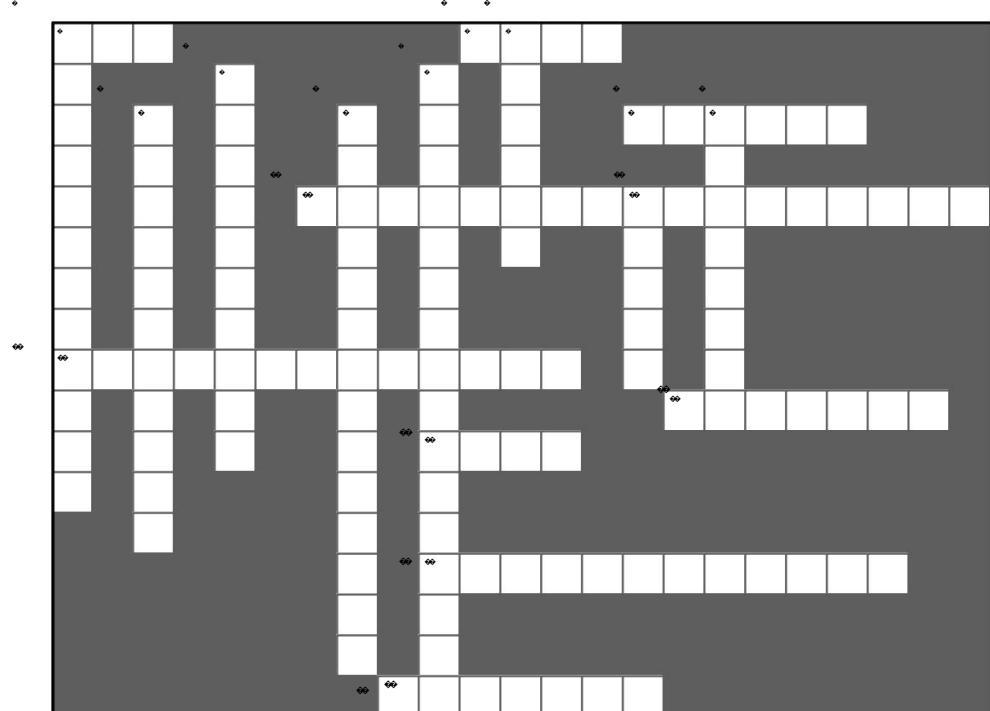
B Hãy cẩn thận với việc triển khai khóa
được kiểm tra kép; nó không an toàn
cho luồng trong các phiên bản
trúc Java 2, phiên bản 5.

B Hãy cẩn thận nếu bạn đang sử dụng nhiều trình tài lợp; điều này có thể làm hỏng việc triển khai Singleton và dẫn đến nhiều trường hợp.

- β Nếu bạn đang sử dụng JVM cũ hơn 1.2, bạn sẽ cần tạo một số đăng ký Singleton để vô hiệu hóa trình thu gom rác.

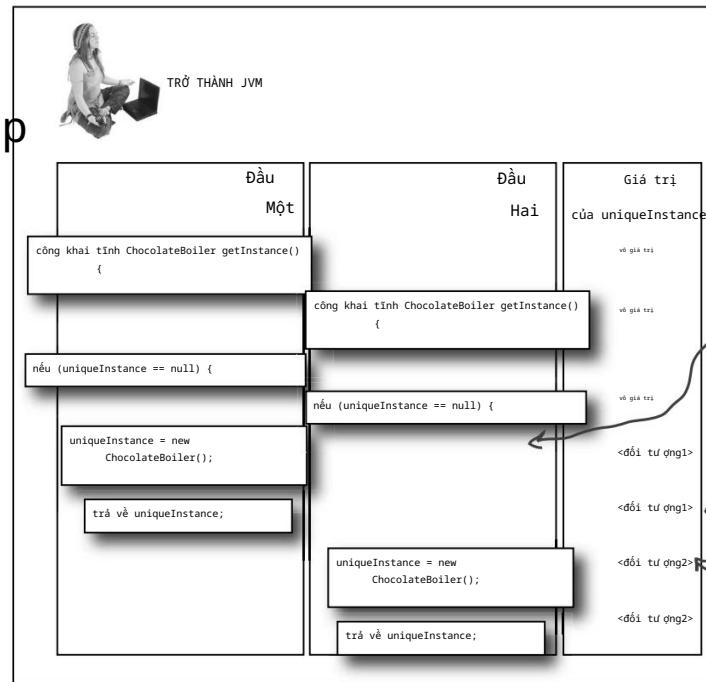


Ngồi xuống, mở hộp sô-cô-la mà bạn đã ợc gửi đến để giải quyết vấn đề da luồng và dành chút thời gian rảnh rỗi để giải câu đố ô chữ nhỏ này; tất cả các từ giải đều nằm trong chương này.



giải bài tập

Giải pháp bài tập



Chuốt bút chì của bạn! Bạn có thể giúp Choc-O-Holic cải thiện lớp ChocolateBoiler bằng cách biến nó thành lớp đơn lẻ không?

```
lớp công khai ChocolateBoiler { boolean riêng tư
    rỗng; boolean riêng tư đã đun sôi;

    riêng tư tĩnh ChocolateBoiler uniqueInstance;

    riêng tư ChocolateBoiler() { rỗng = đúng;
        luộc = sai;
    }

    công khai tĩnh ChocolateBoiler getInstance() { nếu (uniqueInstance ==
        null) {
            uniqueInstance = new ChocolateBoiler();

            } trả về uniqueInstance;
        }

    public void fi ll() { if
        (isEmpty()) { empty =
            false; chosen =
            false; // đồ đầy nồi hơ i
            bằng hỗn hợp sữa/sô cô la
        }
    }

    } // phần còn lại của mã ChocolateBoiler...
}
```

Giải pháp bài tập



Chuốt bút chì của bạn

Đối với mỗi giải pháp, hãy mô tả khả năng áp dụng của nó vào vấn đề sửa chữa Chocolate Mã nồi hơi:

Đóng bộ hóa phư ơng thức getInstance():

Một kỹ thuật đơn giản được đảm bảo là có hiệu quả. Chúng tôi đang như không có bất kỳ

mỗi quan ngại về hiệu suất của nồi đun sô-cô-la, vì vậy đây sẽ là một lựa chọn tốt.

Sử dụng khởi tạo hào hức:

Chúng tôi luôn luôn khởi tạo nồi hơi số cõ la trong mã của mình, do đó khởi tạo tĩnh

trường hợp này sẽ không gây ra mối lo ngại nào. Giải pháp này sẽ hoạt động tốt như phư ơng pháp đóng bộ hóa, mặc dù có

thì ít rõ ràng hơn đối với một nhà phát triển quen thuộc với mô hình chuẩn.

Đã kiểm tra khóa hai lần:

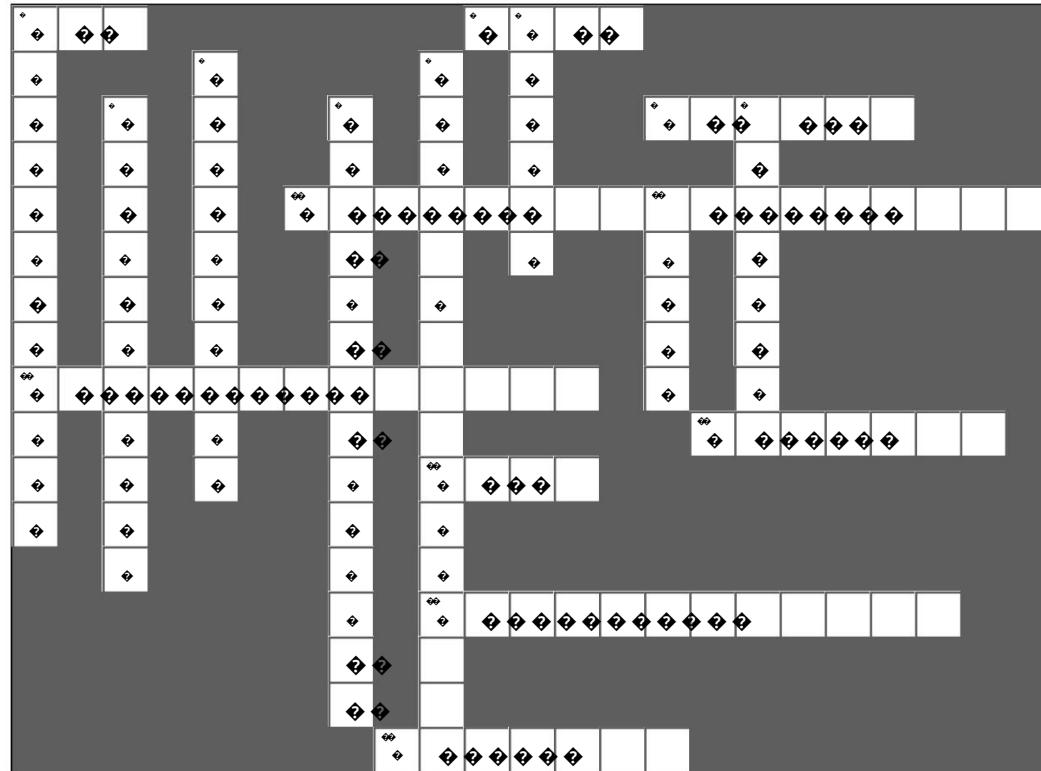
Vì chúng tôi không có mối quan tâm về hiệu suất, việc khóa kiểm tra lại có vẻ như là quá mức cần thiết. Ngoài ra, chúng tôi sẽ

phải đảm bảo rằng chúng ta đang chạy ít nhất Java 5.

giải ô chữ



Giải pháp bài tập



?

?

77 77777777 77777777 77777777 77777777 77777777
7777777777777777

77 7 77777777 77 77 777777 777777 77777777 77
77777777 77 77777777 77777777 77777777

777

7777777777777777 777 777 77777 77777 777777 777777

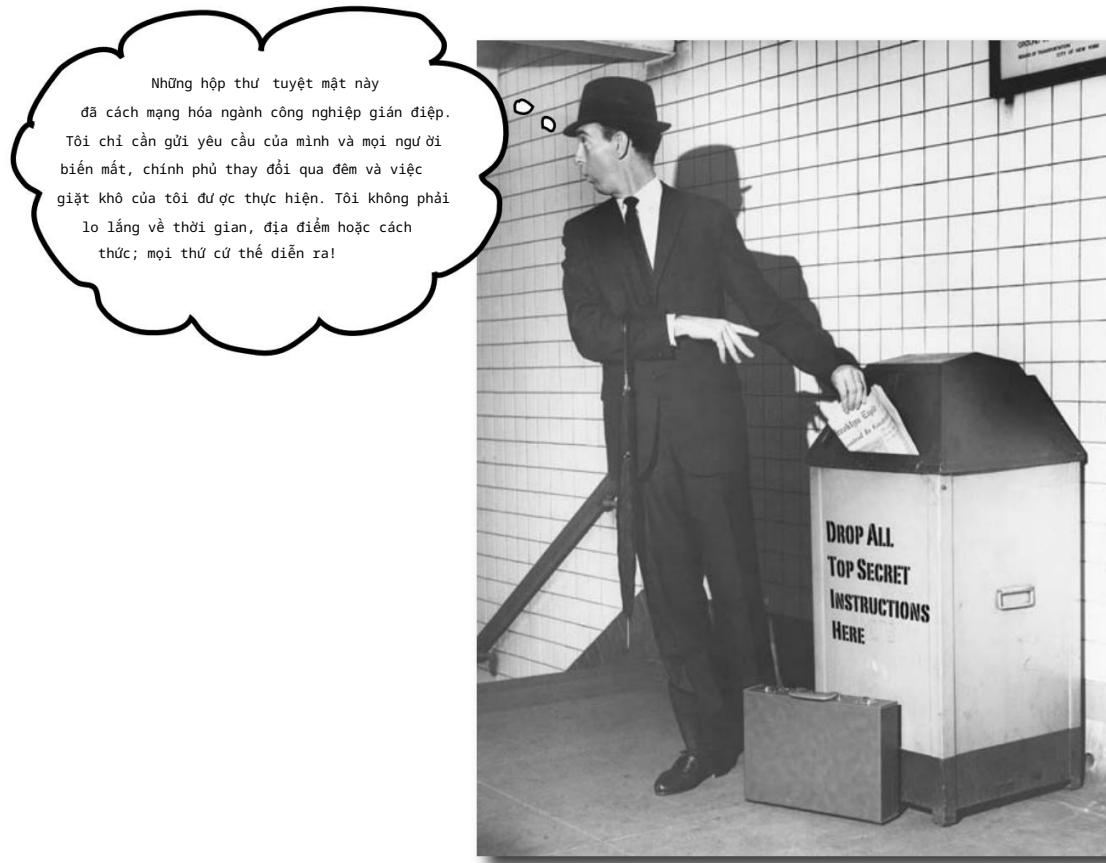
7777777777777777

77 777777 77 777777 777777 777777 777777 77777777

777

777

6 Mẫu lệnh h g Đóng gói lời gọi g

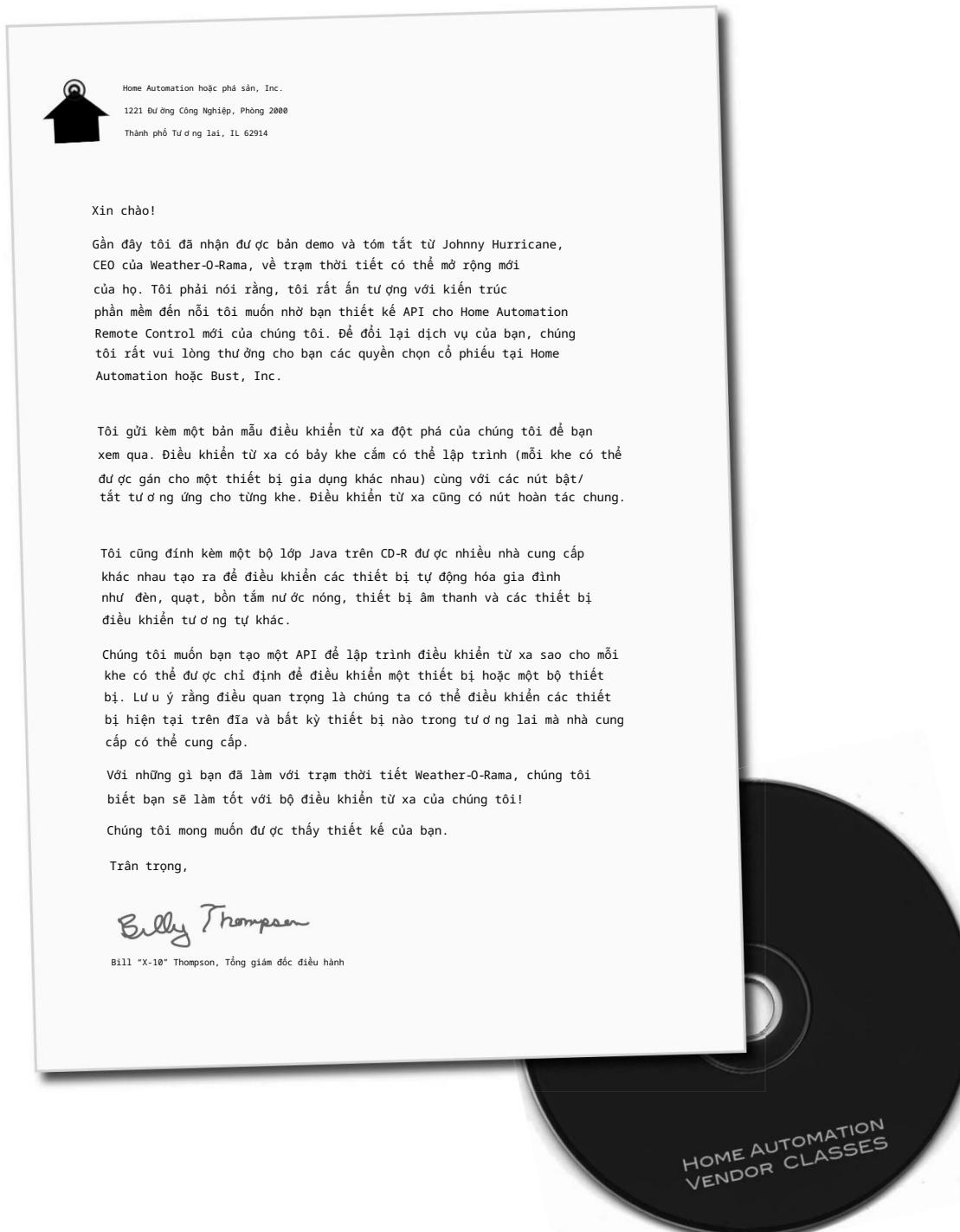


Trong chương này, chúng ta đưa việc đóng gói lên một cấp độ hoàn toàn mới: chúng ta sẽ đóng gói việc gọi
phương thức. Đúng vậy, bằng cách
đóng gói phương thức gọi, chúng ta có thể kết tinh các phần tinh toán để
đổi tương ứng phép tính không cần phải lo lắng về cách thực hiện mọi việc, nó chỉ sử dụng
phương pháp kết tinh của chúng tôi để thực hiện nó. Chúng tôi cũng có thể làm một số điều thông minh độc ác với
những lời gọi phương thức được đóng gói này, như lưu chúng lại để ghi nhật ký hoặc tái sử dụng chúng để
thực hiện hoàn tác trong mã của chúng tôi.

đây là một chương mới

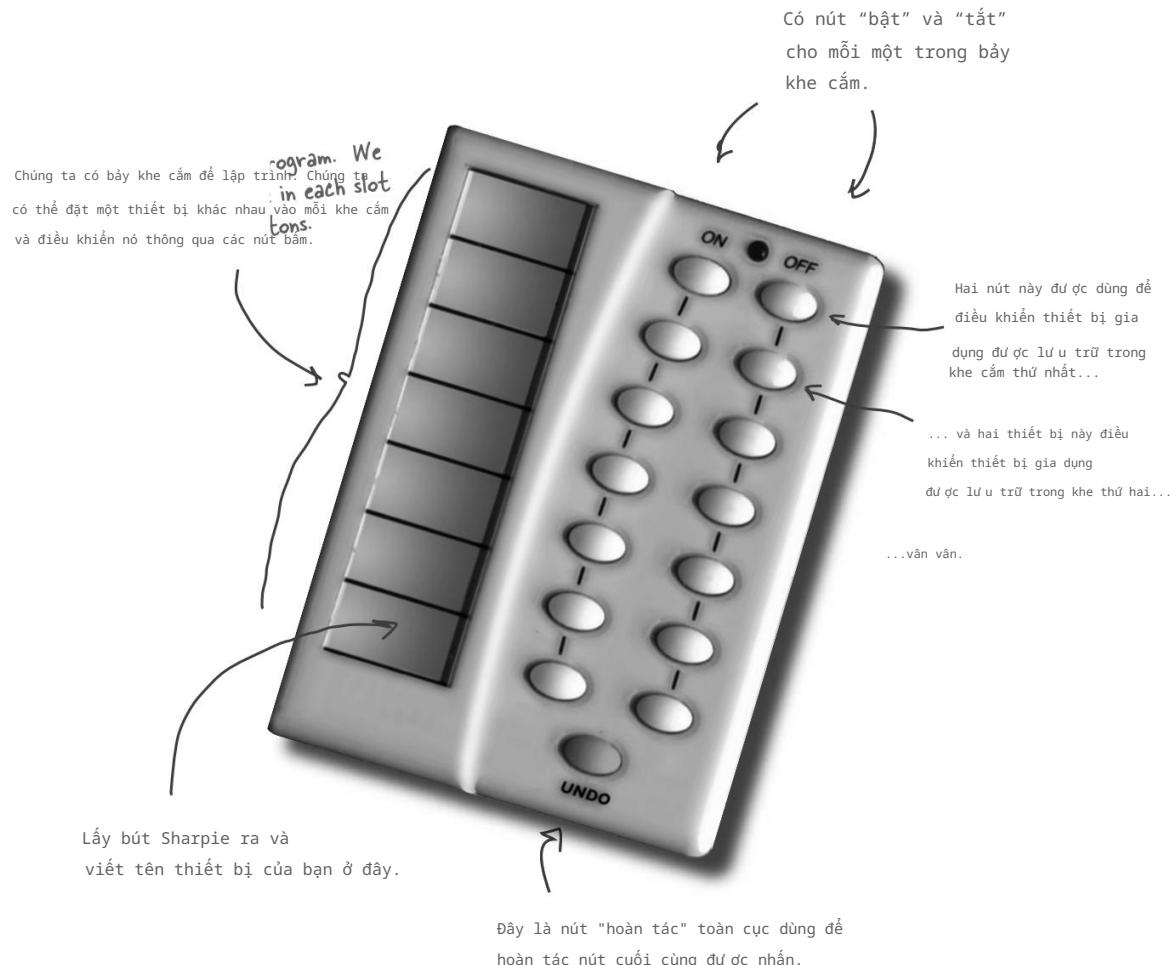
191

tự động hóa nhà hoặc phá sản



mẫu lệnh

Phần cứng miễn phí! Hãy cùng xem xét Remote Control...

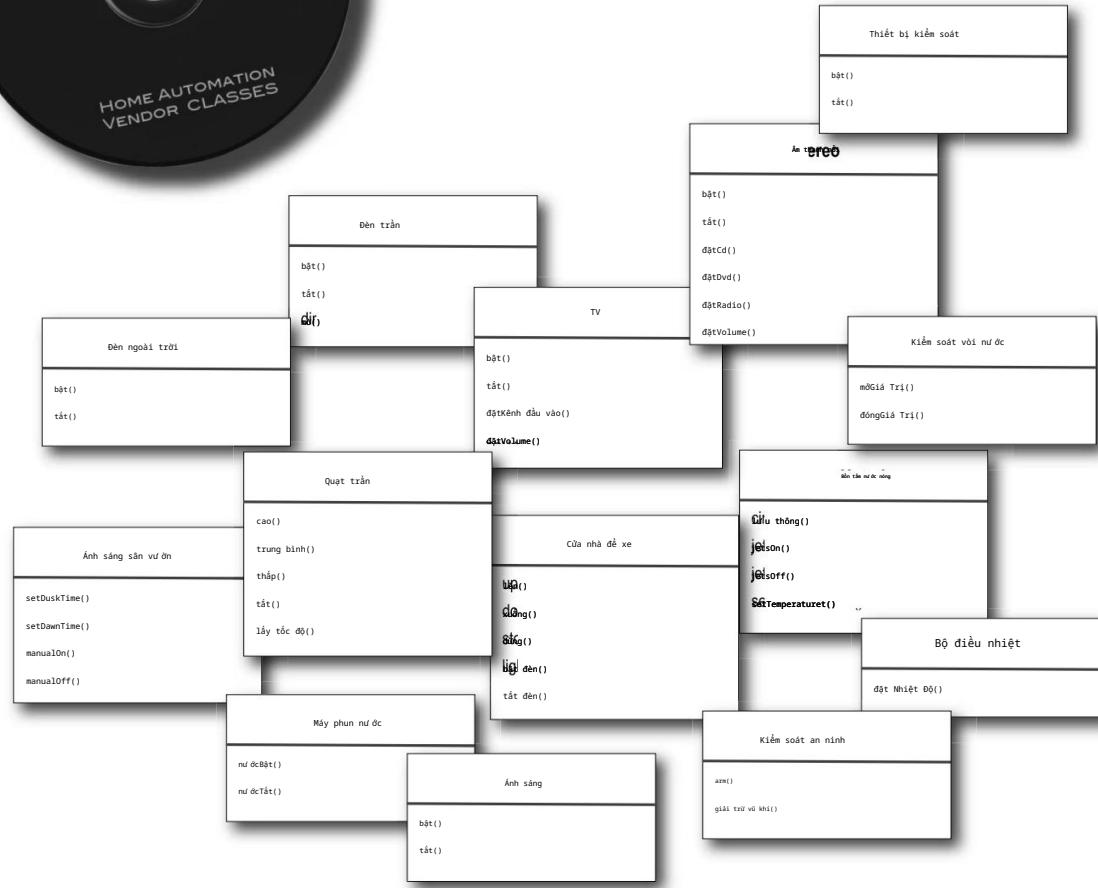


lớp nhà cung cấp từ tự động hóa gia đình



Xem xét các lớp nhà cung cấp

Kiểm tra các lớp nhà cung cấp trên CD-R. Chúng sẽ cung cấp cho bạn một số ý tưởng về giao diện của các đối tượng mà chúng ta cần điều khiển từ xa.

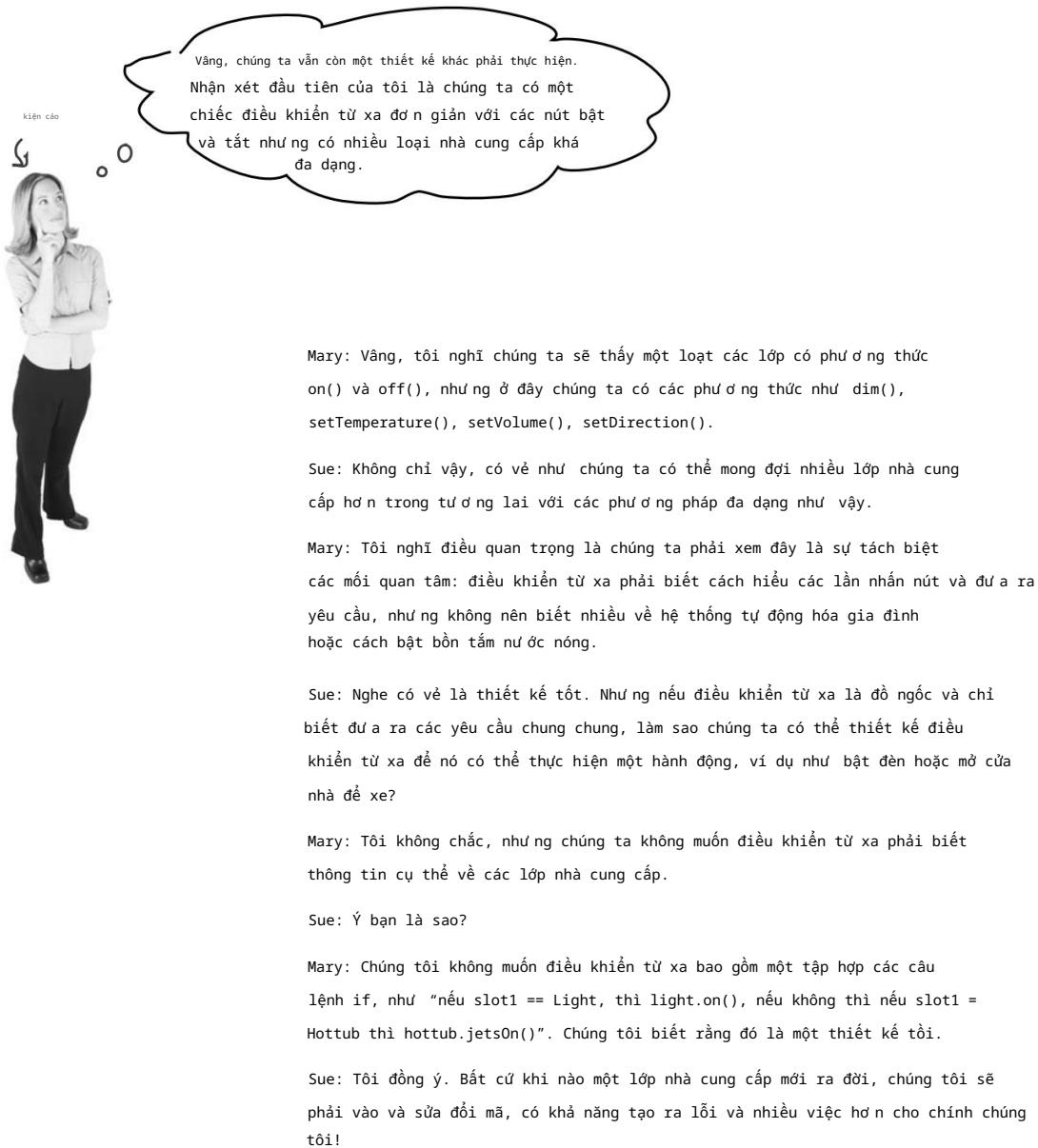


Có vẻ như chúng ta có khá nhiều lớp ở đây, và không có nhiều nỗ lực của ngành để đưa ra một bộ giao diện chung. Không chỉ vậy, có vẻ như chúng ta có thể mong đợi nhiều lớp hơn trong tương lai. Thiết kế API điều khiển từ xa sẽ rất thú vị.

Chúng ta hãy bắt đầu thiết kế.

Cuộc trò chuyện trong ô

Các đồng đội của bạn đang thảo luận về cách thiết kế API điều khiển từ xa...



mẫu lệnh có thể hoạt động



Mary: Vâng? Hãy kể cho chúng tôi thêm nhé.

Joe: Mẫu lệnh cho phép bạn tách người yêu cầu hành động khỏi đối tượng thực sự thực hiện hành động. Vì vậy, ở đây người yêu cầu sẽ là điều khiển từ xa và đối tượng thực hiện hành động sẽ là một thể hiện của một trong các lớp nhà cung cấp của bạn.

Sue: Làm sao có thể như vậy đư ợc? Làm sao chúng ta có thể tách chúng ra? Rốt cuộc, khi tôi nhấn một nút, điều khiển từ xa phải bật đèn.

Joe: Bạn có thể làm điều đó bằng cách đưa "đối tượng lệnh" vào thiết kế của mình. Một đối tượng lệnh đóng gói một yêu cầu thực hiện một việc gì đó (như bật đèn) trên một đối tượng cụ thể (ví dụ, đối tượng đèn phòng khách). Vì vậy, nếu chúng ta lưu trữ một đối tượng lệnh cho mỗi nút, khi nút đư ợc nhấn, chúng ta sẽ yêu cầu đối tượng lệnh thực hiện một số công việc. Điều khiển từ xa không biết công việc đó là gì, nó chỉ có một đối tượng lệnh biết cách giao tiếp với đúng đối tượng để hoàn thành công việc.

Vì vậy, bạn thấy đấy, điều khiển từ xa đã đư ợc tách khỏi vật thể sáng!

Sue: Nghe có vẻ như mọi việc đang đi đúng hướng.

Mary: Tuy nhiên, tôi vẫn đang gặp khó khăn trong việc hiểu đư ợc mô hình này.

Joe: Vì các vật thể tách biệt nhau nên hơi khó để hình dung cách thức hoạt động thực sự của mô hình này.

Mary: Để tôi xem liệu tôi có ít nhất là có ý tưởng đúng không: sử dụng mẫu này, chúng ta có thể tạo một API trong đó các đối tượng lệnh này có thể đư ợc tải vào các khe nút, cho phép mã từ xa vẫn rất đơn giản. Và, các đối tượng lệnh đóng gói cách thực hiện tác vụ tự động hóa tại nhà cùng với đối tượng cần thực hiện tác vụ đó.

Joe: Vâng, tôi nghĩ vậy. Tôi cũng nghĩ mẫu này có thể giúp bạn với nút Hoàn tác, nhưng tôi chưa học phần đó.

Mary: Nghe có vẻ rất đáng khích lệ, nhưng tôi nghĩ mình cần phải nỗ lực hơn nữa để thực sự "hiểu" đư ợc mô hình.

Sue: Tôi cũng vậy.

mẫu lệnh

Trong khi đó, quay lại Diner...,

hoặc, Giới thiệu tóm tắt về Command Pattern

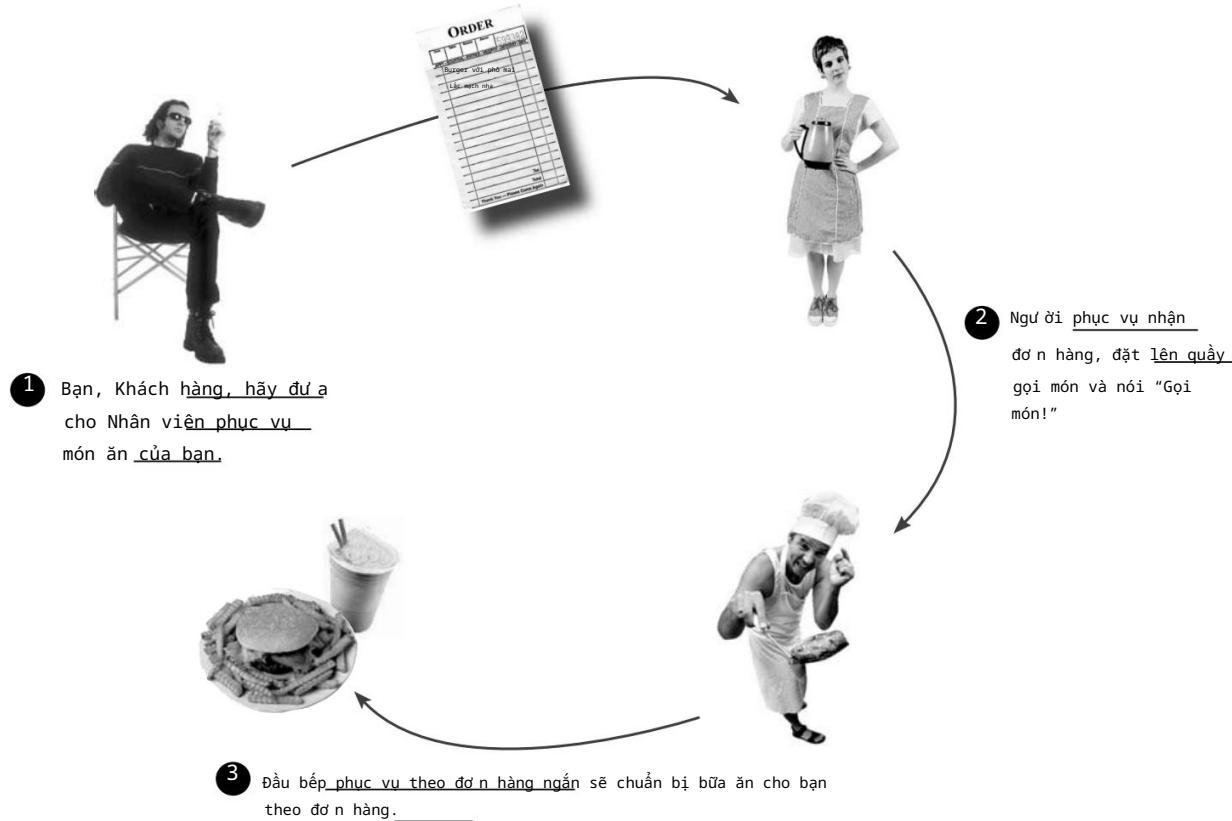
Như Joe đã nói, thật khó để hiểu được Mẫu lệnh chỉ bằng cách nghe mô tả. Như ng đừng lo, chúng ta có một số người bạn sẵn sàng giúp đỡ: bạn còn nhớ người phục vụ thân thiện của chúng ta trong Chương 1 không? Đã lâu rồi chúng ta không ghé thăm Alice, Flo và đầu bếp phục vụ đồ ăn nhanh, như chúng ta có lý do chính đáng để quay lại (vâng, ngoài đồ ăn và cuộc trò chuyện tuyệt vời): người phục vụ sẽ giúp chúng ta hiểu được Mẫu lệnh.

Vậy, hãy quay lại quán ăn một chút và nghiên cứu các tương tác giữa khách hàng, nhân viên phục vụ, các đơn hàng và đầu bếp phục vụ nhanh. Thông qua các tương tác này, bạn sẽ hiểu được các đối tượng liên quan đến Mẫu lệnh và cũng cảm nhận được cách tách rời hoạt động. Sau đó, chúng ta sẽ loại bỏ API điều khiển từ xa đó.



Đang kiểm tra tại Objectville Diner...

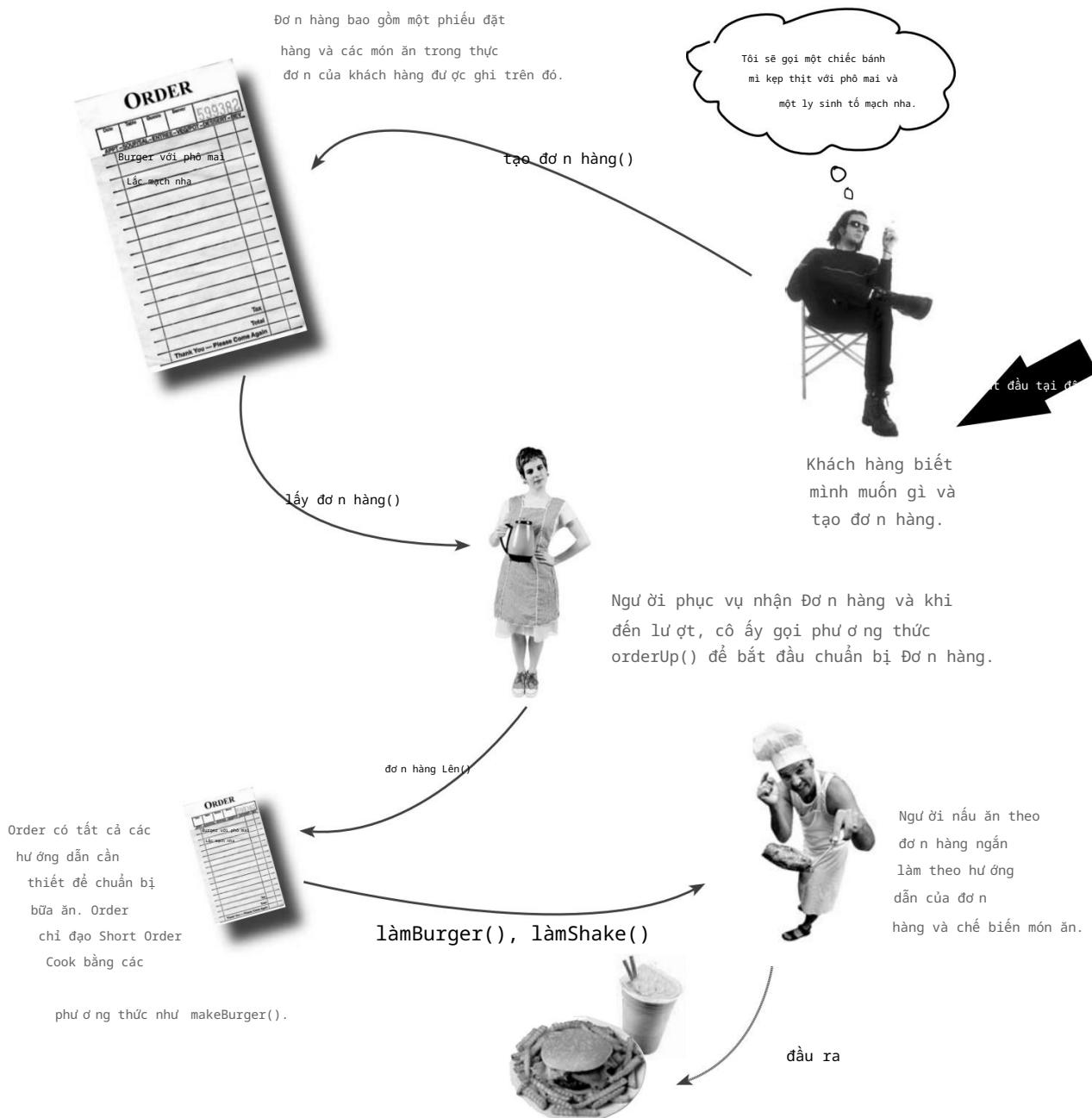
Để rồi, chúng ta đều biết Diner hoạt động như thế nào:



quán ăn

Chúng ta hãy cùng nghiên cứu sự tự o ng tác này chi tiết hơn một chút...

...và vì Diner này nằm trong Objectville, chúng ta hãy cùng suy nghĩ về các đối tượng và phư o ng thức đư ợc gọi liên quan nhé!

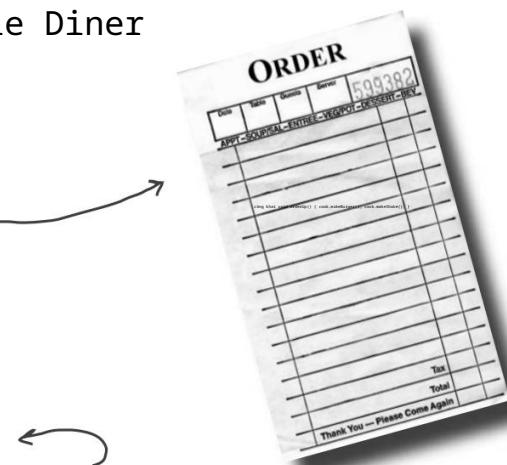


mẫu lệnh

Vai trò và trách nhiệm của Objectville Diner

Phiếu đặt hàng nêu rõ yêu cầu chuẩn bị một bữa ăn.

Hãy nghĩ về Phiếu đặt hàng như một đối tượng, một đối tượng hoạt động như một yêu cầu chuẩn bị một bữa ăn. Giống như bất kỳ đối tượng nào, nó có thể được truyền xung quanh - từ Nữ phục vụ đến quầy đặt hàng, hoặc đến Nữ phục vụ tiếp theo tiếp quản ca làm việc của mình. Nó có một giao diện chỉ bao gồm một phương thức, `orderUp()`, đóng gói các hành động cần thiết để chuẩn bị bữa ăn. Nó cũng có một tham chiếu đến đối tượng cần chuẩn bị nó (trong trường hợp của chúng ta, là Cook). Nó được đóng gói theo cách Nữ phục vụ không cần biết những gì có trong đơn đặt hàng hoặc thậm chí ai chuẩn bị bữa ăn; cô ấy chỉ cần truyền phiếu qua cửa sổ đặt hàng và gọi "Đặt hàng!"



Công việc của nhân viên phục vụ là lấy phiếu gọi món và gọi phuơng thức `orderUp()` trên đó.

quan tâm đến những gì đang diễn ra
Được rồi, trong cuộc sống thực một cô hầu bàn có lẽ sẽ
Phiếu đặt hàng và người nấu nó, nhưng đây
là Objectville... hãy làm việc với chúng tôi tại đây!

Waitress có cách làm dễ dàng: nhận `order` từ khách hàng, tiếp tục phục vụ khách hàng cho đến khi khách hàng quay lại quầy `order`, sau đó gọi phuơng thức `orderUp()` để chuẩn bị bữa ăn. Như chúng ta đã thảo luận, trong Objectville, Waitress thực sự không lo lắng về những gì có trong `order` hoặc ai sẽ chuẩn bị; cô ấy chỉ biết rằng các phiếu `order` có phuơng thức `orderUp()` mà cô ấy có thể gọi để hoàn thành công việc.

Bây giờ, trong suốt cả ngày, phuơng thức `takeOrder()` của Người phục vụ được tham số hóa với các phiếu đặt hàng khác nhau từ nhiều khách hàng khác nhau, như người đó không làm cô ấy bối rối; cô ấy biết rằng tất cả các phiếu đặt hàng đều hỗ trợ phuơng thức `orderUp()` và cô ấy có thể gọi `orderUp()` bất cứ khi nào cô ấy cần chuẩn bị một bữa ăn.

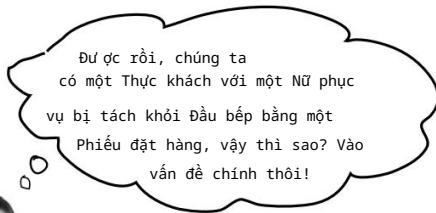
Người nấu ăn theo đơn đặt hàng ngắn có kiến thức cần thiết để chuẩn bị bữa ăn.

Short Order Cook là đối tượng thực sự biết cách chế biến bữa ăn. Khi Waitress đã gọi phuơng thức `orderUp()`; Short Order Cook sẽ tiếp quản và triển khai tất cả các phuơng thức cần thiết để tạo ra bữa ăn.

Lưu ý rằng Người phục vụ và Người nấu ăn hoàn toàn tách biệt: Người phục vụ có Phiếu gọi món tóm tắt thông tin chi tiết về bữa ăn; cô ấy chỉ cần gọi một phuơng thức cho mỗi đơn hàng để chuẩn bị món ăn. Tương tự như vậy, Đầu bếp nhận hướng dẫn từ Phiếu đặt hàng; anh ta không bao giờ cần phải giao tiếp trực tiếp với Người phục vụ.



quán ăn là một mô hình cho mẫu lệnh



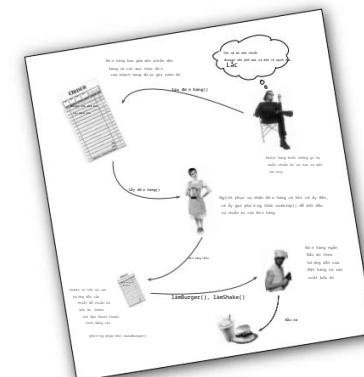
Kiên nhẫn, chúng ta sắp đạt đư ợc iồi...

Hãy nghĩ về Diner như một mô hình cho một mẫu thiết kế cho phép chúng ta tách một đối tượng tạo ra yêu cầu khỏi các đối tượng nhận và thực hiện các yêu cầu đó. Ví dụ, trong API điều khiển từ xa của chúng ta, chúng ta cần tách mà đư ợc gọi khi chúng ta nhấn nút khỏi các đối tượng của các lớp cụ thể của nhà cung cấp thực hiện các yêu cầu đó. Điều gì sẽ xảy ra nếu mỗi khe cắm của điều khiển từ xa chứa một đối tượng như đối tượng phiếu đặt hàng của Diner? Sau đó, khi nhấn nút, chúng ta có thể chỉ cần gọi phương thức `orderUp()` với phương thức "orderUp()" trên đối tượng này và bật đèn mà không cần điều khiển từ xa biết chi tiết về cách thực hiện những điều đó hoặc những đối tượng nào đang thực hiện chúng.

Bây giờ, chúng ta hãy chuyển chủ đề một chút và liên kết tất cả cuộc trò chuyện trong Diner này với Command Pattern...

não Apower

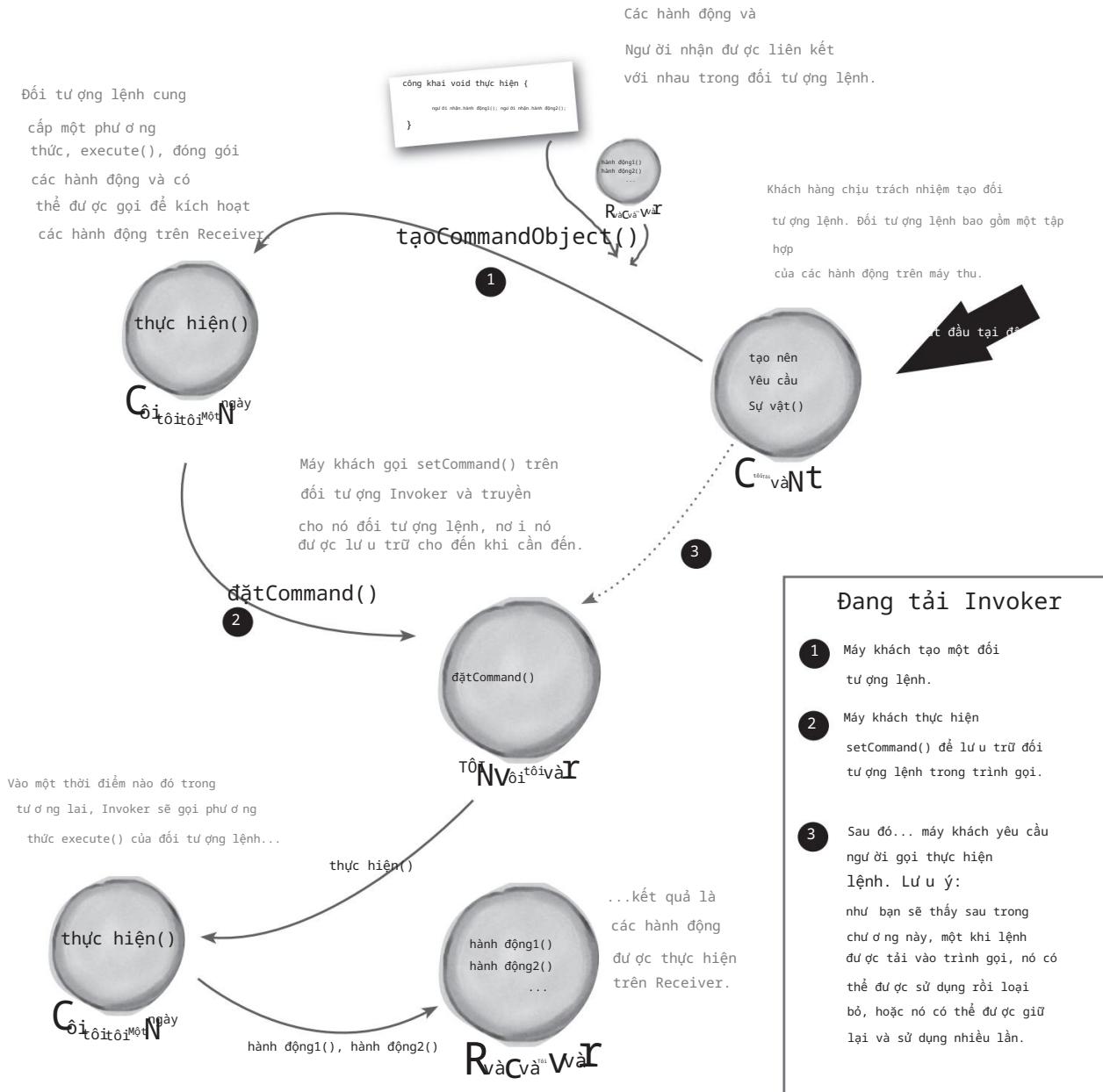
Trước khi tiếp tục, hãy dành thời gian nghiên cứu sơ đồ ở hai trang trước cùng với các vai trò và trách nhiệm của Diner cho đến khi bạn nghĩ mình đã nắm đư ợc các đối tượng và mối quan hệ của Objectville Diner. Sau khi hoàn thành, hãy chuẩn bị để nắm vững Command Pattern!



mẫu lệnh

Từ Diner đến Command Pattern

Đến đây rồi, chúng ta đã dành đủ thời gian trong Objectville Diner để biết rõ ràng cả các tính cách và trách nhiệm của họ. Bây giờ chúng ta sẽ làm lại sơ đồ Diner để phản ánh Command Pattern. Bạn sẽ thấy rằng tất cả người chơi đều giống nhau; chỉ có tên là thay đổi.



Bạn đang ở đây 4 201

Ai làm gì?

* WHO DOES ? WHAT ? *

Ghép các đối tượng và phương thức của quán ăn với các tên tương ứng từ Mẫu lệnh.

Quán ăn

Mẫu lệnh

Cô hầu bàn

Yêu cầu

Nấu ăn theo đơn đặt hàng ngắn

thực hiện()

đơn hàng Lên()

Khách hàng

Đặt hàng

Người triệu hồi

Khách hàng

Người nhận

lấy đơn hàng()

đặtCommand()

mẫu lệnh

Đối tượng lệnh đầu tiên của chúng tôi

Đã đến lúc chúng ta xây dựng đối tượng lệnh đầu tiên của mình chưa? Hãy tiếp tục và viết một số mã cho điều khiển từ xa. Mặc dù chúng ta vẫn chưa tìm ra cách thiết kế API điều khiển từ xa, việc xây dựng một vài thứ từ dưới lên có thể giúp chúng ta...



Triển khai giao diện lệnh

Trước tiên là: tất cả các đối tượng lệnh đều triển khai cùng một giao diện, bao gồm một phương thức. Trong Diner, chúng tôi gọi phương thức này là `orderUp()`; tuy nhiên, chúng tôi thường chỉ sử dụng tên `execute()`.

Đây là giao diện Lệnh:

```
Giao diện công khai Lệnh {
    công khai void thực hiện();
}

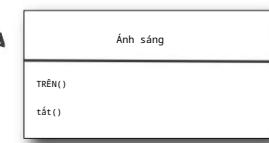
```

Đơn giản. Tất cả những gì chúng ta cần là một phương thức gọi là `execute()`.

Thực hiện lệnh bật đèn

Bây giờ, giả sử bạn muốn thực hiện lệnh bật đèn.

Tham khảo tập hợp các lớp nhà cung cấp của chúng tôi, lớp Light có hai phương thức: `on()` và `off()`. Sau đây là cách bạn có thể triển khai điều này dưới dạng lệnh:



```
lớp công khai LightOnCommand thực hiện Command {
    Ánh sáng nhẹ;

    công khai LightOnCommand(Ánh sáng ánh sáng) {
        this.light = ánh sáng;
    }

    công khai void thực hiện() {
        sáng.bật();
    }
}
```

Đây là một lệnh, do đó chúng ta cần triển khai giao diện Lệnh.

Trình xây dựng đư ợc truyền ánh sáng cụ thể mà lệnh này sẽ điều khiển - ví dụ như đèn phòng khách - và lưu nó trong biến thể hiện ánh sáng. Khi `execute` đư ợc gọi, đây là đối tượng ánh sáng sẽ là Receiver của yêu cầu.

Phương thức `execute` gọi phương thức `on()` trên đối tượng nhận, đó là ánh sáng mà chúng ta đang kiểm soát.

Bây giờ bạn đã có lớp `LightOnCommand`, hãy xem liệu chúng ta có thể sử dụng nó hay không...

sử dụng lệnh đối tượng

Sử dụng đối tượng lệnh

Để rõ ràng, chúng ta hãy làm cho mọi thứ đơn giản hơn: giả sử chúng ta có một chiếc điều khiển từ xa chỉ có một nút và khe tự động ứng để giữ thiết bị cần điều khiển:

```
lớp công khai SimpleRemoteControl {
    Khe cắm lệnh;

    công khai SimpleRemoteControl() {}

    công khai void setCommand(Lệnh lệnh) {
        slot = lệnh;
    }

    nút công khai voidWasPressed() {
        slot. thực thi();
    }
}
```

Chúng ta có một khe để giữ lệnh, khe này sẽ điều khiển một thiết bị.

Chúng tôi có một phương pháp để thiết lập lệnh mà khe cắm sẽ điều khiển. Phương pháp này có thể được gọi nhiều lần nếu máy khách của mã này muốn thay đổi hành vi của nút điều khiển từ xa.

Phương pháp này được gọi khi nút được nhấn. Tất cả những gì chúng ta làm là lấy lệnh hiện tại được liên kết với khe và gọi phương thức execute() của nó.

Tạo một bài kiểm tra đơn giản để sử dụng Điều khiển từ xa

Đây chỉ là một đoạn mã nhỏ để kiểm tra điều khiển từ xa đơn giản. Chúng ta hãy cùng xem và chúng tôi sẽ chỉ ra cách các mảnh ghép khớp với sơ đồ Mẫu lệnh:

```
lớp công khai RemoteControlTest {
    public static void main(String[] args) {
        SimpleRemoteControl từ xa = new SimpleRemoteControl();
        Ánh sáng ánh sáng = Ánh sáng mới();
        LightOnCommand lightOn = new LightOnCommand(ánh sáng);

        remote.setCommand(lightOn);
        remote.buttonWasPressed();
    }
}
```

Đây chính là mô hình Khách hàng chỉ huy của chúng tôi. Remote là Invoker của chúng ta; nó sẽ được truyền một đối tượng lệnh có thể được sử dụng để thực hiện các yêu cầu.

Bây giờ chúng ta tạo một đối tượng Light, đây sẽ là Người nhận yêu cầu.

Tại đây, hãy tạo một lệnh và truyền Receiver cho lệnh đó.

Và sau đó chúng ta mở phông nút đang được nhấn.

Ở đây, hãy truyền lệnh cho Invoker.

Đây là kết quả khi chạy mã thử nghiệm này!

```
Cửa sổ chỉnh sửa tệp Tạo giúp DinerFoodYum
%java Kiểm tra điều khiển từ xa
Ánh sáng đang bật
%
```

mẫu lệnh



Chuột bút chì của bạn

Đư ợc rồi, đã đến lúc bạn triển khai lớp GarageDoorOpenCommand. Trước tiên, cung cấp mã cho lớp bên dưới.
Bạn sẽ cần sơ đồ lớp GarageDoor.

```
lớp công khai GarageDoorOpenCommand thực hiện
Command {
```

Cửa nhà để xe
lên()
xuống()
đóng()
bật sáng()
tắt sáng()

}



Mã của bạn ở đây

Bây giờ bạn đã có lớp của mình, đầu ra của đoạn mã sau là gì? (Gợi ý:
phương thức up() của GarageDoor sẽ in ra "Garage Door is Open" khi hoàn
tất.)

```
lớp công khai RemoteControlTest { công khai
    tĩnh void main(String[] args) {
        SimpleRemoteControl từ xa = new SimpleRemoteControl();
        Ánh sáng ánh sáng = Ánh sáng mới();
        Cửa Garage garageDoor = new GarageDoor();
        LightOnCommand lightOn = new LightOnCommand(ánh sáng);
        GarageDoorOpenCommand garageOpen =
            GarageDoorOpenCommand(garageDoor) mới;
```

```
        từ xa. setCommand(lightOn); từ xa.
        buttonWasPressed(); từ xa.
        setCommand(garageOpen); từ xa.
        buttonWasPressed();
    }
}
```

Đầu ra của bạn ở đây.

Cửa sổ chỉnh sửa tệp Tạo giúp GreenEggs&Ham
%java Kiểm tra điều khiển từ xa



mẫu lệnh đư ợc xác định

Mẫu lệnh đư ợc xác định

Bạn đã hoàn thành thời gian của mình trong Objectville Diner, bạn đã triển khai một phần API điều khiển từ xa và trong quá trình này, bạn đã có một bức tranh khá tốt về cách các lớp và đối tượng tương tác trong Command Pattern. Ngày giờ chúng ta sẽ định nghĩa Command Pattern và hoàn thiện tất cả các chi tiết.

Chúng ta hãy bắt đầu với định nghĩa chính thức của nó:

Mẫu lệnh đóng gói một yêu cầu dưới dạng một đối tượng, do đó cho phép bạn tham số hóa các đối tượng khác với các yêu cầu khác nhau, xếp hàng hoặc ghi nhật ký yêu cầu và hỗ trợ các hoạt động có thể hoàn tác.

Hãy cùng tìm hiểu điều này. Chúng ta biết rằng một đối tượng lệnh đóng gói một yêu cầu bằng cách liên kết một tập hợp các hành động trên một bộ thu cụ thể. Để đạt được điều này, nó đóng gói các hành động và bộ thu thành một đối tượng chỉ hiển thị một phương thức, execute().

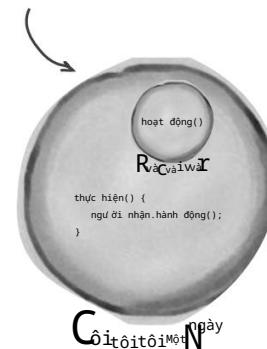
Khi được gọi, execute() khiến các hành động đư ợc gọi trên máy thu. Từ bên ngoài, không có đối tượng nào khác thực sự biết hành động nào đư ợc thực hiện trên máy thu nào; chúng chỉ biết rằng nếu chúng gọi phương thức execute(), yêu cầu của chúng sẽ đư ợc phục vụ.

Chúng ta cũng đã thấy một vài ví dụ về việc tham số hóa một đối tượng bằng lệnh. Quay trở lại quán ăn, Waitress đã đư ợc tham số hóa với nhiều đơn hàng trong suốt cả ngày. Trong điều khiển từ xa đơn giản, đầu tiên chúng tôi đã tải khe nút bằng lệnh "bật đèn" và sau đó thay thế bằng lệnh "mở cửa nhà để xe".

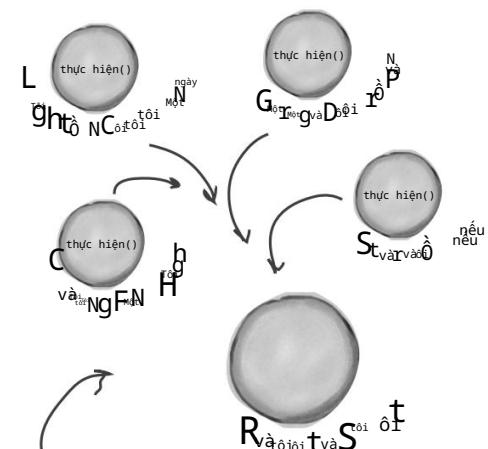
Giống như Waitress, khe cắm từ xa của bạn không quan tâm đến đối tượng lệnh mà nó có, miễn là nó triển khai giao diện Lệnh.

Nhưng gì chúng ta chưa gặp phải là sử dụng lệnh để triển khai hàng đợi và nhật ký và hỗ trợ các hoạt động hoàn tác. Điều lo lắng, đó là những phần mở rộng khá đơn giản của Mẫu lệnh cơ bản và chúng ta sẽ sớm tìm hiểu về chúng. Chúng ta cũng có thể dễ dàng hỗ trợ cái đư ợc gọi là Mẫu lệnh Meta sau khi chúng ta đã có những kiến thức cơ bản. Mẫu lệnh Meta cho phép bạn tạo các lệnh macro để bạn có thể thực thi nhiều lệnh cùng một lúc.

Một yêu cầu đư ợc đóng gói.



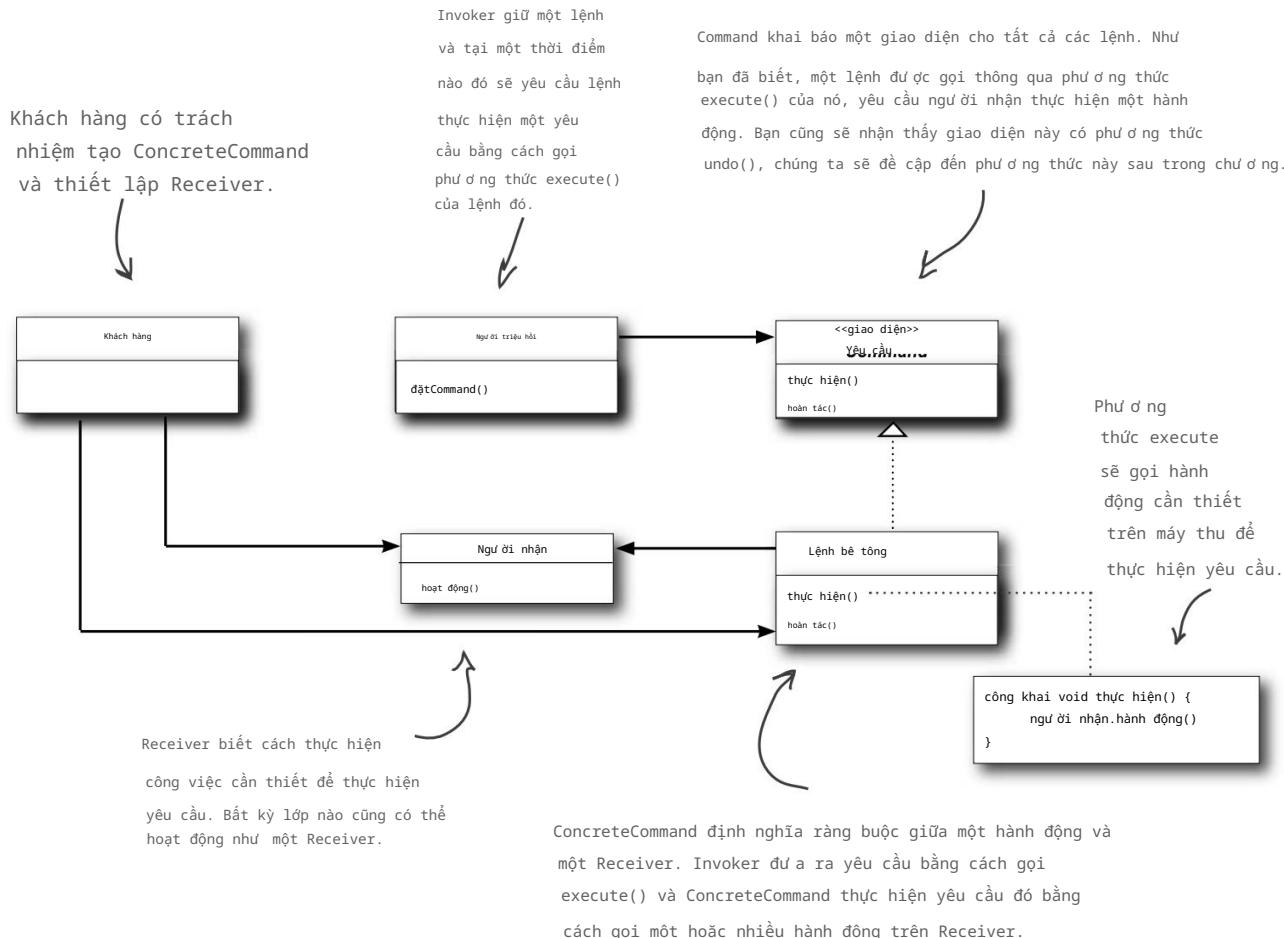
Có một ngày



Người gọi - ví dụ như một khe cắm của máy điều khiển từ xa - có thể đư ợc tham số hóa bằng các yêu cầu khác nhau.

mẫu lệnh

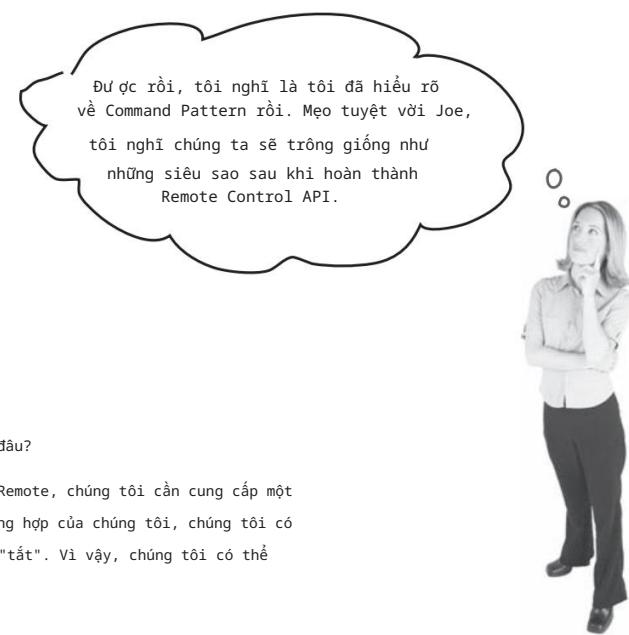
Mẫu lệnh đư ợc định nghĩa: sơ đồ lớp



não Apower

Thiết kế của Mẫu lệnh hỗ trợ việc tách biệt bên gọi yêu cầu và bên nhận yêu cầu như thế nào?

chúng ta bắt đầu từ đâu ?



Mary: Tôi cũng vậy. Vậy chúng ta bắt đầu từ đâu?

Sue: Giống như chúng tôi đã làm trong SimpleRemote, chúng tôi cần cung cấp một cách để gán lệnh cho các khe cắm. Trong trường hợp của chúng tôi, chúng tôi có bảy khe cắm, mỗi khe cắm có một nút "bật" và "tắt". Vì vậy, chúng tôi có thể gán lệnh cho điều khiển từ xa như thế này:

```
onCommands[0] = onCommand; offCommands[0] =  
offCommand;
```

Mary: Có lý, ngoại trừ các vật thể Ánh sáng. Làm sao điều khiển từ xa biết đư ợc phòng khách và đèn bếp?

Sue: À, đúng rồi, không phải vậy! Remote không biết gì ngoài cách gọi execute() trên đối tượng lệnh tương ứng khi một nút đư ợc nhấn.

Mary: Vâng, tôi hiểu phần nào, nhưng trong quá trình triển khai, làm sao chúng ta có thể đảm bảo đúng đối tượng đang bật và tắt đúng thiết bị?

Sue: Khi chúng ta tạo các lệnh để tải vào điều khiển từ xa, chúng ta tạo một LightCommand đư ợc liên kết với đối tượng đèn phòng khách và một LightCommand khác đư ợc liên kết với đối tượng đèn bếp. Hãy nhớ rằng, người nhận yêu cầu sẽ bị liên kết với lệnh mà nó đư ợc đóng gói trong đó. Vì vậy, khi nút đư ợc nhấn, không ai quan tâm đến đèn nào là đèn nào, điều đúng đắn chỉ xảy ra khi phu ơ ng thực execute() đư ợc gọi.

Mary: Tôi nghĩ là tôi hiểu rồi. Hãy triển khai điều khiển từ xa và tôi nghĩ điều này sẽ rõ ràng hơn!

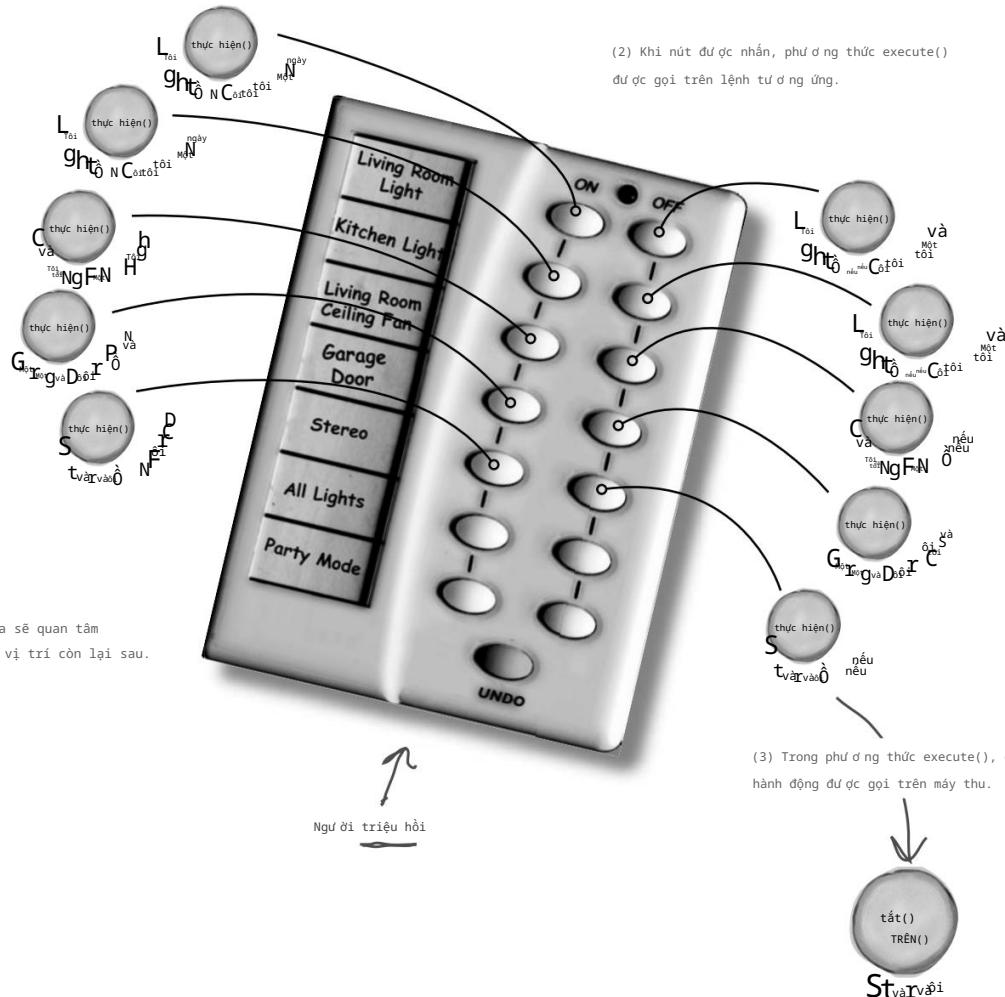
Sue: Nghe có vẻ ổn đấy. Hãy thử xem...

mẫu lệnh

Gán lệnh cho các khe

Vì vậy, chúng ta có một kế hoạch: Chúng ta sẽ gán mỗi khe cho một lệnh trong điều khiển từ xa. Điều này biến điều khiển từ xa thành invoker của chúng ta. Khi một nút được nhấn, phương thức execute() sẽ được gọi trên lệnh tương ứng, dẫn đến các hành động được gọi trên bộ thu (như đèn, quạt trần, dàn âm thanh nổi).

(1) Mỗi ô có một lệnh.



thực hiện điều khiển từ xa

Thực hiện điều khiển từ xa

```
lớp công khai RemoteControl {
    Lệnh[] trên Lệnh;
    Lệnh[] tắtLệnh;
```

Lần này, điều khiển từ xa sẽ xử lý
bảy lệnh Bật và Tắt, chúng ta sẽ giữ
chúng trong các mảng tương ứng.

```
công khai RemoteControl() {
    onCommands = Lệnh mới[7];
    offCommands = Lệnh mới[7];

    Lệnh noCommand = new NoCommand();
    đối với (int i = 0; i < 7; i++) {
        onCommands[i] = noCommand;
        offCommands[i] = noCommand;
    }
}
```

Trong hàm tạo, tắt cả những gì chúng ta
cần làm là khởi tạo và khởi tạo on và off
mảng.

```
public void setCommand(int khe, Lệnh onCommand, Lệnh offCommand) {
    onCommands[khe] = onCommand;
    offCommands[khe] = offCommand;
}
```

Phương thức setCommand() lấy một vị trí khe cắm
và một lệnh Bật và Tắt để lưu trữ trong khe
cắm đó. Nó đặt các lệnh này vào mảng bật và tắt
để sử dụng sau.

```
công khai void onButtonWasPushed(int slot) {
    onCommands[slot].thực thi();
}

công khai void offButtonWasPushed(int slot) {
    tắt Lệnh[slot].thực thi();
}
```

Khi nhấn nút Bật hoặc Tắt,
phần cứng sẽ xử lý việc gọi
các phương thức tương ứng
onButtonWasPushed() hoặc
offButtonWasPushed().

```
công khai String toString() {
    StringBuffer stringBuff = new StringBuffer();
    stringBuff.append("\n----- Điều khiển từ xa -----");
    đối với (int i = 0; i < onCommands.length; i++) {
        stringBuff.append("[khe cắm " + i + "] " + onCommands[i].getClass().getName()
            + " " + offCommands[i].getClass().getName() + "\n");
    }
    trả về stringBuff.toString();
}
```

Chúng tôi đã ghi đè toString() để in ra từng khe và lệnh
tương ứng. Bạn sẽ thấy chúng tôi sử dụng điều này khi kiểm
tra điều khiển từ xa.

mẫu lệnh

Thực hiện các lệnh

Vâng, chúng ta đã bắt đầu triển khai LightOnCommand cho SimpleRemoteControl. Chúng ta có thể cắm cùng mã đó vào dây và mọi thử hoạt động hoàn hảo. Lệnh tắt cũng không khác gì; thực tế LightOffCommand trông như thế này:

```
lớp công khai LightOffCommand thực hiện Command {
    Ánh sáng nhẹ;

    công khai LightOffCommand(Ánh sáng ánh sáng) {
        this.light = ánh sáng;
    }

    công khai void thực hiện() {
        tắt đèn();
    }
}
```

LightOffCommand hoạt động theo cùng một cách chính xác như LightOnCommand, ngoại trừ việc chúng ta ràng buộc bộ thu với một hành động khác: phương thức off().

Hãy thử một cái gì đó khó hơn một chút; tại sao không viết lệnh bật và tắt cho Stereo? Được rồi, tắt thì dễ, chúng ta chỉ cần liên kết Stereo với phương thức off() trong StereoOffCommand. Bật thì phức tạp hơn một chút; giả sử chúng ta muốn viết StereoOnWithCDCommand...

Âm thanh nội
TRÊN()
tắt()
đặtCD()
đặtDvd()
thiết lậpRadio()
đặtVolume()

```
lớp công khai StereoOnWithCDCommand thực hiện Command {
    Âm thanh nội âm thanh nội;

    công khai StereoOnWithCDCommand(Stereo âm thanh nội) {
        this.stereo = âm thanh nội;
    }

    công khai void thực hiện() {
        stereo.bật();
        stereo.setCD();
        stereo.setVolume(11);
    }
}
```

Giống như LightOnCommand, chúng ta truyền vào thẻ hiện của âm thanh nội mà chúng ta sẽ điều khiển và lưu trữ nó trong một biến thẻ hiện cục bộ.

Để thực hiện yêu cầu này, chúng ta cần gọi ba phương pháp trên dàn âm thanh: đầu tiên, bật dàn âm thanh lên, sau đó cài đặt để phát CD và cuối cùng cài đặt âm lượng ở mức 11. Tại sao lại là 11? Vâng, nó tốt hơn 10, đúng không?

Không tệ lắm. Hãy xem xét phần còn lại của các lớp nhà cung cấp; đến giờ, bạn chắc chắn có thể hoàn thành phần còn lại của các lớp Lệnh mà chúng ta cần cho chúng.

kiểm tra điều khiển từ xa

Đưa điều khiển từ xa vào thử nghiệm

Công việc của chúng tôi với điều khiển từ xa đã gần hoàn thành; tất cả những gì chúng tôi cần làm là chạy một số thử nghiệm và tập hợp một số tài liệu để mô tả API. Home Automation hoặc Bust, Inc. chắc chắn sẽ rất ấn tượng, bạn không nghĩ vậy sao? Chúng tôi đã đưa ra được một thiết kế cho phép họ sản xuất một điều khiển từ xa dễ bảo trì và họ sẽ không gặp khó khăn gì khi thuyết phục các nhà cung cấp viết một số lớp lệnh đơn giản trong tương lai vì chúng rất dễ viết.

Chúng ta hãy cùng kiểm tra đoạn mã này nhé!

```

lớp công khai RemoteLoader {

    public static void main(String[] args) {
        RemoteControl remoteControl = new RemoteControl();

        Ánh sáng livingRoomLight = new Light("Phòng khách");
        Bếp sángLight = new Light("Bếp");
        CeilingFan ceilingFan= new CeilingFan("Phòng khách");
        GarageDoor garageDoor = new GarageDoor("");
        Âm thanh nồi âm thanh nồi = âm thanh nồi mới("Phòng khách");

        LightOnCommand livingRoomLightOn = new
            LightOnCommand(livingRoomLight);
        LightOffCommand livingRoomLightOff = new
            LightOffCommand(livingRoomLight);
        LightOnCommand kitchenLightOn = new
            LightOnCommand(kitchenLight);
        LightOffCommand kitchenLightOff = lệnh LightOff
            mới(kitchenLight);

        CeilingFanOnCommand ceilingFanOn = lệnh mới
            CeilingFanOn(ceilingFan);
        Lệnh CeilingFanOff ceilingFanOff = lệnh mới
            CeilingFanOff(ceilingFan);

        GarageDoorUpLệnh garageDoorUp =
            GarageDoorUpCommand mới(garageDoor);
        GarageDoorDownLệnh garageDoorDown =
            GarageDoorDownCommand mới(garageDoor);

        StereoOnWithCDCommand stereoOnWithCD =
            lệnh StereoOnWithCDCommand(stereo);
        Lệnh StereoOff stereoOff =
            mới StereoOffCommand(stereo);
    }
}

```

The code is annotated with curly braces and explanatory text:

- A brace groups the first five lines of the class definition, with the annotation: "Đặt tất cả các thiết bị vào đúng vị trí của chúng."
- A brace groups the four Light-related command definitions, with the annotation: "Tạo tất cả các đối tượng Light Command."
- A brace groups the two CeilingFan command definitions, with the annotation: "Tạo nút Bật và Tắt cho quạt trần."
- A brace groups the two GarageDoor command definitions, with the annotation: "Tạo lệnh Lên và Xuống cho Garage."
- A brace groups the two Stereo command definitions, with the annotation: "Tạo lệnh Bật và Tắt âm thanh nồi."

mẫu lệnh

```
remoteControl.setCommand(0, Đèn phòng khách Bật, Đèn phòng khách Tắt);  
remoteControl.setCommand(1, kitchenLightOn, kitchenLightOff);  
remoteControl.setCommand(2, ceilingFanOn, ceilingFanOff);  
remoteControl.setCommand(3, stereoOnWithCD, stereoOff);  
  
System.out.println(Điều khiển từ xa);  
  
remoteControl.onButtonWasPushed(0);  
remoteControl.offButtonWasPushed(0);  
remoteControl.onButtonWasPushed(1);  
remoteControl.offButtonWasPushed(1);  
remoteControl.onButtonWasPushed(2);  
remoteControl.offButtonWasPushed(2);  
remoteControl.onButtonWasPushed(3);  
remoteControl.offButtonWasPushed(3);  
}  
}  
  
Bây giờ chúng ta đã có  
tất cả các lệnh, chúng  
ta có thể tải chúng  
vào các khe cắm từ xa.  
  
Đây là nơi chúng ta sử dụng phư ơng  
thức toString() để in ra từng khe cắm từ  
xa và lệnh mà khe cắm đó được gán cho.  
  
Được rồi, chúng ta đã sẵn sàng rồi!  
Bây giờ, chúng ta bù ớc qua từng khe  
và nhấn nút Bật và Tắt.
```

Bây giờ, chúng ta hãy kiểm tra quá trình thực hiện thử nghiệm điều khiển từ xa của chúng ta...

```
Cửa sổ chính sửa tệp Trợ giúp LệnhGetThingsDone

% java RemoteLoader
----- Điều khiển từ xa -----
[khe 0] headfirst.command.remote.LightOnCommand headfirst.command.remote.LightOffCommand
[khe 1] headfirst.command.remote.LightOnCommand headfirst.command.remote.LightOffCommand
[khe 2] headfirst.command.remote.CeilingFanOnCommand headfirst.command.remote.CeilingFanOffCommand
[khe 3] headfirst.command.remote.StereoOnWithCDCommand headfirst.command.remote.StereoOffCommand
[khe 4] headfirst.command.remote.NoCommand headfirst.command.remote.NoCommand
[khe 5] headfirst.command.remote.NoCommand headfirst.command.remote.NoCommand
[khe 6] headfirst.command.remote.NoCommand headfirst.command.remote.NoCommand

Đèn phòng khách đang bật
Đèn phòng khách đã tắt
Đèn bếp đang bật
Đèn bếp đã tắt
Quạt trần phòng khách ở mức cao
Quạt trần phòng khách đã tắt
Âm thanh nổi trong phòng khách đang bật
Hệ thống âm thanh nổi phòng khách được thiết lập cho đầu vào CD
Âm lượng âm thanh nổi phòng khách được đặt ở mức 11
Đài phát thanh phòng khách đã tắt

%
Trên khe cắm
Tắt khe cắm
←
Các lệnh của chúng tôi đang hoạt động! Hãy nhớ rằng,
đầu ra từ mỗi thiết bị đều đến từ các lớp nhà cung cấp.
Ví dụ, khi một vật thể phát sáng được bật, nó sẽ in ra
thông báo "Đèn phòng khách đang bật".
```

đối tượng rỗng



Bắt tốt. Chúng tôi đã lén lút thêm một chút gì đó vào đó. Trong điều khiển từ xa, chúng tôi không muốn kiểm tra xem lệnh có được tải mỗi khi chúng tôi tham chiếu đến một khe cắm hay không. Ví dụ, trong phương thức onButtonWasPushed(), chúng tôi sẽ cần mã như thế này:

```
công khai void onButtonWasPushed(int slot) {
    nếu (onCommands[slot] != null) {
        onCommands[khe].thực thi();
    }
}
```

Vậy, làm sao chúng ta giải quyết đư ợc vấn đề đó? Triển khai một lệnh không làm gì cả!

```
lớp công khai NoCommand thực hiện Command {
    công khai void thực hiện() { }
}
```

Sau đó, trong hàm tạo RemoteControl, theo mặc định, chúng ta gán cho mỗi ô một đối tượng NoCommand và chúng ta biết rằng chúng ta sẽ luôn có một số lệnh để gọi trong mỗi ô.

```
Lệnh noCommand = new NoCommand();
đối với (int i = 0; i < 7; i++) {
    onCommands[i] = noCommand;
    offCommands[i] = noCommand;
}
```

Vì vậy, trong kết quả chạy thử nghiệm của chúng tôi, bạn sẽ thấy các khe chưa đư ợc gán cho lệnh nào, ngoại trừ đối tượng NoCommand mặc định mà chúng tôi đã gán khi tạo RemoteControl.



Mẫu
đáng kính
Đề cập đến

Đối tượng NoCommand là một ví dụ về đối tượng null. Đối tượng null hữu ích khi bạn không có đối tượng có ý nghĩa để trả về, nhưng bạn muốn xóa trách nhiệm xử lý null khỏi máy khách. Ví dụ, trong điều khiển từ xa của chúng tôi, chúng tôi không có đối tượng có ý nghĩa để gán cho từng khe cắm ngay từ đầu, vì vậy chúng tôi đã cung cấp đối tượng NoCommand đóng vai trò là đối tượng thay thế và không làm gì khi phương thức thực thi của nó đư ợc gọi.

Bạn sẽ tìm thấy cách sử dụng Null Object kết hợp với nhiều Design Pattern và đôi khi bạn thậm chí còn thấy Null Object đư ợc liệt kê là một Design Pattern.

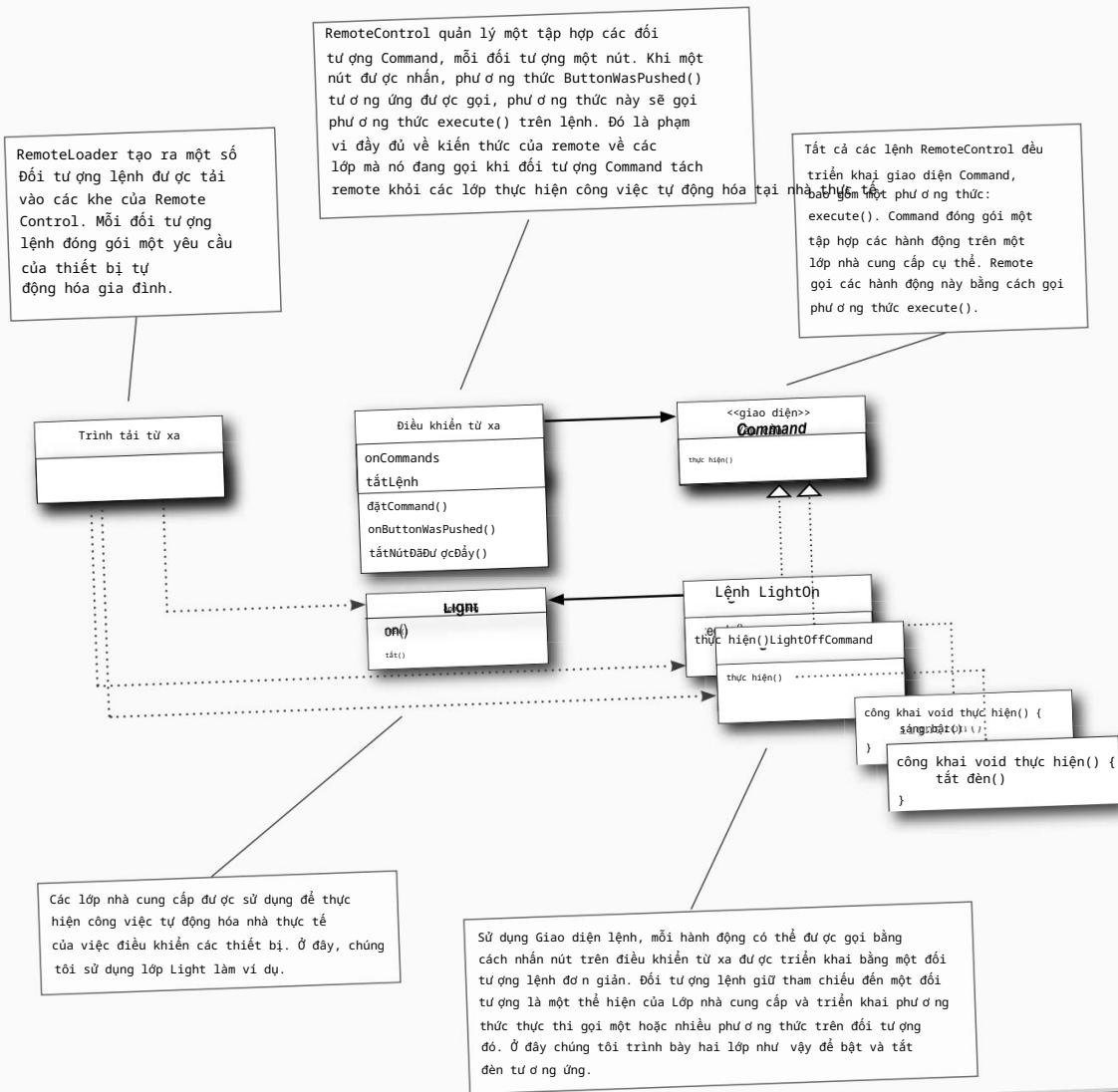
mẫu lệnh

Đã đến lúc viết tài liệu đó...

Thiết kế API điều khiển từ xa cho hệ thống tự động hóa gia đình hoặc Bust, Inc.,

Chúng tôi rất vui mừng được giới thiệu với bạn giao diện lập trình ứng dụng và thiết kế sau đây cho Điều khiển từ xa tự động hóa gia đình của bạn. Mục tiêu thiết kế chính của chúng tôi là giữ cho mã điều khiển từ xa đơn giản nhất có thể để không cần thay đổi khi các lớp nhà cung cấp mới được tạo ra. Để đạt được mục đích này, chúng tôi đã sử dụng Mẫu lệnh để tách hợp lý lớp RemoteControl khỏi các Lớp nhà cung cấp. Chúng tôi tin rằng điều này sẽ giảm chi phí sản xuất từ xa cũng như giảm đáng kể chi phí bảo trì liên tục của bạn.

Sơ đồ lớp sau đây cung cấp tổng quan về thiết kế của chúng tôi:



đừng quên hoàn tác



Chúng ta đang làm gì?

Đơn giản, chúng ta cần thêm chức năng để hỗ trợ nút hoàn tác trên điều khiển từ xa. Nó hoạt động như thế này: giả sử Đèn phòng khách tắt và bạn nhấn nút bật trên điều khiển từ xa. Rõ ràng là đèn sẽ bật. Bây giờ nếu bạn nhấn nút hoàn tác thì hành động cuối cùng sẽ bị đảo ngược - trong trường hợp này đèn sẽ tắt. Trước khi đi vào các ví dụ phức tạp hơn, hãy để đèn hoạt động với nút hoàn tác:

- Khi các lệnh hỗ trợ hoàn tác, chúng có phương thức `undo()` phản ánh phương thức `execute()` phương pháp. Đầu tiên `execute()` đã làm gì lần cuối, `undo()` sẽ đảo ngược. Vì vậy, trước khi chúng ta có thể thêm `undo` vào lệnh của mình, chúng ta cần thêm phương pháp `undo()` vào giao diện `Command`:

```
Giao diện công khai Lệnh {
    công khai void thực hiện();
    công khai void undo();
}
```

Đây là phương thức `undo()` mới.

Thật đơn giản.

Bây giờ, chúng ta hãy tìm hiểu sâu hơn về lệnh `Light` và triển khai phương thức `undo()`.

mẫu lệnh

- 2 Hãy bắt đầu với LightOnCommand: nếu phuong thuc execute() cua LightOnCommand duoc goi, thi phuong thuc on() duoc goi lan cuoi. Chung ta biет rang undo() can thuc hien nguyen lai bang cach goi phuong thuc off().

```

lớp công khai LightOnCommand thực hiện Command {
    Ánh sáng nhẹ;

    công khai LightOnCommand(Ánh sáng ánh sáng) {
        this.light = ánh sáng;
    }

    công khai void thực hiện() {
        sáng.bật();
    }

    công khai void undo() {
        tắt đèn();
    }
}

```

execute() sẽ bật đèn, do đó undo() chỉ đơn giản là tắt đèn trở lại.

Quá dễ! Bây giờ đến LightOffCommand. Ở đây phuong thuc undo() chỉ cần gọi phuong thuc on() của Light.

```

lớp công khai LightOffCommand thực hiện Command {
    Ánh sáng nhẹ;

    công khai LightOffCommand(Ánh sáng ánh sáng) {
        this.light = ánh sáng;
    }

    công khai void thực hiện() {
        tắt đèn();
    }

    công khai void undo() {
        sáng.bật();
    }
}

```

Và ở đây, undo() sẽ bật đèn trở lại!

Có cách nào dễ hơn không? Được rồi, chúng ta vẫn chưa xong; chúng ta cần thêm một chút hỗ trợ vào Điều khiển từ xa để xử lý việc theo dõi nút cuối cùng được nhấn và nút hoàn tác được nhấn.

thực hiện hoàn tác

- 3 Để thêm hỗ trợ cho nút hoàn tác, chúng ta chỉ cần thực hiện một vài thay đổi nhỏ đối với lớp Remote Control. Đây là cách chúng ta sẽ thực hiện: chúng ta sẽ thêm một biến thể hiện mới để theo dõi lệnh cuối cùng được gọi; sau đó, bất cứ khi nào nút hoàn tác được nhấn, chúng ta sẽ truy xuất lệnh đó và gọi phương thức undo() của nó.

```

lớp công khai RemoteControlWithUndo {
    Lệnh[] trên Lệnh;
    Lệnh[] tắtLệnh;
    Lệnh undoCommand; ← Đây là nơi chúng ta sẽ lưu trữ lệnh cuối cùng được thực thi cho nút hoàn tác.

    công khai RemoteControlWithUndo() {
        onCommands = Lệnh mới[7];
        offCommands = Lệnh mới[7];

        Lệnh noCommand = new NoCommand();
        đổi với (int i = 0; i < 7; i++) {
            onCommands[i] = noCommand;
            offCommands[i] = noCommand;
        }
        undoCommand = noCommand; ← Giống như các ô khác, nút hoàn tác bắt đầu bằng NoCommand, do đó, việc nhấn nút hoàn tác trước bất kỳ nút nào khác sẽ không có tác dụng gì cả.
    }

    public void setCommand(int slot, Lệnh onCommand, Lệnh offCommand) {
        onCommands[khe] = onCommand;
        offCommands[khe] = offCommand;
    }

    công khai void onButtonWasPushed(int slot) {
        onCommands[khe].thực thi(); ← Khi một nút được nhấn, chúng ta lấy lệnh và thực hiện lệnh đó trước; sau đó chúng ta lưu tham chiếu đến lệnh đó trong biến thể hiện undoCommand. Chúng ta thực hiện điều này cho cả lệnh "on" và lệnh "off".
        undoCommand = onCommands[khe];
    }

    công khai void offButtonWasPushed(int slot) {
        tắt Lệnh[khe].thực thi();
        undoCommand = offCommands[khe]; ←
    }

    công khai void undoButtonWasPushed() {
        undoCommand.undo(); ← Khi nút hoàn tác được nhấn, chúng ta sẽ gọi phương thức undo() của lệnh được lưu trữ trong undoCommand.
    }

    công khai String toString() {
        // mã toString ở đây...
    }
}

```

mẫu lệnh

Đã đến lúc kiểm tra chất lượng nút Hoàn tác!

Để được rồi, chúng ta hãy sửa lại dây thử một chút để thử nút hoàn tác:

```
lớp công khai RemoteLoader {

    public static void main(String[] args) {
        RemoteControlWithUndo remoteControl = new RemoteControlWithUndo();

        Ánh sáng livingRoomLight = new Light("Phòng khách");
        LightOnCommand livingRoomLightOn = new
            LightOnCommand(livingRoomLight);
        LightOffCommand livingRoomLightOff = new
            LightOffCommand(livingRoomLight);

        remoteControl.setCommand(0, Đèn phòng khách Bật, Đèn phòng khách Tắt);

        remoteControl.onButtonWasPushed(0);
        remoteControl.offButtonWasPushed(0);
        System.out.println(Điều khiển từ xa);
        remoteControl.undoButtonWasPushed();
        remoteControl.offButtonWasPushed(0);
        remoteControl.onButtonWasPushed(0);
        System.out.println(Điều khiển từ xa);
        remoteControl.undoButtonWasPushed();
    }
}
```

Và đây là kết quả thử nghiệm...

```
Cửa sổ chỉnh sửa tệp Trợ giúp UndoCommandsDefyEntropy
% java RemoteLoader
Đèn đang bật
Đèn đã tắt
----- Điều khiển từ xa -----
[khe 0] headfirst.command.undo.LightOnCommand headfirst.command.undo.LightOffCommand
[khe 1] headfirst.command.undo.NoCommand headfirst.command.undo.NoCommand
[khe 2] headfirst.command.undo.NoCommand headfirst.command.undo.NoCommand
[khe 3] headfirst.command.undo.NoCommand headfirst.command.undo.NoCommand
[khe 4] headfirst.command.undo.NoCommand headfirst.command.undo.NoCommand
[khe 5] headfirst.command.undo.NoCommand headfirst.command.undo.NoCommand
[khe 6] headfirst.command.undo.NoCommand headfirst.command.undo.NoCommand
[hoàn tác] headfirst.command.undo.LightOffCommand
----- Điều khiển từ xa -----
[khe 0] headfirst.command.undo.LightOnCommand headfirst.command.undo.LightOffCommand
[khe 1] headfirst.command.undo.NoCommand headfirst.command.undo.NoCommand
[khe 2] headfirst.command.undo.NoCommand headfirst.command.undo.NoCommand
[khe 3] headfirst.command.undo.NoCommand headfirst.command.undo.NoCommand
[khe 4] headfirst.command.undo.NoCommand headfirst.command.undo.NoCommand
[khe 5] headfirst.command.undo.NoCommand headfirst.command.undo.NoCommand
[khe 6] headfirst.command.undo.NoCommand headfirst.command.undo.NoCommand
[hoàn tác] headfirst.command.undo.LightOnCommand
----- Điều khiển từ xa -----
[khe 0] headfirst.command.undo.LightOnCommand headfirst.command.undo.LightOffCommand
[khe 1] headfirst.command.undo.NoCommand headfirst.command.undo.NoCommand
[khe 2] headfirst.command.undo.NoCommand headfirst.command.undo.NoCommand
[khe 3] headfirst.command.undo.NoCommand headfirst.command.undo.NoCommand
[khe 4] headfirst.command.undo.NoCommand headfirst.command.undo.NoCommand
[khe 5] headfirst.command.undo.NoCommand headfirst.command.undo.NoCommand
[khe 6] headfirst.command.undo.NoCommand headfirst.command.undo.NoCommand
[hoàn tác] headfirst.command.undo.LightOnCommand
```

chúng ta cần giữ lại một số trạng thái để hoàn tác

Sử dụng trạng thái để thực hiện Hoàn tác

Đương nhiên rồi, việc triển khai hoàn tác trên Light rất bổ ích như ng hơ i quá dễ. Thông thư ờng, chúng ta cần quản lý một chút trạng thái để triển khai hoàn tác. Hãy thử một cái gì đó thú vị hơn một chút, như CeilingFan từ các lớp nhà cung cấp. Quạt trần cho phép thiết lập một số tốc độ cùng với phư ơng pháp tắt.

Sau đây là mã nguồn của CeilingFan:

```

lớp công khai CeilingFan {
    công khai tĩnh cuối cùng int CAO = 3;
    công khai tĩnh cuối cùng int MEDIUM = 2;
    công khai tĩnh cuối cùng int LOW = 1;
    công khai tĩnh cuối cùng int OFF = 0;
    Vị trí chuỗi;
    tốc độ int;

    public CeilingFan(Chuỗi vị trí) {
        this.location = vị trí;
        tốc độ = TẮT;
    }

    công khai void cao() {
        tốc độ = CAO;
        // mă đăc đặt quạt ở mức cao
    }

    công khai void medium() {
        tốc độ = TRUNG BÌNH;
        // mă đăc đặt quạt ở mức trung bình
    }

    công khai void thấp() {
        tốc độ = THẤP;
        // mă đăc đặt quạt ở mức thấp
    }

    công khai void tắt() {
        tốc độ = TẮT;
        // mă đăc tắt quạt
    }

    công khai int getSpeed() {
        tốc độ trả về;
    }
}

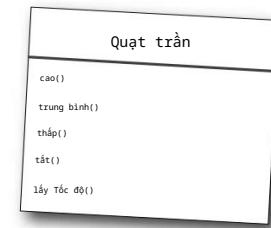
```

Lưu ý rằng lớp CeilingFan giữ trạng thái cục bộ biểu diễn tốc độ của quạt trần.

Ồ, vậy thì để thực hiện lệnh hoàn tác đúng cách, tôi phải tính đến tốc độ trước đó của quạt trần...

Những phư ơng pháp này thiết lập tốc độ của quạt trần.

Chúng ta có thể lấy được tốc độ hiện tại của quạt trần bằng cách sử dụng getSpeed().



mẫu lệnh

Thêm lệnh Hoàn tác vào quạt trần

Bây giờ chúng ta hãy giải quyết việc thêm undo vào các lệnh CeilingFan khác nhau. Để làm như vậy, chúng ta cần theo dõi cài đặt tốc độ cuối cùng của quạt và, nếu phuơng thức undo() được gọi, khôi phục quạt về cài đặt trước đó. Đây là mã cho CeilingFanHighCommand:

```

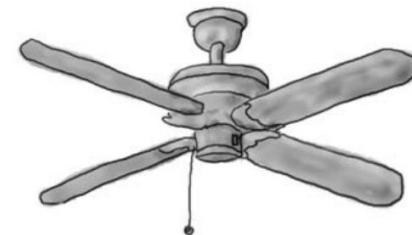
lớp công khai CeilingFanHighCommand thực hiện Command {
    Quạt trầnQuạt trần;
    int prevTốc độ;

    công khai CeilingFanHighCommand(CeilingFan ceilingFan) {
        this.ceilingFan = quạt trần;
    }

    công khai void thực hiện() {
        prevSpeed = ceilingFan.getSpeed();
        ceilingFan.caо();
    }

    công khai void undo() {
        nếu (prevSpeed == CeilingFan.HIGH) {
            ceilingFan.caо();
        } else if (prevSpeed == CeilingFan.MEDIUM) {
            ceilingFan.medium();
        } else if (prevSpeed == CeilingFan.LOW) {
            ceilingFan.thấp();
        } else if (prevSpeed == CeilingFan.OFF) {
            quạt trần.tắt();
        }
    }
}

```



Chúng tôi đã thêm trạng thái cục bộ để theo dõi tốc độ trước đó của quạt.

Trong quá trình thực hiện, trước khi thay đổi tốc độ của quạt, chúng ta cần ghi lại trạng thái trước đó của quạt, để phòng trường hợp chúng ta cần hoàn tác hành động của mình.

Để hoàn tác, chúng ta đặt lại tốc độ quạt về tốc độ trước đó.

não Apower

Chúng ta còn phải viết thêm ba lệnh quạt trần nữa: thấp, trung bình và tắt. Bạn có thể thấy cách thực hiện những lệnh này không?

kiểm tra quạt trần

Hãy chuẩn bị để thử nghiệm quạt trần

Đã đến lúc tải lệnh điều khiển quạt trần lên điều khiển từ xa.

Chúng ta sẽ tải nút bật khe số không với cài đặt trung bình cho quạt và khe số một với cài đặt cao. Cả hai nút tắt tương ứng sẽ giữ lệnh tắt quạt trần.

Đây là tập lệnh thử nghiệm của chúng tôi:

```
lớp công khai RemoteLoader {
```

```
    public static void main(String[] args) {
        RemoteControlWithUndo remoteControl = new RemoteControlWithUndo();
```

```
        CeilingFan ceilingFan = new CeilingFan("Phòng khách");
```

```
        CeilingFanMediumCommand ceilingFanMedium =
            lệnh CeilingFanMediumCommand(ceilingFan);
        CeilingFanHighCommand ceilingFanHigh =
            mới CeilingFanHighCommand(ceilingFan);
        Lệnh CeilingFanOff ceilingFanOff = lệnh mới
            CeilingFanOff(ceilingFan);
```

```
        remoteControl.setCommand(0, ceilingFanMedium, ceilingFanOff);
        remoteControl.setCommand(1, ceilingFanHigh, ceilingFanOff);
```

```
        remoteControl.onButtonWasPushed(0);
        remoteControl.offButtonWasPushed(0);
```

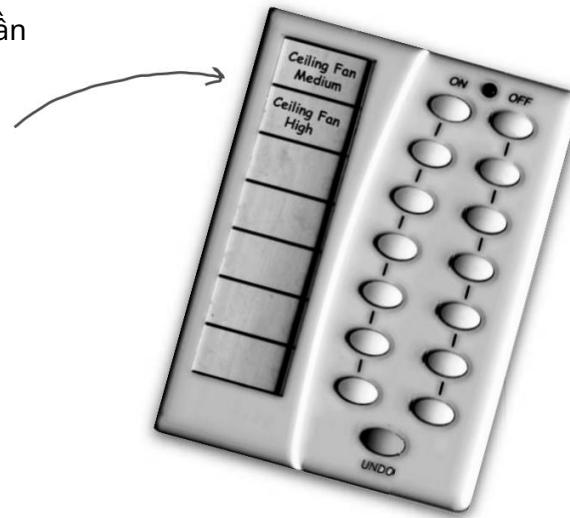
```
        System.out.println("Điều khiển từ xa");
```

```
        remoteControl.undoButtonWasPushed();
```

```
        remoteControl.onButtonWasPushed(1);
        System.out.println("Điều khiển từ xa");
        remoteControl.undoButtonWasPushed();
```

```
}
```

```
}
```



Ceiling Fan
Medium

Ceiling Fan
High

ON ● OFF

UNDO

Ở đây chúng ta khởi tạo ba lệnh: cao, trung bình và tắt.

Ở đây chúng ta đặt medium vào khe zero và high vào khe one.

Chúng ta cũng tải lệnh off.

Đầu tiên, bật quạt ở mức trung bình.

Sau đó tắt nó đi.

Hoàn tác! Nó sẽ trở lại mức trung bình...

Lần này hãy bật ở mức cao.

Và, hoàn tác thêm một lần nữa; nó sẽ trở lại mức trung bình.

mẫu lệnh

Kiểm tra quạt trần...

Đư ợc rồi, hãy bật điều khiển từ xa, nhập lệnh và nhấn một vài nút!

```
% java RemoteLoader

Quạt trần phòng khách ở mức trung bình
Quạt trần phòng khách đã tắt

----- Điều khiển từ xa -----
[khe 0] headfirst.command.undo.NoCommand [khe 1]
headfirst.command.undo.CeilingFanMediumCommand Lệnh

[slot 2] lệnh headfirst.command.undo.CeilingFanHighCommand
[slot 3] headfirst.command.undo.NoCommand [khe 4]
headfirst.command.undo.NoCommand [khe 5]
headfirst.command.undo.NoCommand [khe 6]
headfirst.command.undo.NoCommand [hoàn tác]
headfirst.command.undo.CeilingFanOffCommand

Quạt trần phòng khách ở mức trung bình
Quạt trần phòng khách ở mức cao

----- Điều khiển từ xa -----
[khe 0] headfirst.command.undo.NoCommand [khe 1]
headfirst.command.undo.CeilingFanMediumCommand Lệnh

[slot 2] lệnh headfirst.command.undo.CeilingFanHighCommand
[slot 3] headfirst.command.undo.NoCommand [khe 4]
headfirst.command.undo.NoCommand [khe 5]
headfirst.command.undo.NoCommand [khe 6]
headfirst.command.undo.NoCommand [hoàn tác]
headfirst.command.undo.CeilingFanHighCommand

Quạt trần phòng khách ở mức trung bình
%
```

Bạn có thể thấy rằng sau khi nhấn nút, lệnh sẽ được lưu vào khe nhớ. Khi bạn nhấn nút khác, lệnh cũ sẽ bị thay thế.

Sau đây là các lệnh trên điều khiển từ xa...

...và lệnh hoàn tác sẽ thực thi lệnh cuối cùng, CeilingFanOffCommand.

Bây giờ, hãy bật ở mức cao.

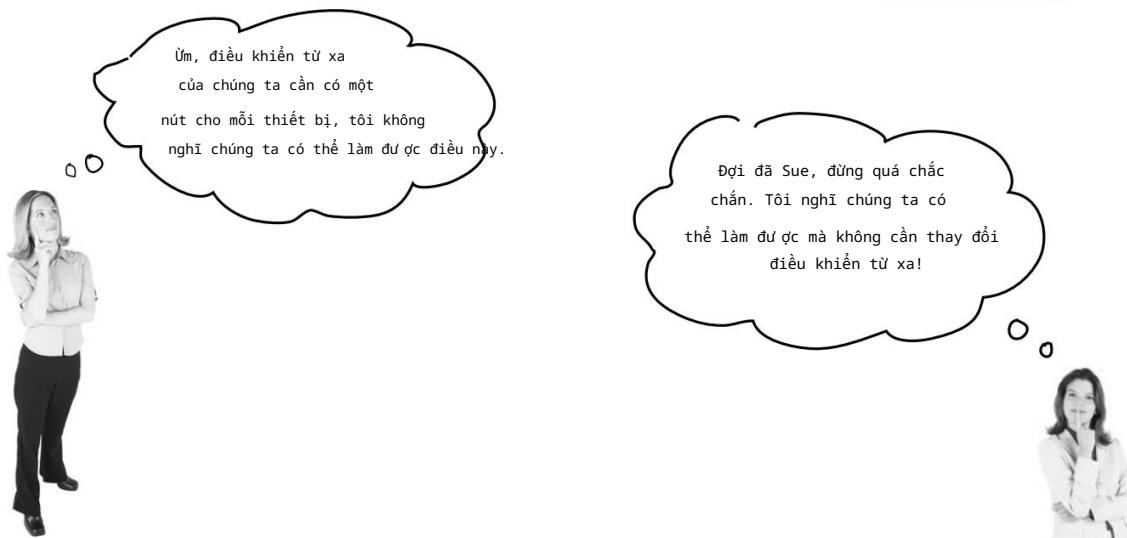
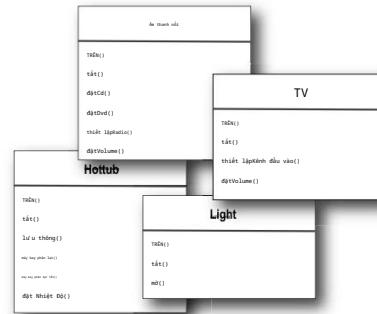
Bây giờ, high là lệnh cuối cùng được thực thi.

Nhấn thêm một lần nữa và quạt trần sẽ quay trở lại tốc độ trung bình.

lệnh macro

Mọi điều khiển từ xa đều cần có Chế độ tiệc tùng!

Có ích gì khi sử dụng điều khiển từ xa nếu bạn không thể nhấn một nút để làm mở đèn, bật dàn âm thanh và TV, chuyển sang chế độ DVD và bật bồn tắm nước nóng?



Ý tưởng của Mary là tạo ra một loại Lệnh mới có thể thực hiện các Lệnh khác... thậm chí nhiều hơn một Lệnh!
Ý tưởng khá hay phải không?

lớp công khai MacroCommand thực hiện Command {
 Lệnh [] lệnh;

```

    công khai MacroCommand(Command[] lệnh) {
        this.commands = lệnh;
    }

```

Lấy một mảng

Lệnh và lưu trữ chúng trong MacroCommand.

công khai void thực hiện() {

```

        đối với (int i = 0; i < lệnh.length; i++) {
            lệnh[i].thực thi();
        }
    }
}

```

Khi lệnh macro đư ợc thực thi từ xa, hãy thực thi từng lệnh một.

mẫu lệnh

Sử dụng lệnh macro

Chúng ta hãy cùng tìm hiểu cách sử dụng lệnh macro:

- 1 Đầu tiên chúng ta tạo tập lệnh mà chúng ta muốn đưa vào macro:

```
Ánh sáng ánh sáng = Ánh sáng mới ("Phòng khách");
TV tv = TV mới ("Phòng khách");
Âm thanh nồi âm thanh nồi = âm thanh nồi mới("Phòng khách");
Hottub hottub = new Hottub();

LightOnCommand lightOn = new LightOnCommand(ánh sáng);
StereoOnCommand stereoOn = new StereoOnCommand(stereo);
TVOnCommand tvOn = new TVOnCommand(tv);
HottubOnCommand hottubOn = HottubOnCommand mới(hottub);
```

Tạo tất cả các thiết bị, đèn, tivi, dàn âm thanh và bồn tắm nước nóng.

Bây giờ hãy tạo tất cả các lệnh On để điều khiển chúng.

 Chuốt bút chì của bạn Chúng ta cũng sẽ cần lệnh cho các nút tắt, viết mã để tạo ra chúng ở đây:

- 2 Tiếp theo, chúng ta tạo hai mảng, một cho các lệnh Bật và một cho các lệnh Tắt, và tải chúng bằng các lệnh tương ứng:

```
Lệnh[] partyOn = { lightOn, stereoOn, tvOn, hottubOn};
Lệnh[] partyOff = { lightOff, stereoOff, tvOff, hottubOff};
```

Tạo một mảng cho Bật và một mảng cho Tắt lệnh...

```
MacroCommand partyOnMacro = new MacroCommand(partyOn);
MacroCommand partyOffMacro = new MacroCommand(partyOff);
```

...và tạo hai macro và lệnh tương ứng để giữ chúng.

- 3 Sau đó chúng ta gán MacroCommand cho một nút như chúng ta vẫn thường làm:

```
remoteControl.setCommand(0, partyOnMacro, partyOffMacro);
```

Gán lệnh macro cho một nút giống như chúng ta gán bất kỳ lệnh nào khác.

Bài tập lệnh macro

4 Cuối cùng, chúng ta chỉ cần nhấn một vài nút và xem nó có hoạt động không.

```
System.out.println(Điều khiển từ xa);
System.out.println("--- Đang đẩy Macro vào---");
remoteControl.onButtonWasPushed(0);
System.out.println("--- Tắt Macro---");
remoteControl.offButtonWasPushed(0);
```



Đây là kết quả.

Cửa sổ chính sửa tệp Trợ giúp Bạn không thể đánh bại ABabka

```
% java RemoteLoader
----- Điều khiển từ xa -----
[khe 0] headfirst.command.party_MACRO [khe 1]
headfirst.command.party_NoCommand [khe 2]
headfirst.command.party_NoCommand [khe 3]
headfirst.command.party_NoCommand [khe 4]
headfirst.command.party_NoCommand [khe 5]
headfirst.command.party_NoCommand [khe 6]
headfirst.command.party_NoCommand [hoàn tác]
headfirst.command.party_NoCommand
```



Sau đây là hai lệnh macro.

```
headfirst.command.party_MACRO
headfirst.command.party_NoCommand
headfirst.command.party_NoCommand
headfirst.command.party_NoCommand
headfirst.command.party_NoCommand
headfirst.command.party_NoCommand
headfirst.command.party_NoCommand
```



Tất cả các lệnh trong macro đều
được thực thi khi chúng ta
gọi macro on...

```
--- Đẩy Macro lên---
Đèn đang bật
Âm thanh nổi trong phòng khách đang bật
TV phòng khách đang bật
Kênh truyền hình phòng khách được thiết lập cho DVD
Bồn tắm nưỚc nóng đang nóng đến mức bốc hơi 104 độ
Bồn tắm nưỚc nóng đang sủi bọt!
```



và khi chúng ta gọi macro
off. Có vẻ như nó hoạt động.

```
--- Tắt Macro---
Đèn đã tắt
Đài phát thanh phòng khách đã tắt
TV phòng khách đã tắt
Bồn tắm nưỚc nóng đang làm mát đến 98 độ
```

%



Bài tập

MacroCommand của chúng ta chỉ thiếu chức năng hoàn tác. Khi nút hoàn tác được nhấn sau lệnh macro, tất cả các lệnh được gọi trong macro phải hoàn tác các hành động trước đó của chúng. Đây là mã cho MacroCommand; hãy tiếp tục và triển khai phư ơng thức undo():

```

lớp công khai MacroCommand thực hiện Command {
    Lệnh [] lệnh;

    công khai MacroCommand(Command[] lệnh) {
        this.commands = lệnh;
    }

    công khai void thực hiện() {
        đối với (int i = 0; i < lệnh.length; i++) {
            lệnh[i].thực thi();
        }
    }

    công khai void undo() {
        // code here
    }
}

```

H: Tôi có phải lúc nào cũng cần máy thu không?
Tại sao đối tượng lệnh không thể triển khai chi tiết phư ơng thức execute()?

A: Nhìn chung, chúng tôi phản ánh cho "sự ngu ngốc" đối tượng lệnh chỉ gọi một hành động trên một bộ thu; tuy nhiên, có nhiều ví dụ về đối tượng lệnh "thông minh" thực hiện hầu hết, nếu không muốn nói là tất cả, logic cần thiết để thực hiện một yêu cầu. Chắc chắn bạn có thể làm điều này; chỉ cần lưu ý rằng bạn sẽ không còn có cùng mức độ tách rời giữa bộ gọi và bộ thu, cũng như bạn sẽ không thể tham số hóa lệnh của mình bằng bộ thu.

không có câu hỏi ngớ ngẩn nào

Q: Làm thế nào tôi có thể thực hiện một lịch sử của các thao tác hoàn tác? Nói cách khác, tôi muốn có thể nhấn nút hoàn tác nhiều lần.

A: Câu hỏi tuyệt vời! Nó khá thực sự dễ dàng; thay vì chỉ giữ một tham chiếu đến lệnh cuối cùng được thực thi, bạn giữ một ngăn xếp các lệnh trước đó. Sau đó, bắt cứ khi nào nhấn hoàn tác, trình gọi của bạn sẽ đẩy mục đầu tiên ra khỏi ngăn xếp và gọi phư ơng thức hoàn tác() của nó.

Q: Tôi có thể vừa thực hiện chế độ Party như một Lệnh bằng cách tạo một PartyCommand và đưa lệnh gọi để thực thi các Lệnh khác vào phư ơng thức execute() của PartyCommand?

A: Bạn có thể; tuy nhiên, bạn sẽ về cơ bản là "mã hóa cứng" chế độ nhóm vào PartyCommand. Tại sao phải mất công? Với MacroCommand, bạn có thể quyết định động các Command nào bạn muốn đưa vào PartyCommand, do đó bạn có thể linh hoạt hơn khi sử dụng MacroCommand. Nhìn chung, MacroCommand là giải pháp thanh lịch hơn và ít yêu cầu mã mới hơn.

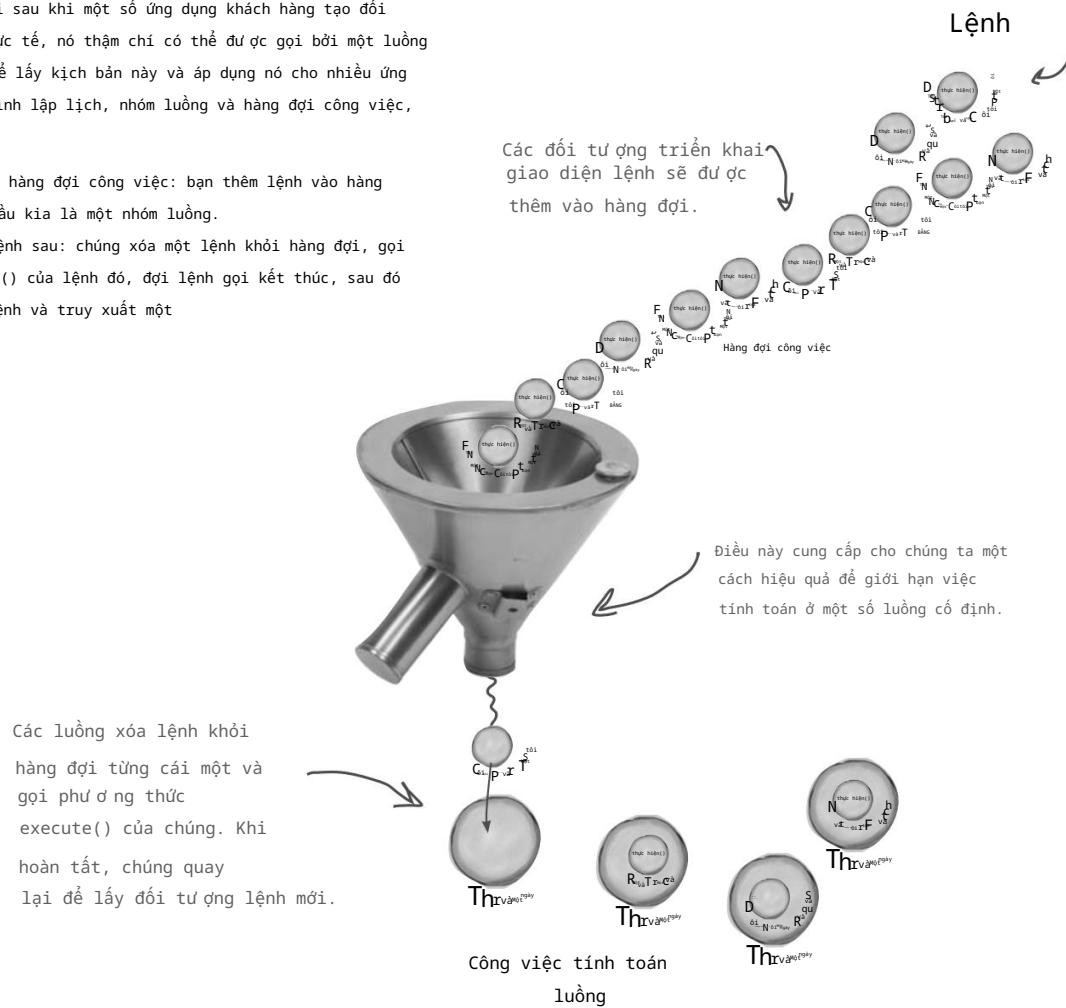
yêu cầu xếp hàng

Thêm các cách sử dụng của Command Pattern: xếp hàng yêu cầu

Lệnh cung cấp cho chúng ta một cách để đóng gói một phần tính toán (một bộ thu và một tập hợp các hành động) và truyền nó xung quanh như một đối tượng hạng nhất. Bây giờ, bản thân phép tính có thể được gọi sau khi một số ứng dụng khách hàng tạo đối tượng lệnh. Trên thực tế, nó thậm chí có thể được gọi bởi một luồng khác. Chúng ta có thể lấy kích bàn này và áp dụng nó cho nhiều ứng dụng hữu ích như trình lập lịch, nhóm luồng và hàng đợi công việc, để nêu tên một số.

Hãy tưởng tượng một hàng đợi công việc: bạn thêm lệnh vào hàng đợi ở một đầu và ở đầu kia là một nhóm luồng.

Các luồng chạy tập lệnh sau: chúng xóa một lệnh khỏi hàng đợi, gọi phương thức execute() của lệnh đó, đợi lệnh gọi kết thúc, sau đó loại bỏ đối tượng lệnh và truy xuất một cái mới.



Lưu ý rằng các lớp hàng đợi công việc được tách biệt hoàn toàn khỏi các đối tượng đang thực hiện tính toán. Một phút, một luồng có thể đang tính toán một phép tính tài chính, và phút tiếp theo, nó có thể đang truy xuất thứ gì đó từ mạng. Các đối tượng hàng đợi công việc không quan tâm; chúng chỉ truy xuất các lệnh và gọi execute(). Tương tự như vậy, miễn là bạn đưa các đối tượng vào hàng đợi triển khai Mẫu lệnh, phương thức execute() của bạn sẽ được gọi khi có luồng.

não Apower

Máy chủ web có thể sử dụng hàng đợi như thế nào? Bạn có thể nghĩ ra những ứng dụng nào khác?

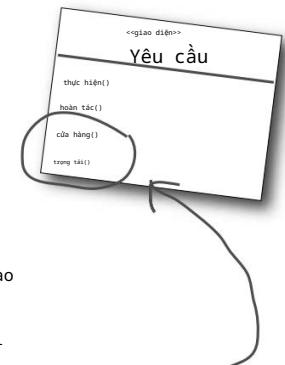
mẫu lệnh

Thêm nhiều cách sử dụng Mẫu lệnh: ghi nhật ký yêu cầu

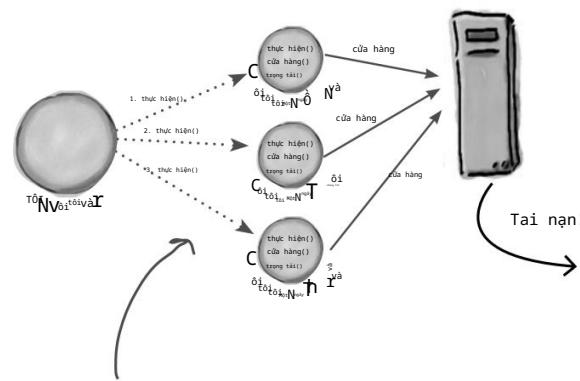
Ngữ nghĩa của một số ứng dụng yêu cầu chúng ta phải ghi lại tất cả các hành động và có thể khôi phục sau khi gặp sự cố bằng cách gọi lại các hành động đó. Command Pattern có thể hỗ trợ ngữ nghĩa này bằng cách thêm hai phương thức: `store()` và `load()`. Trong Java, chúng ta có thể sử dụng tuân tự hóa đối tượng để triển khai các phương thức này, nhưng các cảnh báo thông thường khi sử dụng tuân tự hóa để duy trì vẫn được áp dụng.

Quá trình này diễn ra như thế nào? Khi thực hiện lệnh, chúng ta sẽ lưu trữ lịch sử lệnh trên đĩa. Khi sự cố xảy ra, chúng ta tải lại các đối tượng lệnh và gọi phương thức `execute()` của chúng theo từng đợt và theo thứ tự.

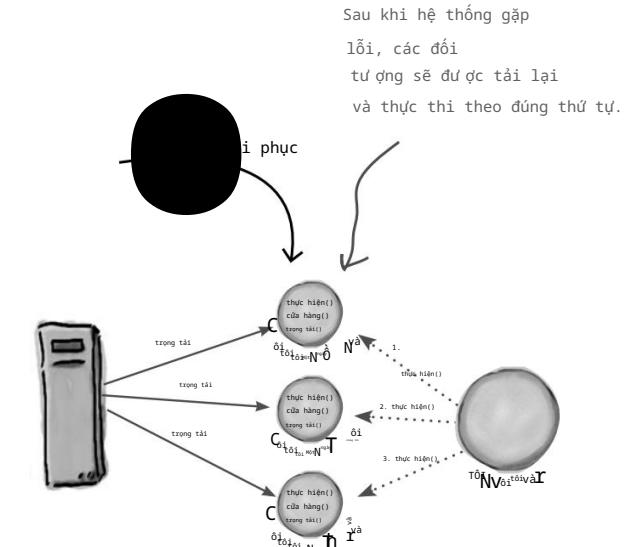
Bây giờ, loại ghi nhật ký này sẽ không có ý nghĩa đối với điều khiển từ xa; tuy nhiên, có nhiều ứng dụng gọi các hành động trên các cấu trúc dữ liệu lớn không thể lưu nhanh mỗi khi có thay đổi. Bằng cách sử dụng ghi nhật ký, chúng ta có thể lưu tất cả các hoạt động kể từ điểm kiểm tra cuối cùng và nếu có lỗi hệ thống, hãy áp dụng các hoạt động đó vào điểm kiểm tra của chúng ta. Lấy ví dụ, một ứng dụng bảng tính: chúng ta có thể muốn triển khai khôi phục lỗi của mình bằng cách ghi nhật ký các hành động trên bảng tính thay vì ghi một bản sao của bảng tính vào đĩa mỗi khi xảy ra thay đổi. Trong các ứng dụng nâng cao hơn, các kỹ thuật này có thể được mở rộng để áp dụng cho các tập hợp hoạt động theo cách giao dịch để tất cả các hoạt động hoàn tất hoặc không có hoạt động nào hoàn tất.



Chúng tôi thêm hai phương pháp để ghi nhật ký.

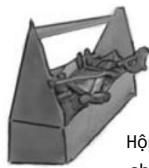


Khi mỗi lệnh được thực thi, nó sẽ được lưu trữ trên đĩa.



Sau khi hệ thống gặp lỗi, các đối tượng sẽ được tải lại và thực thi theo đúng thứ tự.

hộp công cụ thiết kế của bạn



Công cụ cho hộp công cụ thiết kế của bạn

Hộp công cụ của bạn bắt đầu trở nên nặng nề! Trong chương này, chúng tôi đã thêm một mẫu cho phép chúng tôi đóng gói các phương thức vào các đối tượng Command: lưu trữ chúng, truyền chúng xung quanh và gọi chúng khi bạn cần.



ĐIỂM ĐẦU TIÊN

β Mẫu lệnh

tách một đối tượng, đưa ra yêu cầu từ đối tượng biết cách thực hiện đối tượng đó.

ß Đối ứng Command nằm ở trung

tâm của sự tách biệt này và đóng gói một bộ thu với một hành động (hoặc một tập hợp các hành động).

ß Ngươi gọi đưa ra yêu cầu đổi tư ợng

Command bằng cách gọi phương thức execute() của đối tượng đó, phương thức này sẽ gọi các hành động đó trên người nhận.

Þ Invoker có thể được tham số hóa

bằng Lệnh, thậm chí là động khi chạy.

B Lệnh có thể hỗ trợ hoàn tác bằng

cách triển khai phu_σng
thức hoàn tác khôi phục đối tư σng
về trạng thái trư_σc đó trư_σc
khi phu_σng thức execute() đư_σc
gọi lần cuối.

β Macro Commands là một phần mở rộng
đơn giản của Command cho

phép gọi nhiều lệnh. Tuy nhiên
tự như vậy, Macro Commands có
thể dễ dàng hỗ trợ undo().

β Trong thực tế, không hiếm khi các

đối ứng Command "thông minh" tự thực hiện yêu cầu thay vì ủy quyền cho người nhận.

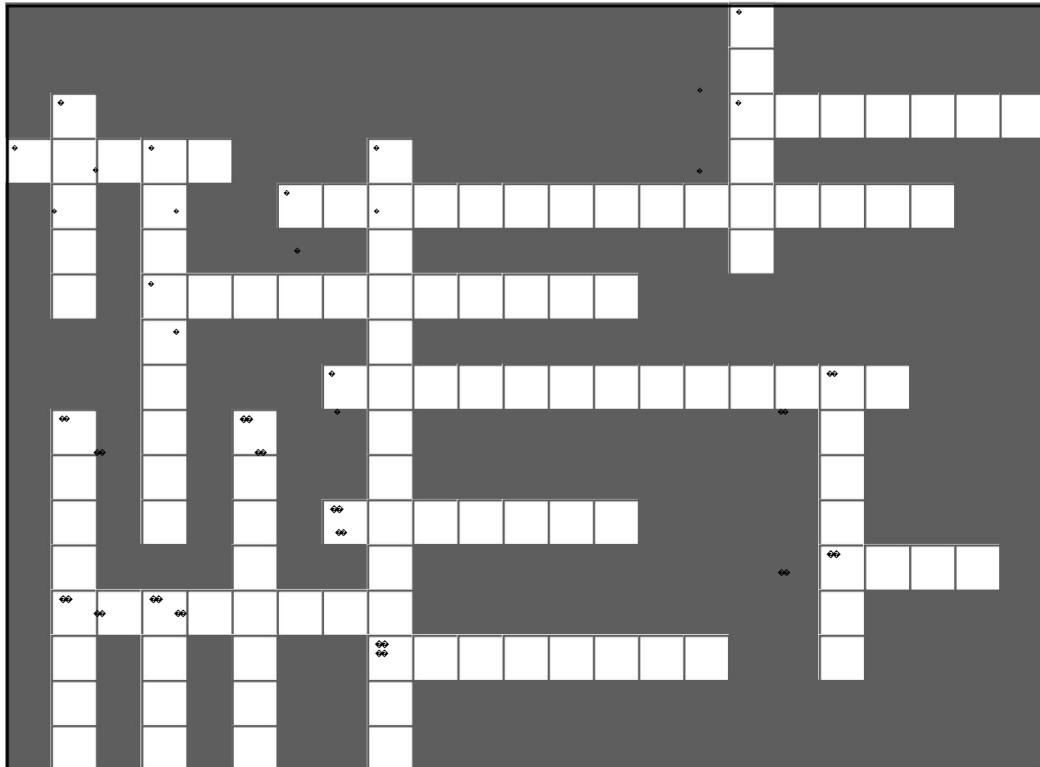
β Lệnh cũng có thể được sử dụng để triển khai hệ thống ghi nhật ký và giao dịch.

mẫu lệnh



Đã đến lúc hít thở sâu và để mọi chuyện lắng xuống

Đây là một trò chơi chữ khác; tất cả các từ giải đều nằm trong chuỗi này.

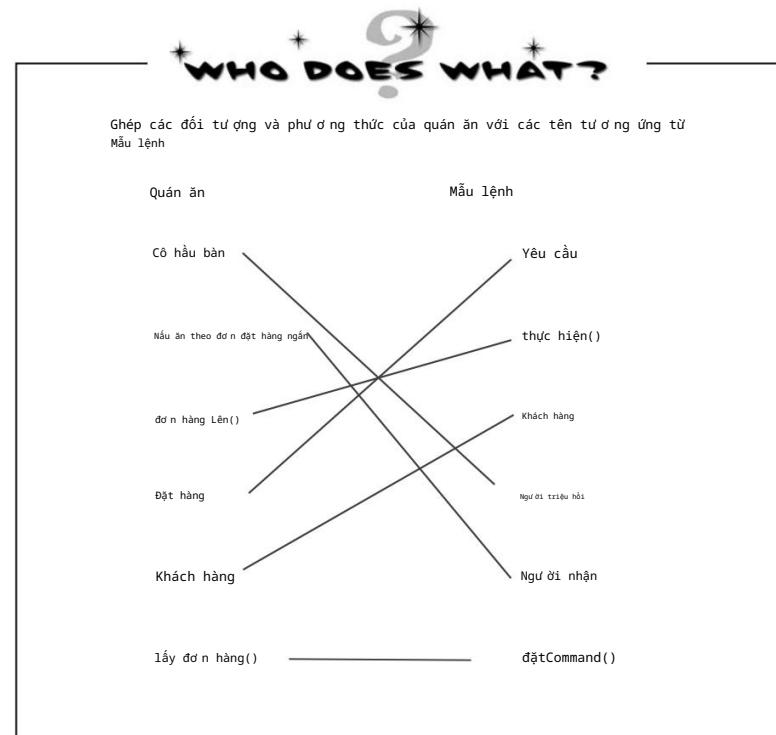


bạn đang ở đây 4 231

giải bài tập



Giải pháp bài tập



 Chuốt bút chì của bạn

```

lớp công khai GarageDoorOpenCommand thực hiện Command {
    GarageDoor garageDoor;
    GarageDoorOpenCommand công khai(GarageDoor garageDoor) {
        this.garageDoor = Cửa nhà để xe;
    }

    công khai void thực hiện()
    {
        garageDoor.up();
    }
}

```

Cửa sổ chính sửa tệp Trò chơi GreenEggs&Ham

```
%java Kiểm tra điều khiển từ xa
Đèn đang bật
Cửa nhà để xe đang mở
%
```

mẫu lệnh



Giải

pháp bài tập



Viết phương thức undo() cho MacroCommand

```

lớp công khai MacroCommand thực hiện Command {
    Lệnh [] lệnh;
    công khai MacroCommand(Command[] lệnh) {
        this.commands = lệnh;
    }

    công khai void thực hiện() {
        đối với (int i = 0; i < lệnh.length; i++) {
            lệnh[i].thực thi();
        }
    }

    công khai void undo() {
        đối với (int i = 0; i < lệnh.length; i++) {
            lệnh[i].hoàn tác();
        }
    }
}

```



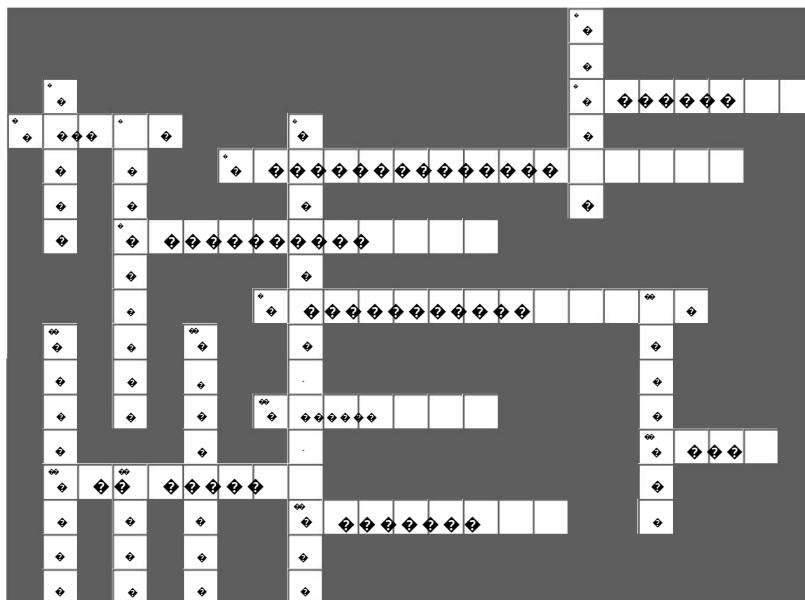
Chuôt bút chì Chúng ta cũng cần lệnh cho nút tắt.

Viết mã để tạo ra chúng ở đây:

```

LightOffCommand lightOff = new LightOffCommand(ánh sáng);
StereoOffCommand stereoOff = new StereoOffCommand(stereo);
Lệnh TVOff tvOff = lệnh TVOff mới(tv);
HottubOffCommand hottubOff = HottubOffCommand mới(hottub);

```



bạn đang ở đây 4 233

Tải xuống tại www.123ask.com

7 Các mẫu Adapter và Facade

g Hiện tại Thích nghi

h
g



Tương chư ởng này, chúng ta sẽ thử những kỹ tích bất khả thi như nhét một cái chốt vuông vào một lỗ tròn. Nghe có vẻ bất khả thi? Không phải khi chúng ta có các Mẫu thiết kế. Bạn còn nhớ Mẫu trang trí không? Chúng ta đã bọc các đối tư ợng để trao cho chúng những trách nhiệm mới. Bây giờ, chúng ta sẽ bọc một số đối tư ợng với mục đích khác: để làm cho giao diện của chúng trông giống như thứ mà chúng không phải. Tại sao chúng ta lại làm như vậy? Vì vậy, chúng ta có thể điều chỉnh một thiết kế mong đợi một giao diện cho một lớp triển khai một giao diện khác. Không chỉ vậy; khi chúng ta đang thực hiện, chúng ta sẽ xem xét một mẫu khác bọc các đối tư ợng để đơn giản hóa giao diện của chúng.

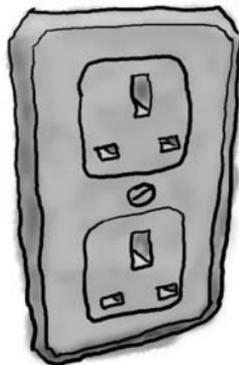
bộ chuyển đổi ở khắp mọi nơi

Bộ chuyển đổi xung quanh chúng ta

Bạn sẽ không gặp khó khăn khi hiểu bộ chuyển đổi 00 là gì vì thế giới thực có rất nhiều bộ chuyển đổi. Ví dụ thế này nhé: Bạn đã bao giờ cần sử dụng máy tính xách tay do Hoa Kỳ sản xuất ở một quốc gia châu Âu chưa? Vậy thì có lẽ bạn cần một bộ chuyển đổi nguồn AC...



Ổ cắm tư ờng Châu Âu



Bộ đổi nguồn AC



Phích cắm AC tiêu chuẩn



Máy tính xách tay của Hoa Kỳ
mong đợi một giao diện khác.

Ổ cắm điện kiểu Châu Âu chỉ
có một giao diện để lấy điện.

Bộ điều hợp chuyển đổi một giao
diện này sang giao diện khác.



Bạn biết bộ chuyển đổi có chức năng gì: nó nằm giữa phích cắm của máy tính xách tay và ổ cắm AC Châu Âu; công việc của nó là điều chỉnh ổ cắm Châu Âu để bạn có thể cắm máy tính xách tay vào và nhận điện. Hoặc hãy xem theo cách này: bộ chuyển đổi thay đổi giao diện của ổ cắm thành giao diện mà máy tính xách tay của bạn mong đợi.

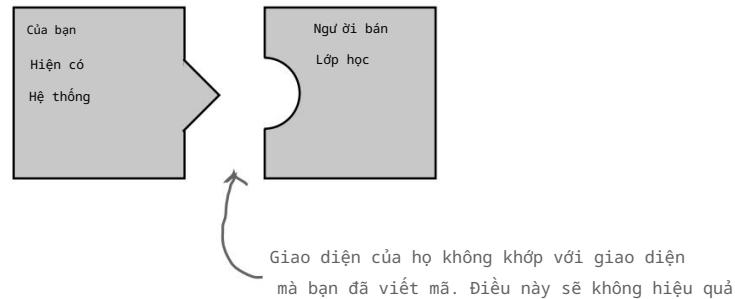
Bạn có thể nghĩ ra bao nhiêu bộ điều hợp thực tế khác?

Một số bộ đổi nguồn AC khá đơn giản - chúng chỉ thay đổi hình dạng của ổ cắm sao cho phù hợp với phích cắm và truyền dòng điện AC thẳng qua - nhưng một số bộ đổi nguồn khác phức tạp hơn ở bên trong và có thể tăng hoặc giảm công suất để phù hợp với nhu cầu của thiết bị.

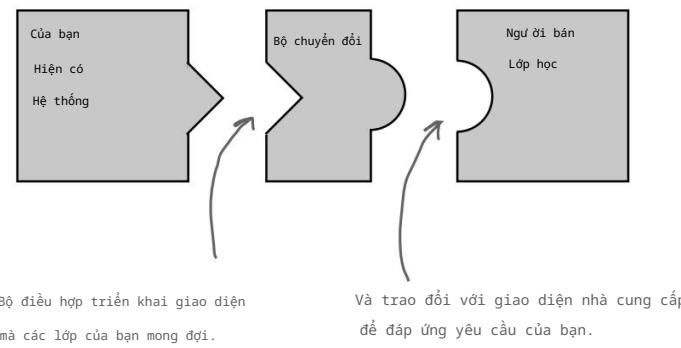
Đứa trẻ, đó là thế giới thực, còn bộ điều hợp hư hỏng thì sao? Vâng, bộ điều hợp 00 của chúng tôi đóng vai trò giống như các đối tác trong thế giới thực của chúng: chúng lấy một giao diện và điều chỉnh nó thành giao diện mà khách hàng mong đợi.

Bộ điều hợp hư ớng đổi tư ợng

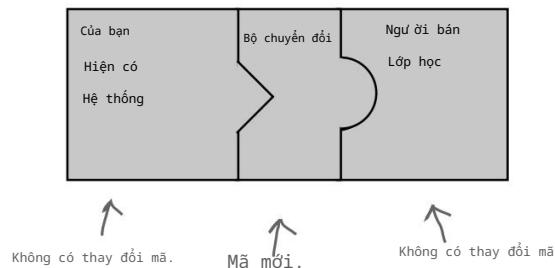
Giả sử bạn có một hệ thống phần mềm hiện có mà bạn cần đưa thư viện lớp của nhà cung cấp mới vào, như ng nhà cung cấp mới này lại thiết kế giao diện của họ khác với nhà cung cấp trước:



Đương rồi, bạn không muốn giải quyết vấn đề bằng cách thay đổi mã hiện tại của mình (và bạn không thể thay đổi mã của nhà cung cấp). Vậy bạn phải làm gì? Vâng, bạn có thể viết một lớp điều chỉnh giao diện nhà cung cấp mới thành giao diện bạn mong đợi.



Bộ điều hợp đóng vai trò trung gian bằng cách tiếp nhận các yêu cầu từ máy khách và chuyển đổi chúng thành các yêu cầu có ý nghĩa trên các lớp nhà cung cấp.



Bạn có thể nghĩ ra giải pháp nào không yêu cầu BẢN phải viết BẤT KỲ mã bổ sung nào để tích hợp các lớp nhà cung cấp mới không? Làm sao để nhà cung cấp cung cấp lớp bộ điều hợp.

bộ chuyển đổi gà tây

Nếu nó đi như một con vịt và kêu như một con vịt,
 thì nó có thể là một con gà tây vịt đư ợc quần bồng
 một bộ phận nối với con vịt...

Đã đến lúc xem bộ điều hợp hoạt động. Bạn còn nhớ những chú vịt của chúng ta trong Chương 1 không? Hãy cùng xem lại phiên bản đơn giản hơn một chút của các giao diện và lớp Duck:



```
giao diện công cộng Duck {  
    công khai void quack();  
    công khai void fly();  
}
```

Lần này, chú vịt của chúng ta triển khai giao diện Duck cho phép chú vịt có thể kêu và bay.

Đây là một phần lớp của Duck, MallardDuck.

```
lớp công khai MallardDuck thực hiện Duck {  
    công khai void quack() {  
        System.out.println("Quack");  
    }  
  
    công khai void fly() {  
        System.out.println("Tôi đang bay");  
    }  
}
```

Cách triển khai đơn giản: con vịt chỉ in ra những gì nó đang làm.

Bây giờ đã đến lúc gặp gỡ loài gia cầm mới nhất:

```
giao diện công cộng Thổ Nhĩ Kỳ {  
    công khai void gobble();  
    công khai void fly();  
}
```

Gà tây không kêu quác, mà chỉ kêu ngẫu nhiên.

Gà tây có thể bay, mặc dù chúng chỉ có thể bay những khoảng cách ngắn.

```

lớp công khai WildTurkey triển khai Turkey {
    công khai void gobble() {
        System.out.println("Gà thét");
    }

    công khai void fly() {
        System.out.println("Tôi đang bay một khoảng cách ngắn");
    }
}

```

Đây là một ví dụ cụ thể về Turkey; giống như Duck, nó chỉ in ra các hành động của mình.

Bây giờ, giả sử bạn thiếu các đối tượng Duck và bạn muốn sử dụng một số đối tượng Turkey thay thế. Rõ ràng là chúng ta không thể sử dụng turkeys ngay vì chúng có giao diện khác.

Vậy, chúng ta hãy viết một Bộ điều hợp:



Mã Gắn

```

lớp công khai TurkeyAdapter triển khai Duck {
    gà tây gà tây;

    public TurkeyAdapter(Thỏ Nhí Kỳ Thỏ Nhí Kỳ) {
        this.turkey = gà tây;
    }

    công khai void quack() {
        gà tây.ngó nghiến();
    }

    công khai void fly() {
        đối với (int i = 0; i < 5; i++) {
            gà tây.ruồi();
        }
    }
}

```

Đầu tiên, bạn cần triển khai giao diện của loại bạn đang điều chỉnh. Đây là giao diện mà khách hàng của bạn mong muốn thấy.

Tiếp theo, chúng ta cần có tham chiếu đến đối tượng mà chúng ta đang điều chỉnh; ở đây chúng ta thực hiện điều đó thông qua hàm tạo.

Bây giờ chúng ta cần triển khai tất cả các phương thức trong giao diện; việc chuyển đổi quack() giữa các lớp rất dễ dàng: chỉ cần gọi phương thức gobble().

Mặc dù cả hai giao diện đều có phương thức fly(), nhưng gà tây bay theo từng đợt ngắn - chúng không thể bay đường dài như vịt. Để ánh xạ giữa phương thức fly() của vịt và gà tây, chúng ta cần gọi phương thức fly() của gà tây năm lần để bù lại.

kiểm tra bộ chuyển đổi

Kiểm tra ô đĩa bộ chuyển đổi

Bây giờ chúng ta chỉ cần một số mã để kiểm tra bộ điều hợp của mình:

```

lớp công khai DuckTestDrive {
    public static void main(String[] args) {
        Vịt Mallard vịt = new MallardDuck();

        Gà tây WildTurkey = WildTurkey mới();
        Duck turkeyAdapter = new TurkeyAdapter(turkey);

        System.out.println("Con gà tây nói rằng...");
        gà tây.ngầu nghiến();
        gà tây.ruồi();

        System.out.println("\nCon vịt nói...");
        testDuck(vịt);

        System.out.println("\nTurkishAdapter cho biết...");
        testDuck(turkeyAdapter);
    }

    void tĩnh testDuck(Vịt vịt) {
        vịt.quạc();
        vịt.ruồi();
    }
}

```

Hãy cùng tạo ra một chú vịt nhé...
 và một con gà tây.
 Sau đó bọc con gà tây trong
 TurkeyAdapter, khiến nó
 trông giống như một con vịt.
 Sau đó, chúng ta hãy thử nghiệm con gà
 tây: làm cho nó kêu, làm cho nó bay.
 Bây giờ chúng ta hãy thử con vịt
 bằng cách gọi phương thức
 testDuck(), phương thức này
 mong đợi một đối tượng Duck.
 Đây là phương
 thức testDuck() của chúng tôi; nó
 lấy một con vịt và gọi các phương thức quack()
 và fly() của nó.

Chạy thử

Cửa sổ chỉnh sửa tệp Tạo giúp Don'tForgetToDoDuck

```
%java Kiểm tra điều khiển từ xa
Người Thổ Nhĩ Kỳ nói rằng...
Ồ ồ ồ
Tôi đang bay một khoảng cách ngắn
Vịt nói...
Lang băm
Tôi đang bay

TurkeyAdapter cho biết...
Ồ ồ ồ
Tôi đang bay một khoảng cách ngắn
```

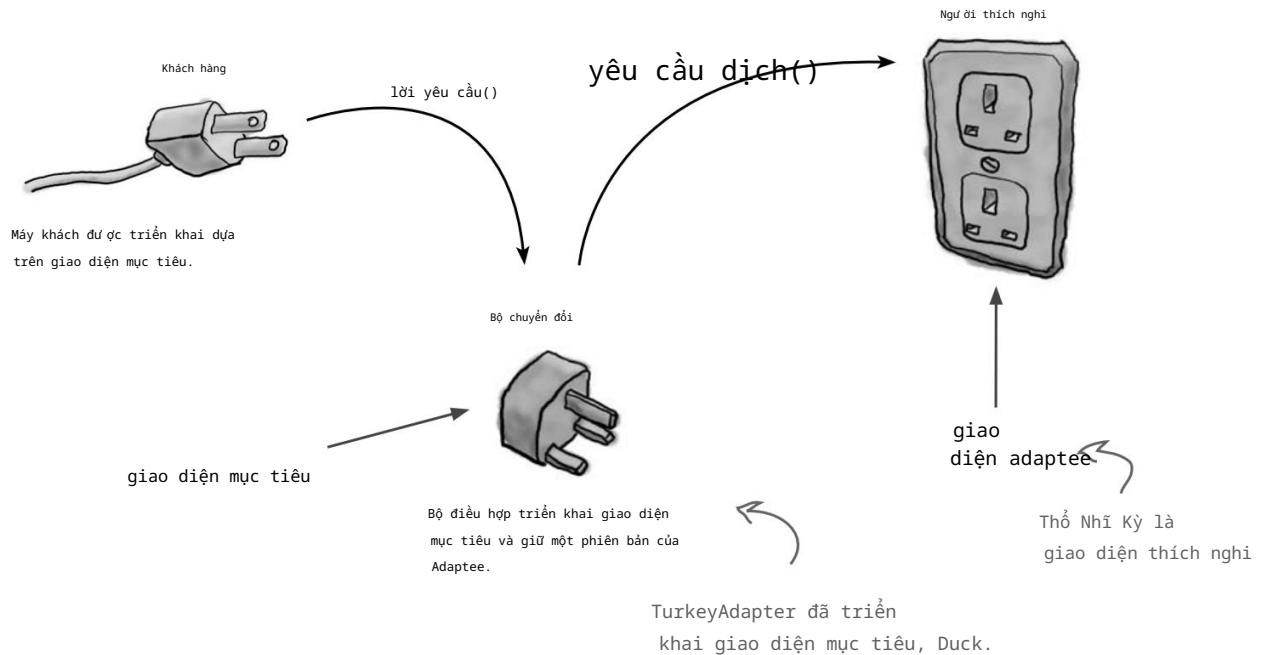
Con gà tây nuốt chửng
 và bay một quãng ngắn.

Vịt kêu và bay giống
 như bạn mong đợi.

Và bộ điều hợp sẽ ngẫu nhiên khi
 gọi quack() và bay một vài lần khi
 gọi fly(). Phương thức testDuck()
 không giờ biết rằng nó có một con
 gà tây cải trang thành một con vịt!

Giải thích về mẫu Adapter

Bây giờ chúng ta đã hiểu Adapter là gì, hãy cùng xem lại tất cả các thành phần một lần nữa.



Sau đây là cách Client sử dụng Adapter

- 1 Máy khách gửi yêu cầu đến bộ điều hợp bằng cách gọi phương thức trên đó sử dụng giao diện đích.
- 2 Bộ điều hợp dịch yêu cầu thành một hoặc nhiều cuộc gọi tới bộ điều hợp để có thể sử dụng giao diện bộ điều hợp.
- 3 Khách hàng nhận được kết quả cuộc gọi và không bao giờ biết có bộ điều hợp đang thực hiện việc dịch thuật.

Lưu ý rằng Khách hàng và Người dùng thích nghi đều có thể tiếp là hai bên tách biệt - không bên nào biết về bên kia.

mẫu bộ điều hợp đư ợc xác định



Chuốt bút chì của bạn

Giả sử chúng ta cũng cần một Bộ chuyển đổi có thể chuyển đổi Việt thành Gà tây.
Hãy gọi nó là DuckAdapter. Viết lớp đó:

Bạn đã xử lý phư ơng pháp bay như thế nào (sau cùng chúng ta đều biết vịt bay lâu hơn gà tây)? Kiểm tra câu trả lời ở cuối chương để biết giải pháp của chúng tôi. Bạn có nghĩ ra cách nào tốt hơn không?

không có Những câu hỏi ngắn

Q: "Thích nghi" đến mức nào?
cần phải làm gì với bộ điều hợp? Có vẻ
như nếu tôi cần triển khai một giao
diện mục tiêu lớn, tôi sẽ phải làm rất
NHIỀU việc.

A: Bạn chắc chắn có thể. Công việc
việc triển khai bộ điều hợp thực sự tì
lệ thuận với kích thước của giao diện mà bạn
cần hỗ trợ như giao diện mục tiêu của mình.
Tuy nhiên, hãy cân nhắc các lựa chọn của
bạn. Bạn có thể làm lại tất cả các lệnh gọi
phía máy khách đến giao diện, điều này sẽ
đến nhiều công việc điều tra và thay đổi mã.
Hoặc bạn có thể cung cấp một lớp gọn gàng có
thể đóng gói tất cả các thay đổi vào trong một lớp.

Q: Bộ chuyển đổi luôn luôn bao bọc một
và chỉ có một lớp?

A: Vai trò của Mô hình Bộ điều hợp là
chuyển đổi một giao diện này sang giao diện khác.
Trong khi hầu hết các ví dụ về mẫu bộ điều hợp đều cho
thấy một bộ điều hợp bao bọc một bộ điều hợp, chúng
ta đều biết rằng thế giới thường lộn xộn hơn một chút.
Vì vậy, bạn có thể gặp phải tình huống mà
một bộ điều hợp chứa hai hoặc nhiều bộ điều
hợp cần thiết để triển khai giao diện mục tiêu.
Điều này liên quan đến một mẫu khác gọi là
Mẫu Facade; mọi người thường nhầm lẫn giữa
hai mẫu này. Nhắc chúng ta xem lại điểm này khi
chúng ta nói về mặt tiền ở phần sau của chương này.

H: Nếu tôi có cả bộ phận cũ và mới thì sao?
trong hệ thống của tôi, các phần cũ mong
đợi giao diện nhà cung cấp cũ, nhưng
chúng tôi đã viết các phần mới để sử
dụng giao diện nhà cung cấp mới? Sẽ
rất khó hiểu khi sử dụng bộ điều hợp ở đây
và giao diện chưa đư ợc đóng gói ở đó. Tôi
không phải sẽ tốt hơn nếu chỉ viết mã cũ
của mình và quên bộ điều hợp sao?

A: Không nhất thiết. Một điều bạn
có thể làm là tạo một Bộ điều hợp hai chiều hỗ
trợ cả hai giao diện. Để tạo một Bộ điều hợp hai
chiều, chỉ cần triển khai cả hai giao diện liên
quan, do đó bộ điều hợp có thể hoạt động như
một giao diện cũ hoặc một giao diện mới.

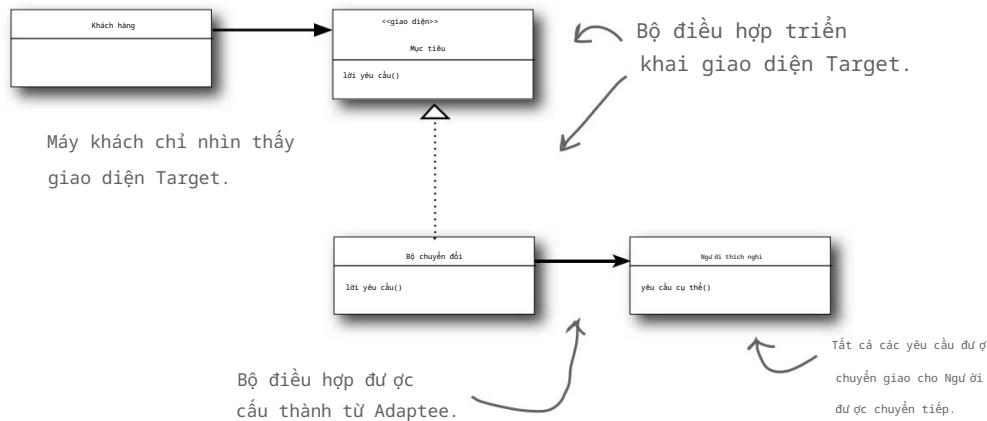
Mẫu bộ điều hợp đư ợc xác định

Đủ về vẹt, gà tây và bộ đổi nguồn AC; chúng ta hãy thực tế và xem định nghĩa chính thức của Mẫu bộ đổi nguồn:

Mẫu Adapter chuyển đổi giao diện của một lớp thành một giao diện khác mà khách hàng mong đợi. Adapter cho phép các lớp làm việc cùng nhau mà nếu không thì không thể làm việc cùng nhau do các giao diện không tương thích.

Bây giờ, chúng ta biết rằng mẫu này cho phép chúng ta sử dụng máy khách có giao diện không tương thích bằng cách tạo Bộ điều hợp thực hiện chuyển đổi. Điều này có tác dụng tách máy khách khỏi giao diện đã triển khai và nếu chúng ta mong đợi giao diện thay đổi theo thời gian, bộ điều hợp sẽ đóng gói thay đổi đó để máy khách không phải sửa đổi mỗi lần cần hoạt động với giao diện khác.

Chúng ta đã xem xét hành vi thời gian chạy của mẫu; hãy cùng xem sơ đồ lớp của nó:



Mẫu Adapter có đầy đủ các nguyên tắc thiết kế OOD tốt: hãy xem xét việc sử dụng thành phần đổi tương ứng để bọc adaptee bằng một giao diện đã thay đổi. Cách tiếp cận này có thêm lợi thế là chúng ta có thể sử dụng một bộ điều hợp với bất kỳ lớp con nào của adaptee.

Ngoài ra, hãy kiểm tra cách mẫu liên kết máy khách với giao diện, không phải triển khai; chúng ta có thể sử dụng một số bộ điều hợp, mỗi bộ chuyển đổi một tập hợp lớp phụ trợ khác nhau. Hoặc, chúng ta có thể thêm các triển khai mới sau đó, miễn là chúng tuân thủ giao diện Target.

bộ điều hợp đối tư ợng và lớp

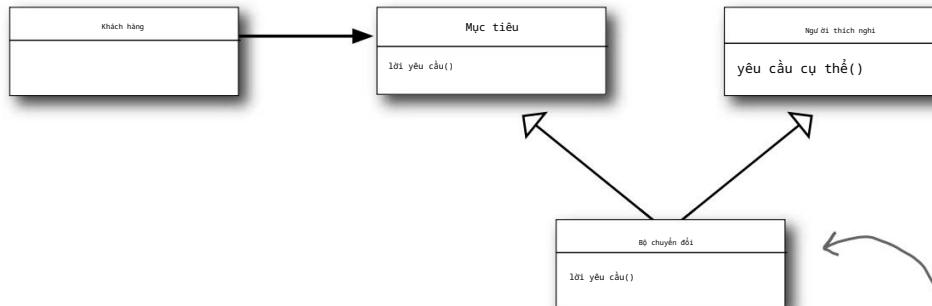
Bộ điều hợp đối tư ợng và lớp

Mặc dù đã xác định được mô hình, chúng tôi vẫn chưa kể cho bạn toàn bộ câu chuyện.

Trên thực tế có hai loại bộ điều hợp: bộ điều hợp đối tư ợng và bộ điều hợp lớp.

Chương này đã đề cập đến bộ điều hợp đối tư ợng và sơ đồ lớp ở trang trước là sơ đồ của bộ điều hợp đối tư ợng.

Vậy class adapter là gì và tại sao chúng tôi chưa nói với bạn về nó? Bởi vì bạn cần đa kế thừa để triển khai nó, điều này không thể thực hiện được trong Java. Nhưng điều đó không có nghĩa là bạn có thể không gặp phải nhu cầu về class adapter trong tư duy lai khi sử dụng ngôn ngữ đa kế thừa yêu thích của mình! Hãy cùng xem sơ đồ lớp cho đa kế thừa.



Thay vì sử dụng com-position để điều chỉnh Adaptee, giờ đây Adapter sẽ phân lớp các lớp Adaptee và Target.

Bạn có thấy quen không? Đúng vậy - điểm khác biệt duy nhất là với bộ điều hợp lớp, chúng ta phân lớp Target và Adaptee, trong khi với bộ điều hợp đối tư ợng, chúng ta sử dụng thành phần để chuyển yêu cầu đến Adaptee.

não Một sức mạnh

Bộ điều hợp đối tư ợng và bộ điều hợp lớp sử dụng hai phương tiện khác nhau để điều chỉnh đối tư ợng được điều chỉnh (thành phần so với kế thừa). Những khác biệt về triển khai này ảnh hưởng đến tính linh hoạt của bộ điều hợp như thế nào?

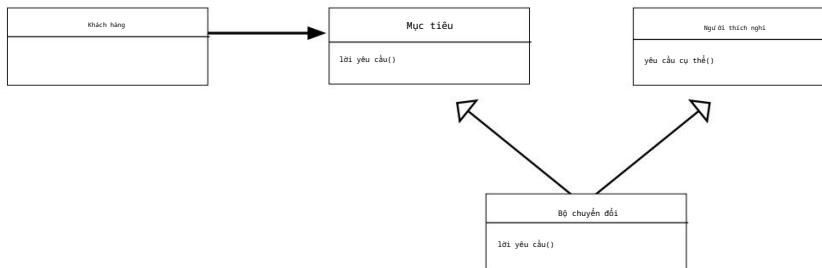
mẫu bộ chuyển đổi



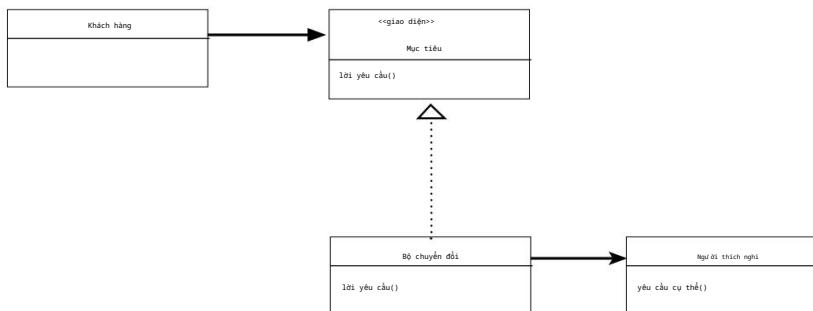
Nam châm vịt

Nhiệm vụ của bạn là lấy nam châm hình con vịt và con gà tây và kéo chúng qua phần sơ đồ mô tả vai trò của loài chim đó trong ví dụ trước đó. (Cố gắng không lật lại các trang.) Sau đó, thêm chú thích của riêng bạn để mô tả cách thức hoạt động.

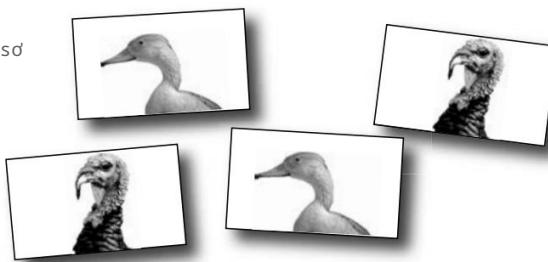
Bộ điều hợp lớp



Bộ điều hợp đối tư ứng



Kéo những phần này vào sơ đồ lớp để hiển thị phần nào của sơ đồ đại diện cho Vịt và phần nào đại diện cho Gà tây.



đáp án bài tập



Nam châm vịt Trả lời

Lưu ý: bộ điều hợp lớp sử dụng đa

ké thừa, do đó bạn không thể thực hiện điều đó trong Java...

Bộ điều hợp lớp

Khách hàng nghĩ rằng anh ta đang nói chuyện với một con vịt.

Mục tiêu là lớp Duck. Đây là lớp mà máy khách gọi các phuơng thức.

Lớp vịt



lời yêu cầu()

Lớp gà tây



yêu cầu cụ thể()

Adapter
request()

Lớp Turkey không có cùng phuơng thức như Duck, nhưng Adapter có thể tiếp nhận các lệnh gọi phuơng thức của Duck và quay lại để gọi các phuơng thức trên Turkey.

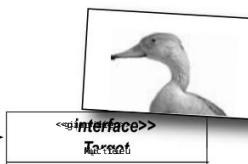
Bộ điều hợp cho phép Gà tây phản hồi các yêu cầu trên Vịt bằng cách mở rộng CẢ HAI lớp (Vịt và Gà tây).

Giao diện con vịt.

Bộ điều hợp đối tượng

Khách hàng nghĩ rằng anh ta đang nói chuyện với một con vịt.

Cũng giống như với Class Adapter, Target là lớp Duck. Đây là những gì mà client gọi các phuơng thức trên.



lời yêu cầu()

Bộ chuyển đổi
lời yêu cầu()

Lớp Turkey không có cùng giao diện với Duck. Nói cách khác, Turkey không có phuơng thức quack(), v.v.

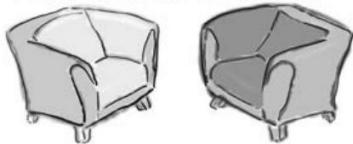


Thỏ Nhí
Kỹ phản đối.

Bộ điều hợp triển khai giao diện Duck, nhưng khi nhận được lệnh gọi phuơng thức, nó sẽ chuyển lệnh gọi đó cho Turkey.

Nhờ có Adapter, Turkey (Adaptee) sẽ nhận được các cuộc gọi mà máy khách thực hiện trên giao diện Duck.

Fireside Chats



Bài nói chuyện tối nay: Bộ điều hợp đối tư ợng và Bộ điều hợp lớp gặp nhau trực tiếp.

Bộ điều hợp đối tư ợng

Vì tôi sử dụng thành phần nên tôi có lợi thế hơn. Tôi không chỉ có thể điều chỉnh một lớp được điều chỉnh mà còn bất kỳ lớp con nào của nó.

Bộ điều hợp lớp

Đúng vậy, tôi gặp rắc rối với điều đó vì tôi cam kết với một lớp adaptee cụ thể, nhưng tôi có một lợi thế lớn vì tôi không phải triển khai lại toàn bộ adaptee của mình. Tôi cũng có thể ghi đè hành vi của adaptee của mình nếu cần vì tôi chỉ đang phân lớp.

Ở nơi tôi sống, chúng tôi thích sử dụng thành phần hơn là kế thừa; bạn có thể tiết kiệm được một vài dòng mã, nhưng tất cả những gì tôi làm là viết một ít mã để chuyển giao cho bên được điều chỉnh.

Chúng tôi muốn mọi thứ được linh hoạt.

Linh hoạt có lẽ, hiệu quả? Không. Sử dụng bộ điều hợp lớp chỉ có một tôi, không phải bộ điều hợp và người được điều hợp.

Bạn lo lắng về một đối tư ợng nhỏ? Bạn có thể nhanh chóng ghi đè một phuong thức, nhưng bất kỳ hành vi nào tôi thêm vào mã bộ điều hợp của mình đều hoạt động với lớp adaptee và tất cả các lớp con của nó.

Vâng, nhưng nếu một lớp con của adaptee bổ sung thêm một số hành vi mới thì sao? Rồi sao nữa?

Này, thôi nào, cho tôi xin một chút, tôi chỉ cần soạn thảo với lớp con để thực hiện việc đó thôi.

Nghe có vẻ lộn xộn...

Bạn muốn thấy sự lộn xộn? Hãy nhìn vào gư ơng!

bộ chuyển đổi thế giới thực

Bộ điều hợp thế giới thực

Hãy cùng xem xét việc sử dụng một Adapter đơn giản trong thế giới thực (ít nhất là một thứ gì đó nghiêm túc hơn Ducks)...

Người đếm thế giới cũ

Nếu bạn đã làm việc với Java một thời gian, bạn có thể nhớ rằng các kiểu bộ sưu tập ban đầu (Vector, Stack, Hashtable và một số kiểu khác) triển khai phương thức elements(), trả về một Enumeration. Giao diện Enumeration cho phép bạn duyệt qua các phần tử của một bộ sưu tập mà không cần biết chi tiết về cách chúng được quản lý trong bộ sưu tập đó.

Phép liệt kê có giao diện đơn giản.

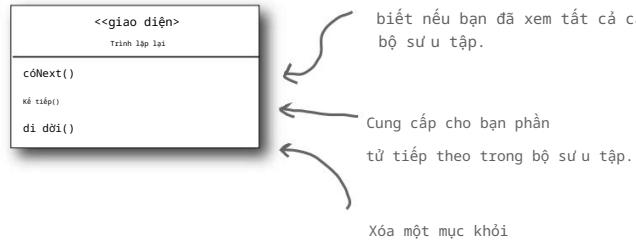


Trình lặp thế giới mới

Khi Sun phát hành các lớp Collections gần đây hơn, họ bắt đầu sử dụng giao diện Iterator, giống như Enumeration, cho phép bạn lặp qua một tập hợp các mục trong một bộ sưu tập, nhưng cũng bổ sung thêm khả năng xóa các mục.

Tương tự như `hasMoreElements()` trong giao diện Enumeration.

Phương pháp này chỉ cho bạn biết nếu bạn đã xem tất cả các mục trong bộ sưu tập.

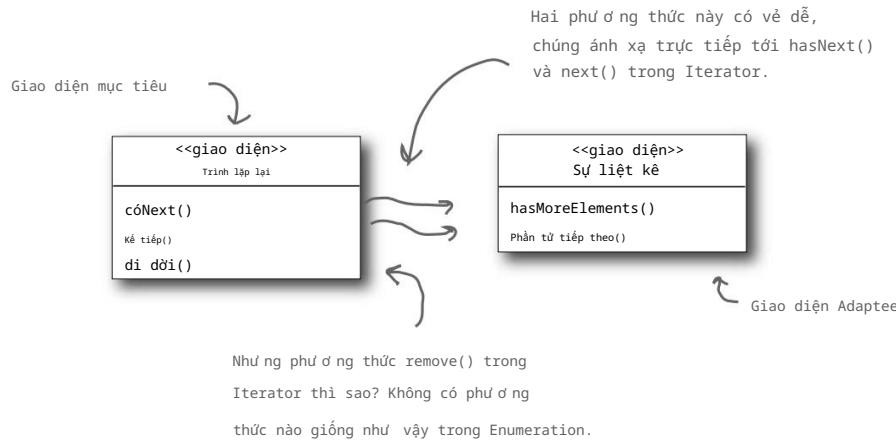


Và hôm nay...

Chúng ta thường phải đổi mặt với mã cũ hiển thị giao diện Enumerator, nhưng chúng ta muốn mã mới của mình chỉ sử dụng Iterator. Có vẻ như chúng ta cần xây dựng một bộ điều hợp.

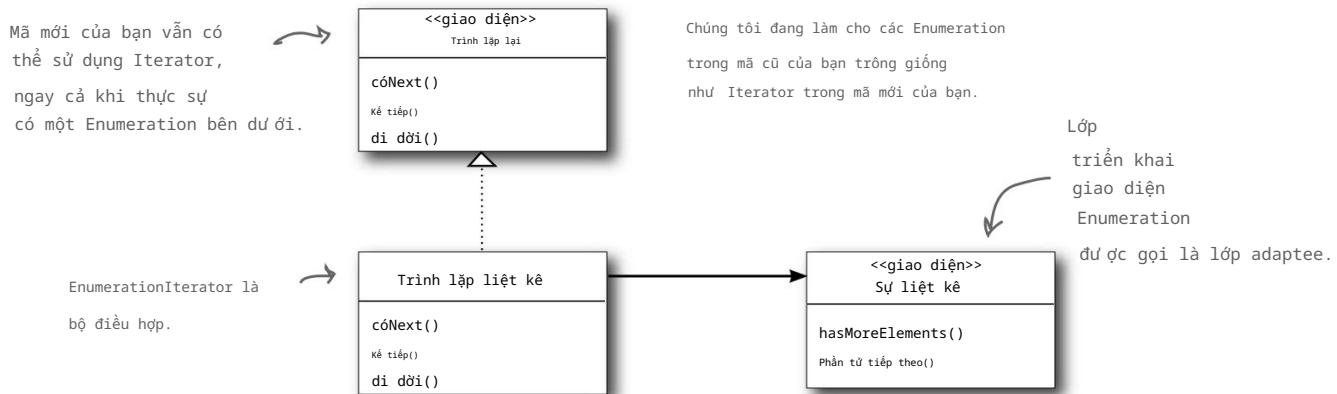
Điều chỉnh một Enumeration thành một Iterator

Đầu tiên, chúng ta sẽ xem xét hai giao diện để tìm ra cách các phương thức ánh xạ từ giao diện này sang giao diện kia. Nói cách khác, chúng ta sẽ tìm ra cách gọi phương thức nào trên adaptee khi máy khách gọi phương thức trên mục tiêu.



Thiết kế bộ chuyển đổi

Đây là những gì các lớp nên trông như thế này: chúng ta cần một bộ điều hợp triển khai giao diện Target và được tạo thành từ một adaptee. Các phương thức hasNext() và next() sẽ dễ dàng ánh xạ từ target đến adaptee: chúng ta chỉ cần truyền chúng qua. Nhưng bạn làm gì với remove()? Hãy suy nghĩ về nó trong giây lát (và chúng ta sẽ giải quyết nó ở trang tiếp theo).
Bây giờ, đây là sơ đồ lớp:



bộ điều hợp lặp lại liệt kê

Xử lý phư ơng thức remove()

Vâng, chúng ta biết Enumeration không hỗ trợ remove. Đó là giao diện "chỉ đọc".

Không có cách nào để triển khai phư ơng thức remove() hoạt động đầy đủ trên bộ điều hợp. Điều tốt nhất chúng ta có thể làm là ném ngoại lệ thời gian chạy. May mắn thay, các nhà thiết kế giao diện Iterator đã lường trước nhu cầu này và định nghĩa phư ơng thức remove() để nó hỗ trợ UnsupportedOperationException.

Đây là trường hợp bộ điều hợp không hoàn hảo; khách hàng sẽ phải cẩn thận với các trường hợp ngoại lệ tiềm ẩn, như ng miến là khách hàng cẩn thận và bộ điều hợp được ghi chép đầy đủ thì đây là giải pháp hoàn toàn hợp lý.

Viết bộ điều hợp EnumerationIterator

Sau đây là mã đơn giản như ng hiệu quả cho tất cả các lớp cũ vẫn tạo ra Enumeration:

```

lớp công khai EnumerationIterator thực hiện Iterator {
    Kiểu liệt kê enum;
    công khai EnumerationIterator(Enumeration enum) {
        this.enum = enum;
    }
    boolean công khai hasNext() {
        trả về enum.hasMoreElements();
    }
    Đối tượng công khai tiếp theo() {
        trả về enum.nextElement();
    }
    công khai void remove() {
        ném UnsupportedOperationException mới();
    }
}

```

Vì chúng ta đang điều chỉnh Enumeration thành Iterator, nên Adapter của chúng ta sẽ triển khai giao diện Iterator... nó phải trông giống như một Iterator.

Chúng tôi đang điều chỉnh phép liệt kê. Chúng tôi sử dụng thành phần nên chúng tôi lưu trữ nó trong một biến thể hiện.

Phư ơng thức hasNext() của Iterator được chuyển giao cho phư ơng thức hasMoreElements() của Enumeration...

... và phư ơng thức next() của Iterator được chuyển giao cho phư ơng thức nextElement() của Enumerations.

Thật không may, chúng ta không thể hỗ trợ phư ơng thức remove() của Iterator, vì vậy chúng ta phải bỏ cuộc (nói cách khác là chúng ta bỏ cuộc!). Ở đây chúng ta chỉ ném một ngoại lệ.



Bài tập

Mặc dù Java đã đi theo hướng của Iterator, nhưng vẫn có rất nhiều mã máy khách cũ phụ thuộc vào giao diện Enumeration, do đó, một Adapter chuyển đổi Iterator thành Enumeration cũng khá hữu ích.

Viết một Adapter điều chỉnh Iterator thành Enumeration. Bạn có thể kiểm tra mã của mình bằng cách điều chỉnh ArrayList. Lớp ArrayList hỗ trợ giao diện Iterator nhưng không hỗ trợ Enumeration (tốt, ít nhất là chưa).

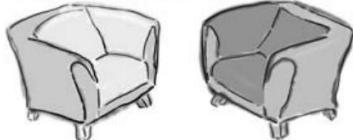
não Apower

Một số bộ đổi nguồn AC không chỉ thay đổi giao diện mà còn bổ sung các tính năng khác như chống sét lan truyền, đèn báo và nhiều tính năng khác.

Nếu bạn định triển khai những tính năng này, bạn sẽ sử dụng mẫu nào?

trò chuyện bên lò sưởi: người trang trí và người điều chỉnh

Fireside Chats



Người trang trí

Bộ chuyển đổi

Bài nói chuyện tối nay: Mẫu trang trí và mẫu bộ điều hợp thảo luận về sự khác biệt của chúng.

Tôi quan trọng. Công việc của tôi liên quan đến trách nhiệm - bạn biết rằng khi một Decorator tham gia, sẽ có một số trách nhiệm hoặc hành vi mới được thêm vào thiết kế của bạn.

Các bạn muốn mọi vinh quang trong khi chúng tôi, những người chuyển đổi, đang làm công việc bẩn thỉu: chuyển đổi giao diện. Công việc của chúng tôi có thể không hấp dẫn, nhưng khách hàng của chúng tôi chắc chắn đánh giá cao việc chúng tôi làm cho cuộc sống của họ đơn giản hơn.

Điều đó có thể đúng, như ng đừng nghĩ chúng tôi không làm việc chăm chỉ. Khi chúng tôi phải trang trí một giao diện lớn, trời ạ, điều đó có thể tốn rất nhiều mă.

Hãy thử làm một bộ điều hợp khi bạn phải kết hợp nhiều lớp lại với nhau để cung cấp giao diện mà khách hàng của bạn mong đợi. Giờ thì khó rồi. Nhưng chúng tôi có câu nói: "một khách hàng không liên kết là một khách hàng hạnh phúc".

Dễ thư ơng. Đừng nghĩ rằng chúng ta nhận được tất cả vinh quang; đôi khi tôi chỉ là một decorator được bao bọc bởi không biết bao nhiêu decorator khác. Khi một cuộc gọi phuơng thức được giao cho bạn, bạn không biết có bao nhiêu decorator khác đã xử lý nó và bạn không biết rằng bạn sẽ được chú ý vì những nỗ lực phục vụ yêu cầu của mình.

Này, nếu bộ điều hợp đang làm tốt nhiệm vụ của mình, khách hàng của chúng tôi thậm chí không biết chúng tôi ở đó. Đó có thể là một công việc vô ơn.

mẫu bộ chuyển đổi

Người trang trí

Bộ chuyển đổi

Như ng điều tuyệt vời về chúng tôi, những người điều hợp, là chúng tôi cho phép khách hàng sử dụng các thư viện và tập hợp con mới mà không cần thay đổi bất kỳ mã nào, họ chỉ cần dựa vào chúng tôi để thực hiện chuyển đổi cho họ. Này, đó là một ngách, nhưng chúng tôi giỏi về điều đó.

Vâng, chúng tôi, những người trang trí cũng làm như vậy, chỉ là chúng tôi cho phép thêm hành vi mới vào các lớp mà không thay đổi mã hiện có.
Tôi vẫn nói rằng các bộ điều hợp chỉ là những người trang trí lạ mắt - ý tôi là, giống như chúng tôi, bạn bao bọc một đổi tư duy.

Không, không, không, không hề. Chúng tôi luôn chuyển đổi giao diện của những gì chúng tôi gói, còn bạn thì không bao giờ làm vậy. Tôi cho rằng decorator giống như một bộ điều hợp; chỉ là bạn không thay đổi giao diện!

Ồ, không. Nhiệm vụ của chúng ta trong cuộc sống là mở rộng hành vi hoặc trách nhiệm của những đồ vật mà chúng ta gói ghém, chúng ta không phải là người chỉ đơn giản đi qua.

Này, bạn gọi ai là người chuyển tiếp đơn giản vậy? Hãy xuống đây và chúng ta sẽ xem bạn có thể chuyển đổi một vài giao diện trong bao lâu!

Có lẽ chúng ta nên đồng ý bắt đồng quan điểm. Chúng ta có vẻ khá giống nhau trên lý thuyết, nhưng rõ ràng là chúng ta có mục đích khác xa nhau .

Ồ vâng, tôi đồng ý với bạn.

Ai làm gì?

Và bây giờ là điều gì đó khác biệt...

Có một mô hình khác trong chương này.

Bạn đã thấy cách Adapter Pattern chuyển đổi giao diện của một lớp thành giao diện mà khách hàng mong đợi. Bạn cũng biết chúng ta đạt được điều này trong Java bằng cách gói đổi tư ợng có giao diện không tương thích với đổi tư ợng triển khai giao diện chính xác.

Bây giờ chúng ta sẽ xem xét một mẫu thay đổi giao diện, nhưng vì một lý do khác: để đơn giản hóa giao diện. Nó đạt được đặt tên khéo léo là Mẫu Facade vì mẫu này ẩn tất cả sự phức tạp của một hoặc nhiều lớp đằng sau một mặt tiền sạch sẽ, sáng sửa.



Ghép mỗi mẫu với mục đích của nó:

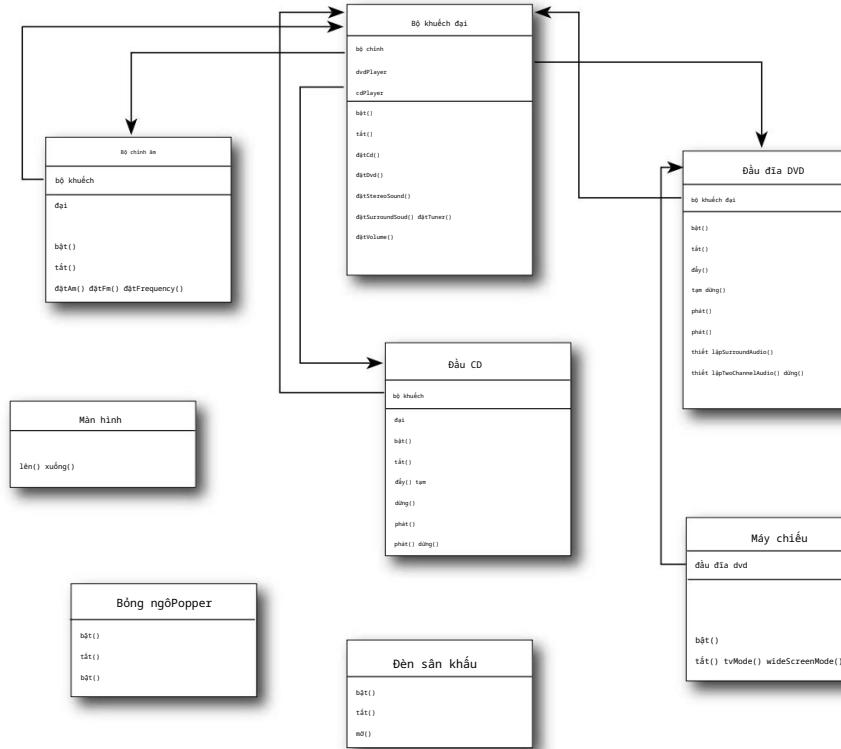
Mẫu	Ý định
Người trang trí	Chuyển đổi một giao diện sang giao diện khác
Bộ chuyển đổi	Không thay đổi giao diện như những tăng thêm trách nhiệm
Mặt tiền	Làm cho giao diện đơn giản hơn

Rạp hát tại nhà ngọt ngào

Trước khi đi sâu vào chi tiết về Mẫu mặt tiền, chúng ta hãy cùng xem xét một nỗi ám ảnh đang ngày càng lan rộng trên toàn quốc: xây dựng rạp hát tại nhà cho riêng bạn.

Bạn đã nghiên cứu và lắp ráp được một hệ thống hoàn chỉnh với đầu đĩa DVD, hệ thống video chiếu, màn hình tự động, âm thanh vòm và thậm chí cả máy nổ bong bóng.

Kiểm tra tất cả các thành phần bạn đã ghép lại với nhau:



Đó là rất nhiều lớp học, rất nhiều tư ng
tác và một bộ giao diện lớn
để học và sử dụng

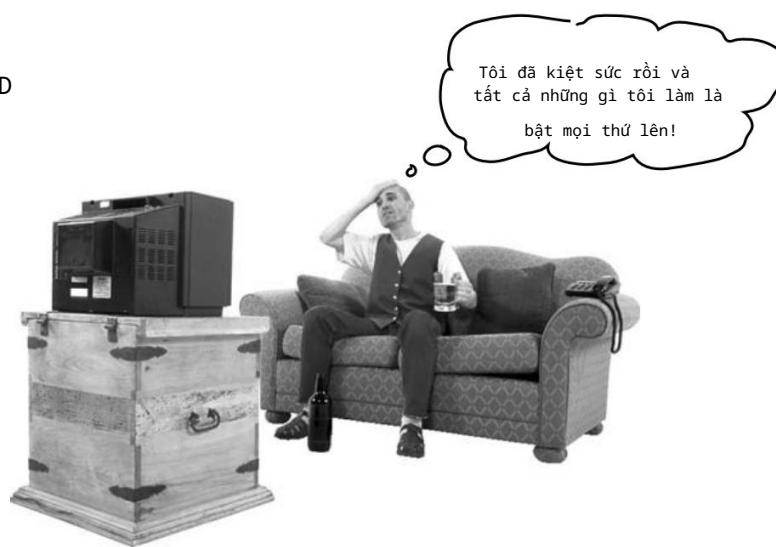
Bạn đã dành nhiều tuần để chạy dây, lắp máy chiếu, thực hiện tất cả các kết nối và tinh chỉnh. Böyle giờ là lúc đưa tất cả vào hoạt động và thư ờng thức một bộ phim...

nhiệm vụ xem phim

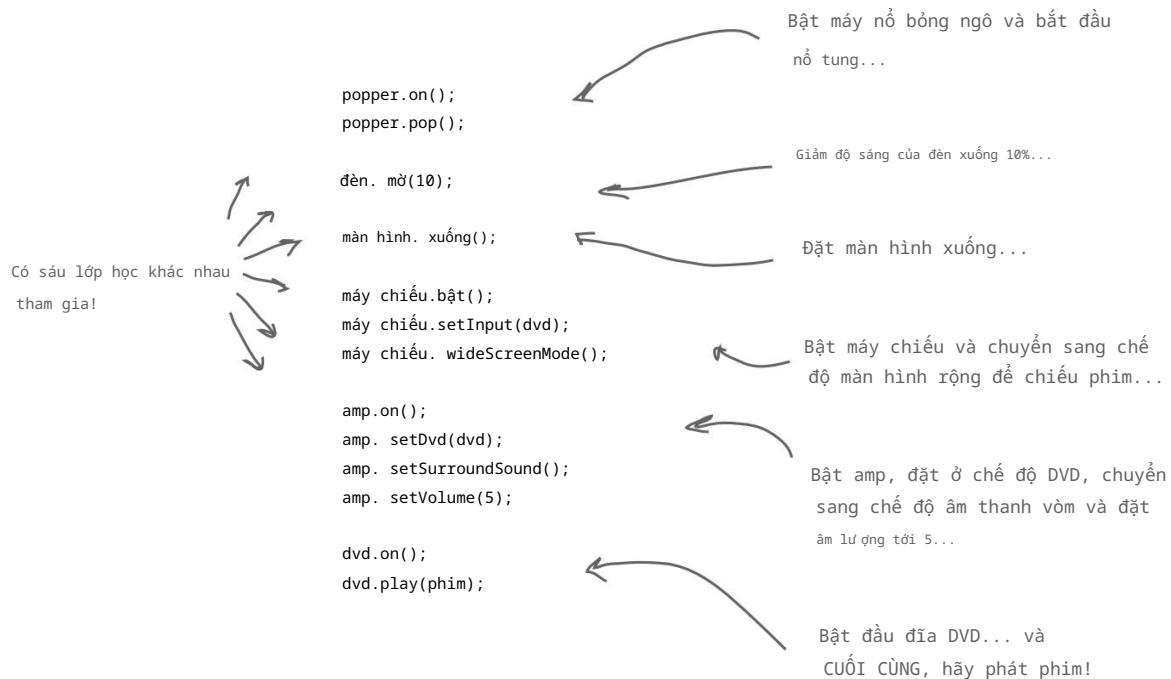
Xem phim (theo cách khó khăn)

Chọn một đĩa DVD, thư giãn và chuẩn bị cho phép thuật điện ảnh. Ô, chỉ có một điều - để xem phim, bạn cần thực hiện một vài nhiệm vụ:

- ① Bật máy nổ bóng ngô
- ② Bắt đầu popper popping
- ③ Làm mờ đèn
- ④ Đặt màn hình xuống
- ⑤ Bật máy chiếu lên
- ⑥ Đặt đầu vào máy chiếu thành DVD
- ⑦ Đặt máy chiếu ở chế độ màn hình rộng
- ⑧ Bật bộ khuếch đại âm thanh
- ⑨ Đặt bộ khuếch đại vào đầu vào DVD
- ⑩ Đặt bộ khuếch đại ở chế độ âm thanh vòm
- ⑪ Đặt âm lượng của bộ khuếch đại ở mức trung bình (5)
- ⑫ Bật đầu DVD
- ⑬ Bắt đầu phát đầu DVD



Hãy cùng kiểm tra những tác vụ tương tự đó theo các lớp và các phương thức gọi cần thiết để thực hiện chúng:



Như ng vẫn còn nhiều hơn thế nữa...

Khi bộ phim kết thúc, bạn tắt mọi thứ như thế nào?

Chẳng phải bạn sẽ phải làm lại tất cả những điều này theo chiều ngược lại sao?

β Nghe đĩa CD hay nghe radio có phức tạp hơn không?

β Nếu bạn quyết định nâng cấp hệ thống, có thể bạn sẽ phải học một quy trình hơi khác một chút.

Vậy phải làm gì? Sự phức tạp khi sử dụng rạp hát tại nhà đang trở nên rõ ràng!

Hãy cùng xem Facade Pattern có thể giúp chúng ta thoát khỏi tình trạng hỗn loạn này như thế nào để có thể thu ông thức bộ phim nhé...

đèn máy ảnh mặt tiền

Đèn, Camera, Mặt tiền!

Facade chính là thứ bạn cần: với Facade Pattern, bạn có thể sử dụng một hệ thống phức tạp và làm cho nó dễ sử dụng hơn bằng cách triển khai một lớp Facade cung cấp một giao diện hợp lý hơn. Đừng lo lắng; nếu bạn cần sức mạnh của hệ thống con phức tạp, nó vẫn ở đó để bạn sử dụng, nhưng nếu tất cả những gì bạn cần là một giao diện đơn giản, Facade sẽ ở đó dành cho bạn.

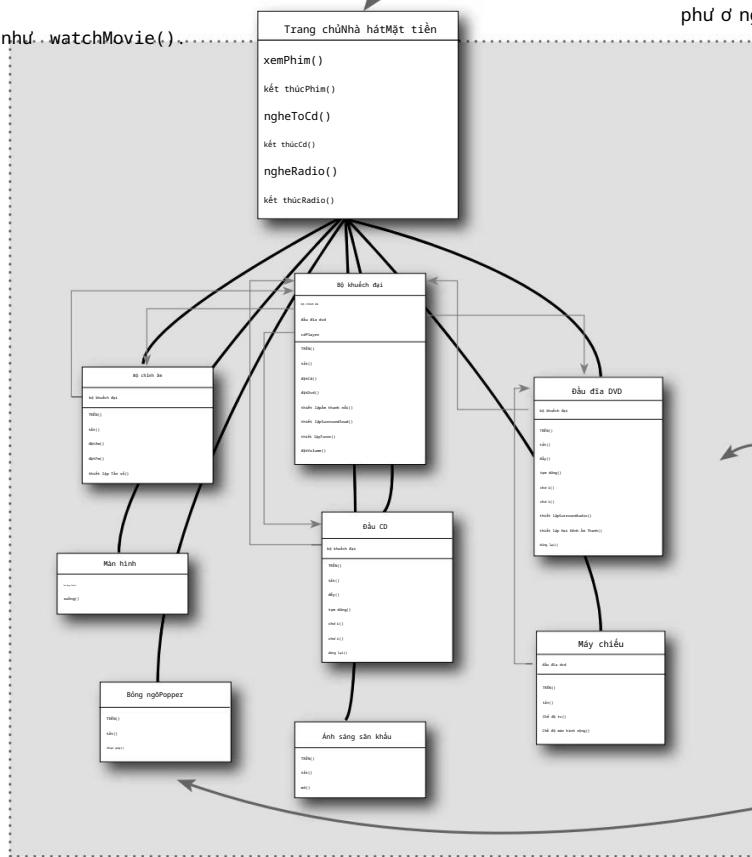
Hãy cùng xem Facade hoạt động như thế nào:

- Đư ợc rồi, đèn lúc tạo Facade cho hệ thống rạp hát tại nhà. Để làm điều này, chúng ta tạo một lớp mới HomeTheaterFacade, lớp này hiển thị một số phu օ ng th ứ c đ ờ n gi ǎn như .watchMovie().....

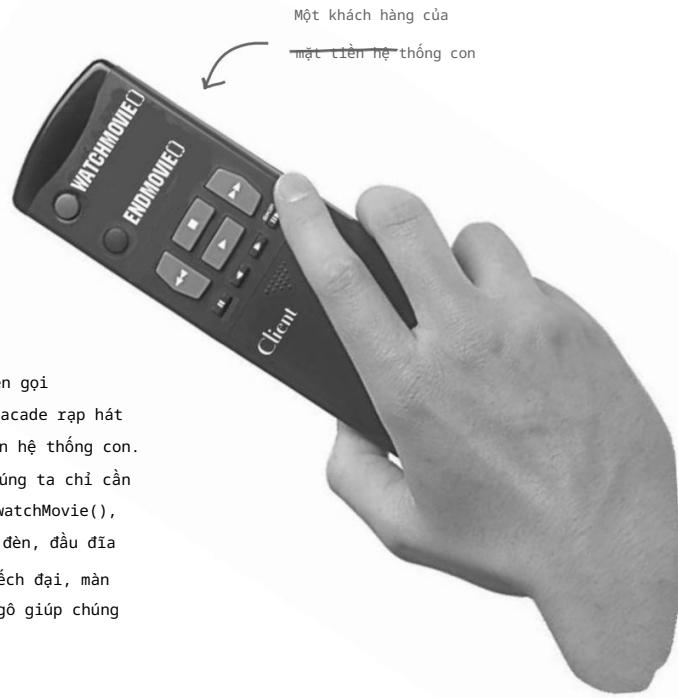
Mặt tiền

- Lớp Facade xử lý các thành phần rạp hát tại nhà như một hệ thống con và gọi hệ thống con đó để triển khai phu օ ng th ứ c watchMovie() của nó.

Hệ thống con mà Facade đang đơn giản hóa.



mẫu bộ chuyển đổi

xemPhim()

- 3 Mã máy khách của bạn hiện gọi các phuơng thức trên Facade rạp hát tại nhà, không phải trên hệ thống con. Bởi giờ dễ xem phim, chúng ta chỉ cần gọi một phuơng thức, `watchMovie()`, và nó sẽ giao tiếp với đèn, đầu đĩa DVD, máy chiếu, bộ khuếch đại, màn hình và máy làm bóng ngồi giúp chúng ta.



- 4 Facade vẫn để lại hệ thống con có thể truy cập để sử dụng trực tiếp. Nếu bạn cần chức năng nâng cao của các lớp hệ thống con, chúng có sẵn để bạn sử dụng.

mặt tiền so với bộ chuyển đổi

không có Những câu hỏi ngắn

Q: Nếu Facade bao gồm các lớp hệ thống con, làm thế nào để máy khách cần chức năng cấp thấp hơn có thể truy cập vào chúng?

A: Mặt tiền không “bao bọc” các lớp hệ thống con; chúng chỉ cung cấp một giao diện đơn giản hóa cho chức năng của chúng. Các lớp hệ thống con vẫn có sẵn để sử dụng trực tiếp bởi các máy khách cần sử dụng các giao diện cụ thể hơn. Đây là một thuộc tính hay của Facade Pattern: nó cung cấp một giao diện đơn giản hóa trong khi vẫn cung cấp đầy đủ chức năng của hệ thống cho những người có thể cần đến nó.

Q: Mặt tiền có thêm bất kỳ chức năng hay nó chỉ chuyển từng yêu cầu đến hệ thống con?

A: Mặt tiền có thể tự do thêm vào “thông minh” ngoài việc sử dụng hệ thống con. Ví dụ, trong khi mặt tiền rạp hát tại nhà của chúng tôi không triển khai bất kỳ hành vi mới nào, nó đủ thông minh để biết rằng máy nổ bóng ngô phải được bật trước khi nó có thể nổ (cũng như thông tin chi tiết về cách bật và dàn dựng một buổi chiếu phim).

Q: Mỗi hệ thống con chỉ có một mặt tiền?

A: Không nhất thiết. Mẫu chắc chắn cho phép tạo ra bất kỳ số lượng mặt tiền nào cho một hệ thống con nhất định.

Q: Lợi ích của mặt tiền là gì? ngoài việc bây giờ tôi có giao diện đơn giản hơn?

A: Mẫu mặt tiền cũng cho phép bạn tách biệt việc triển khai máy khách của mình khỏi bất kỳ hệ thống con nào. Giả sử bạn được tăng lương và quyết định nâng cấp rạp hát tại nhà của mình lên tất cả các thành phần mới có giao diện khác nhau. Vâng, nếu bạn mã hóa máy khách của mình thành facade thay vì hệ thống con, thì mã máy khách của bạn không cần phải thay đổi, chỉ cần facade (và hy vọng là nhà sản xuất sẽ cung cấp điều đó!).

Q: Vậy cách để phân biệt giữa Adapter Pattern và Facade Pattern là adapter bao bọc một lớp còn Facade có thể biểu diễn nhiều lớp?

A: Không! Hãy nhớ, Mẫu Bộ chuyển đổi thay đổi giao diện của một hoặc nhiều lớp thành một giao diện mà máy khách mong đợi. Trong khi hầu hết các ví dụ trong sách giáo khoa đều cho thấy bộ điều hợp thích ứng với một lớp, bạn có thể cảm thấy ứng với nhiều lớp để cung cấp giao diện mà máy khách đư ợc mã hóa. Tương tự như vậy, Facade có thể cung cấp một giao diện đơn giản hóa cho một lớp duy nhất có giao diện rất phức tạp.

Sự khác biệt giữa hai loại này không nằm ở số lượng lớp mà chúng “gói”, mà nằm ở mục đích của chúng. Mục đích của Adapter Pattern là thay đổi giao diện để nó khớp với giao diện mà khách hàng mong đợi. Mục đích của Facade Pattern là cung cấp giao diện đơn giản hóa cho hệ thống con.

Facade không chỉ

đơn giản hóa giao diện mà còn tách biệt máy khách khỏi hệ thống các thành phần.

Facades và bộ

điều hợp có thể bao gồm nhiều lớp, như ng mục đích của facades là đơn giản hóa, trong khi mục đích của bộ điều hợp là chuyển đổi giao diện sang thứ gì đó khác.

Xây dựng mặt tiền rạp hát tại nhà của bạn

Chúng ta hãy cùng tìm hiểu về quá trình xây dựng HomeTheaterFacade:

Bước đầu tiên là sử dụng bối cảnh để mặt tiền có thể truy cập vào tất cả các thành phần của hệ thống con:

```

lớp công khai HomeTheaterFacade {
    Bộ khuếch đại amp;
    Bộ chỉnh âm;
    Đầu đĩa DVD DVD;
    Đầu đĩa CD CdPlayer;
    Máy chiếu máy chiếu;
    Đèn TheaterLights;
    Màn hình màn hình;
    Bóng ngô;

    public HomeTheaterFacade(Bộ khuếch đại amp,
        Bộ chỉnh âm,
        Đầu đĩa DVD DVD,
        Đầu đĩa CD,
        Máy chiếu máy chiếu,
        Màn hình màn hình,
        Đèn TheaterLights,
        Bóng ngô (popper popper) {

        this.amp = amp;
        this.tuner = bộ chỉnh dây;
        this.dvd = dvd;
        this.cd = cd;
        this.projector = máy chiếu;
        this.screen = màn hình;
        this.lights = đèn;
        this.popper = người làm popper;
    }

    // các phương pháp khác ở đây
}

```

Sau đây là thành phần; đây là tất cả các thành phần của hệ thống con mà chúng ta sẽ sử dụng.

Facade được truyền tham chiếu đến từng thành phần của hệ thống con trong hàm tạo của nó.

Sau đó, Facade gán từng thành phần cho biến thể hiện tương ứng.

Chúng tôi sắp hoàn thành những thông tin này...

thực hiện mặt tiền

Triển khai giao diện đơn giản hóa

Bây giờ là lúc kết hợp các thành phần của hệ thống con thành một giao diện thống nhất.

Hãy triển khai các phương thức `watchMovie()` và `endMovie()`:

```
public void watchMovie(String movie) {
    System.out.println("Chuẩn bị xem phim thôi...");
    popper.on();
    popper.pop();
    đèn.mờ(10);
    màn hình.xuống();
    máy chiếu.bật();
    máy chiếu.wideScreenMode();
    amp.on();
    amp.setDvd(dvd);
    amp.setSurroundSound();
    amp.setVolume(5);
    dvd.on();
    dvd.play(phim);
}
```

```
công khai void endMovie() {
    System.out.println("Đang đóng rạp chiếu phim...");
    popper.tắt();
    đèn.bật();
    screen.up();
    máy chiếu.tắt();
    amp.tắt();
    dvd.dừng();
    dvd.eject();
    dvd.tắt();
}
```

 `watchMovie()` tuân theo cùng một trình tự mà chúng ta phải thực hiện thủ công trước đó, nhưng gói gọn nó trong một phương thức tiện dụng thực hiện tất cả công việc. Lưu ý rằng đối với mỗi tác vụ, chúng ta đang giao phó trách nhiệm cho thành phần tương ứng trong hệ thống con.

 . Và `endMovie()` sẽ xử lý việc tắt mọi thứ cho chúng ta. Một lần nữa, mỗi tác vụ được giao cho thành phần thích hợp trong hệ thống con.

não Apower

Hãy nghĩ về những mặt tiền mà bạn đã gặp trong Java API.
Bạn muốn có một vài cái mới ở đâu?

Đã đến lúc xem phim (cách dễ dàng)

Đến giờ biểu diễn rồi!



```

lớp công khai HomeTheaterTestDrive {
    public static void main(String[] args) {
        // khởi tạo các thành phần ở đây
    }

    Trang chủNhà hátMặt tiền nhàNhà hát =
        HomeTheaterFacade mới (amp, bộ chính, dvd, cd,
        máy chiếu, màn hình, đèn, popper);

    homeTheater.watchMovie("Raiders of the Lost Ark");
    homeTheater.endMovie();
}

```

Ở đây chúng ta đang tạo các thành phần ngay trong quá trình chạy thử. Thông thường, khách hàng được cung cấp một mặt tiền, không cần phải tự xây dựng.

Đầu tiên, bạn khởi tạo Facade với tất cả các thành phần trong hệ thống con.

Sử dụng giao diện đơn giản để bắt đầu xem phim trước, sau đó tắt phim.

Đây là kết quả.

Gọi hàm watchMovie() của Facade sẽ thực hiện tất cả công việc này cho chúng ta...

```

Cửa sổ chính sửa tập tin Trợ giúp RắnTại sao phải là rắn?
%java HomeTheaterKiểm traDrive

Hãy chuẩn bị xem phim nhé...
Máy nổ bóng ngô trên
Popcorn Popper nổ bóng ngô!
Đèn trần rạp chiếu phim mờ đi 10%
Màn hình rạp chiếu phim đang hạ xuống
Máy chiếu Top-O-Line trên
Máy chiếu Top-O-Line ở chế độ màn hình rộng (tỷ lệ khung hình 16x9)
Bộ khuếch đại Top-O-Line đang bật
Bộ khuếch đại Top-O-Line cài đặt đầu DVD cho đầu DVD Top-O-Line
Bộ khuếch đại âm thanh vòm Top-O-Line (5 loa, 1 loa siêu trầm)
Bộ khuếch đại Top-O-Line cài đặt âm lượng ở mức 5
Đầu DVD Top-O-Line trên
Đầu DVD Top-O-Line đang phát "Raiders of the Lost Ark"
Đóng cửa rạp chiếu phim...
Máy nổ bóng ngô
Đèn trần nhà hát bật
Màn hình rạp chiếu phim đang được dựng lên
Tắt máy chiếu Top-O-Line
Bộ khuếch đại Top-O-Line tắt
Đầu DVD Top-O-Line đã dừng "Raiders of the Lost Ark"
Đầu phát DVD Top-O-Line đã ra
Đầu DVD Top-O-Line tắt
%

```

...và thế là chúng ta đã xem xong phim, vì vậy lệnh endMovie() sẽ tắt mọi thứ.

mẫu mặt tiền đư ợc xác định

Mẫu mặt tiền đư ợc xác định

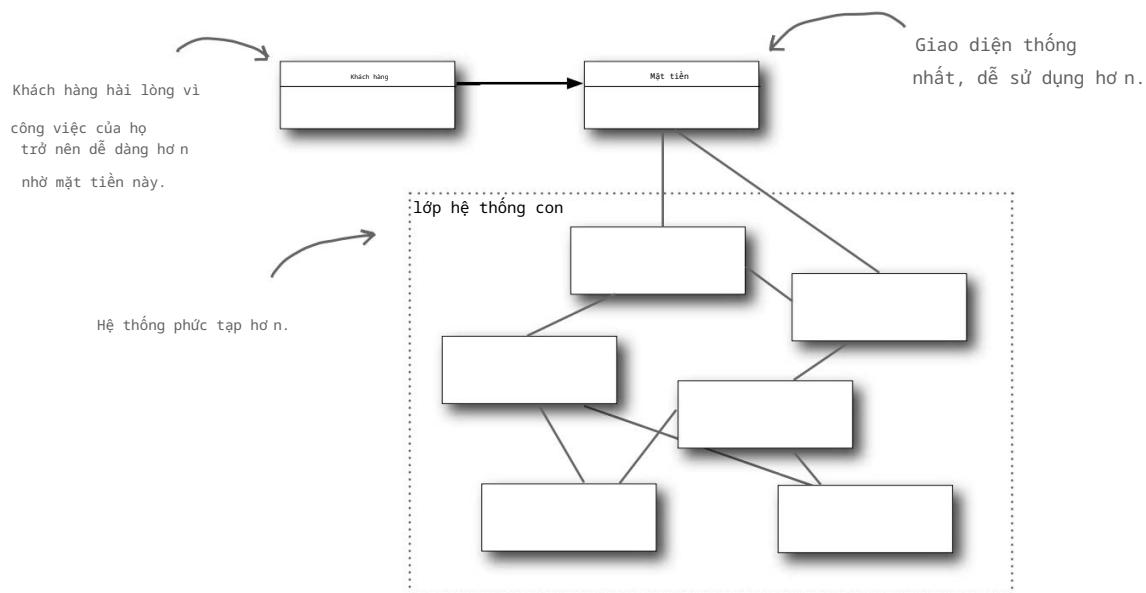
Để sử dụng Facade Pattern, chúng ta tạo một lớp đơn giản hóa và thống nhất một tập hợp các lớp phức tạp hơn thuộc về một số hệ thống con. Không giống như nhiều mẫu, Facade khá đơn giản; không có sự trừu tượng hóa nào khiến bạn phải đau đầu. Nhưng điều đó không làm cho nó kém mạnh mẽ hơn: Facade Pattern cho phép chúng ta tránh sự kết hợp chặt chẽ giữa máy khách và hệ thống con, và như bạn sẽ thấy ngay sau đây, cũng giúp chúng ta tuân thủ một nguyên tắc huyền ống đối tư duy mới.

Trước khi giới thiệu nguyên tắc mới này, chúng ta hãy cùng xem định nghĩa chính thức của mẫu này:

Mẫu Facade cung cấp một giao diện thống nhất cho một tập hợp các giao diện trong một hệ thống con. Facade định nghĩa một giao diện cấp cao hơn giúp hệ thống con dễ sử dụng hơn.

Không có nhiều điều ở đây mà bạn chưa biết, nhưng một trong những điều quan trọng nhất cần nhớ về một mẫu là mục đích của nó. Định nghĩa này cho chúng ta biết rõ ràng rằng mục đích của mặt tiền là làm cho một hệ thống con dễ sử dụng hơn thông qua một giao diện đơn giản hóa.

Bạn có thể thấy điều này trong sơ đồ lớp của mẫu:



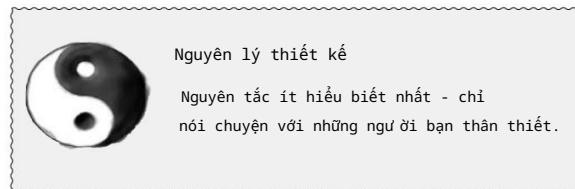
Vậy là xong; bạn đã có thêm một mẫu nữa! Bây giờ, đã đến lúc áp dụng nguyên lý OO mới.

Hãy cẩn thận, điều này có thể thách thức một số giả định!

Nguyên tắc ít kiến thức nhất

Nguyên tắc ít kiến thức nhất hứa hẹn dẫn chúng ta giảm sự tương tác giữa các đối tượng xuống chỉ còn một vài "người bạn" thân thiết.

Nguyên tắc thường được nêu như sau:



Nhưng điều này có ý nghĩa gì trong thực tế? Nghĩa là khi bạn thiết kế một hệ thống, đối với bất kỳ đối tượng nào, hãy cẩn thận về số lượng lớp mà nó tương tác và cách nó tương tác với các lớp đó.

Nguyên tắc này ngăn cản chúng ta tạo ra các thiết kế có số lượng lớn các lớp được ghép nối với nhau để các thay đổi trong một phần của hệ thống lan truyền sang các phần khác. Khi bạn xây dựng nhiều sự phụ thuộc giữa nhiều lớp, bạn đang xây dựng một hệ thống mong manh, tổn kém để duy trì và phức tạp để những người khác hiểu.

não Apower

Mã này được ghép nối với bao nhiêu lớp?

```
công khai float getTemp() {
    trả về.getThermometer().getTemperature();
}
```

nguyên tắc ít kiến thức nhất

Làm thế nào để KHÔNG giành được bạn bè và ảnh hưởng đến đối tư ợng

Được thôi, nhưng làm sao bạn không làm điều này? Nguyên tắc cung cấp một số hướng dẫn: lấy bất kỳ đối tư ợng nào; bây giờ từ bất kỳ phuơng thức nào trong đối tư ợng đó, nguyên tắc cho chúng ta biết rằng chúng ta chỉ nên gọi các phuơng thức thuộc về:

β Bản thân đối tư ợng

β Các đối tư ợng được truyền vào như một tham số cho phuơng thức

β Bất kỳ đối tư ợng nào mà phuơng thức tạo ra hoặc khởi tạo

β Bất kỳ thành phần nào của đối tư ợng

Lưu ý rằng các hướng dẫn này yêu cầu chúng ta không gọi các phuơng thức trên các đối tư ợng được trả về từ việc gọi các phuơng thức khác!!

Hãy nghĩ về một "thành phần" như bất kỳ đối tư ợng nào được thay đổi bởi một biến thể hiện. Nói cách khác, hãy nghĩ về điều này như một mối quan hệ CÓ-MỘT.

Nghe có vẻ hơi nghiêm ngặt phải không? Có hại gì khi gọi phuơng thức của một đối tư ợng mà chúng ta nhận được từ một cuộc gọi khác?
Vâng, nếu chúng ta làm như vậy, thì chúng ta sẽ đưa ra yêu cầu về phần phụ của đối tư ợng khác (và tăng số lượng đối tư ợng mà chúng ta biết trực tiếp). Trong những trường hợp như vậy, nguyên tắc buộc chúng ta phải yêu cầu đối tư ợng đưa ra yêu cầu thay cho chúng ta; theo cách đó, chúng ta không cần phải biết về các đối tư ợng thành phần của nó (và chúng ta giữ cho vòng tròn bạn bè của mình nhỏ). Ví dụ:

Không có
Nguyên tắc

```
công khai float getTemp() {
    Nhiệt kế nhiệt kế = station.getThermometer();
    trả về nhiệt kế.getTemperature();
}
```



Ở đây chúng ta lấy đối tư ợng nhiệt kế từ trạm và sau đó tự gọi phuơng thức getTemperature().

Với
Nguyên tắc

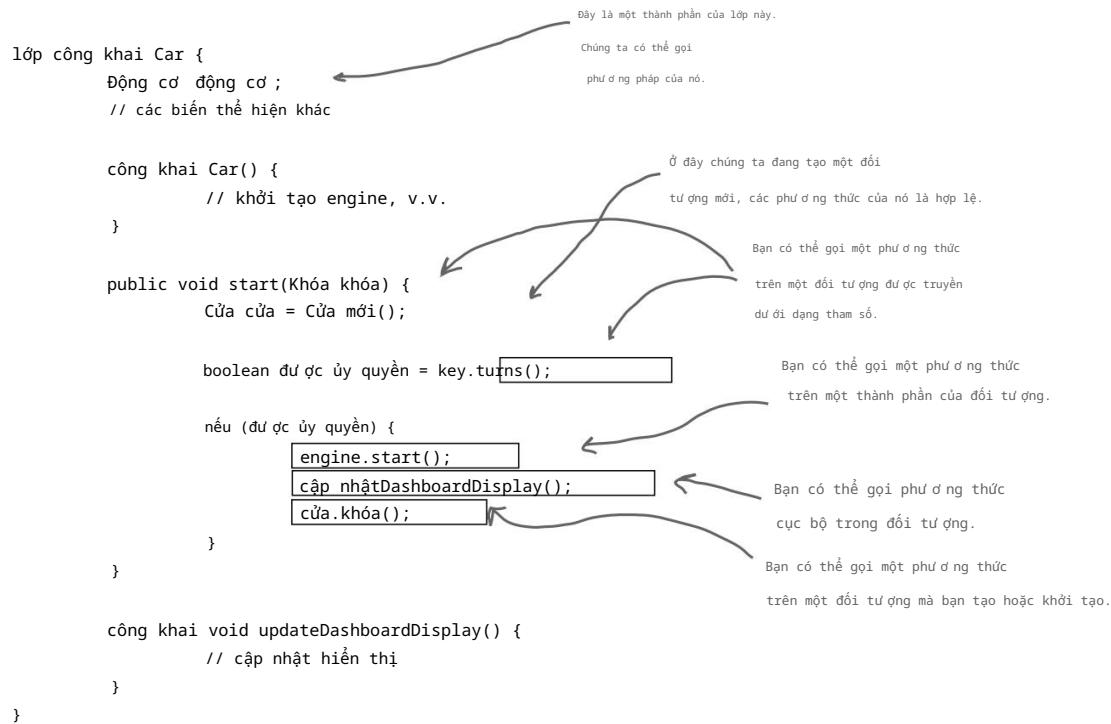
```
công khai float getTemp() {
    trả về station.getTemperature();
}
```



Khi áp dụng nguyên tắc này, chúng ta thêm một phuơng thức vào lớp Station để thực hiện yêu cầu tới nhiệt kế thay chúng ta. Điều này làm giảm số lượng lớp học mà chúng ta phải phụ thuộc.

Giữ cho các lệnh gọi phu ơng thức của bạn nằm trong giới hạn...

Sau đây là lớp Car minh họa tất cả các cách bạn có thể gọi phu ơng thức và vẫn tuân thủ Nguyên tắc kiến thức tối thiểu:



không có Những câu hỏi ngắn

H: Có một nguyên lý khác đư ợc gọi là Luật Demeter; chúng liên quan như thế nào?

A: Hai cái đó là một và giống nhau và bạn sẽ thấy những thuật ngữ này đư ợc trộn lẫn với nhau. Chúng tôi thích sử dụng Nguyên tắc ít kiến thức nhất vì một vài lý do: (1) tên gọi trực quan hơn và (2) việc sử dụng từ "Luật" ngữ ý rằng chúng ta luôn phải

áp dụng nguyên tắc này. Trên thực tế, không có nguyên tắc nào là luật, tất cả các nguyên tắc đều phải đư ợc sử dụng khi và ở nơi chúng hữu ích. Mọi thiết kế đều liên quan đến sự đánh đổi (trừu tư ơng so với tốc độ, không gian so với thời gian, v.v.) và trong khi các nguyên tắc cung cấp hướng dẫn, tất cả các yêu tố đều phải đư ợc tính đến trước khi áp dụng chúng.

Q: Có như ợc điểm nào không? để áp dụng Nguyên tắc kiến thức tối thiểu?

A: Vâng; trong khi nguyên tắc giảm sự phụ thuộc giữa các đối tượng và các nghiên cứu đã chỉ ra rằng điều này làm giảm việc bảo trì phần mềm, cũng như việc áp dụng nguyên tắc này dẫn đến nhiều lớp "wrapper" đư ợc viết để xử lý các lệnh gọi phu ơng thức đến các thành phần khác. Điều này có thể dẫn đến tăng độ phức tạp và thời gian phát triển cũng như giảm hiệu suất thời gian chạy.

vi phạm nguyên tắc ít hiểu biết nhất



Chuốt bút chì của bạn

Có lớp nào trong số này vi phạm Nguyên tắc ít kiến thức nhất không?

Tại sao nên hoặc không nên?

```
Nhà công cộng {
    Trạm thời tiết;
    // các phương thức và hàm tạo khác
    công khai float getTemp() { trả về
        trạm.getThermometer().getTemperature();
    }
}
```

```
Nhà công cộng {
    Trạm thời tiết;
    // các phương thức và hàm tạo khác
    public float getTemp() { Nhiệt kế
        nhiệt kế = station.getThermometer(); return getTempHelper(nhiệt kế);
    }
    public float getTempHelper(Nhiệt kế nhiệt kế) { return nhiệt kế.getTemperature();
    }
}
```



Khu vực mũ cứng. Hãy cẩn thận với những giả định sai lầm

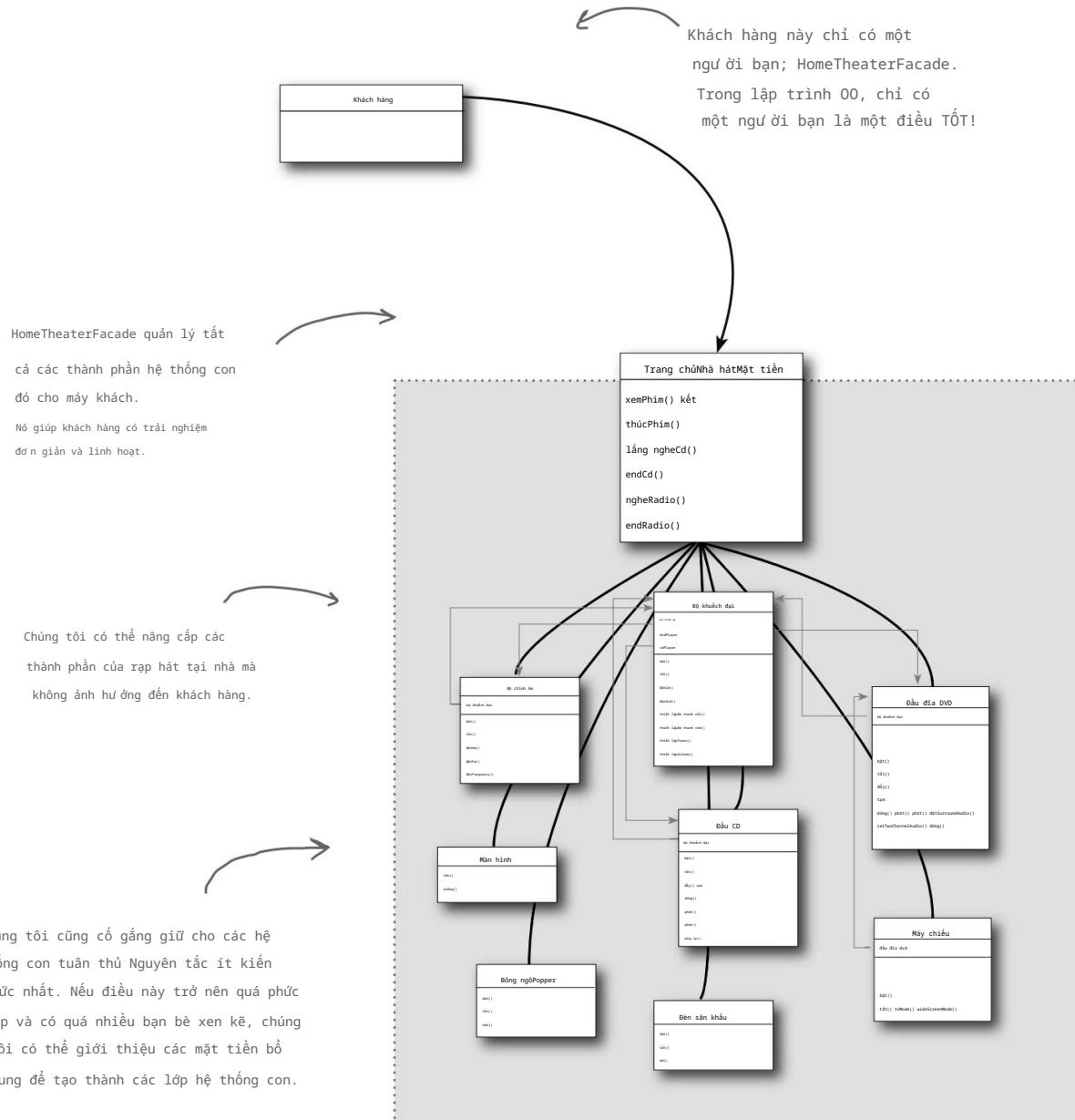
não Apower

Bạn có thể nghĩ ra cách sử dụng Java phổ biến nào vi phạm Nguyên tắc kiến thức tối thiểu không?

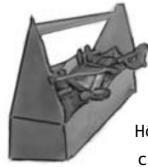
Bạn có nên quan tâm không?

Trả lời: Thẻ còn Sy

Mặt tiền và Nguyên tắc ít kiến thức nhất



hộp công cụ thiết kế của bạn



Công cụ cho hộp công cụ thiết kế của bạn

Hộp công cụ của bạn bắt đầu trở nên nặng nề! Trong chương này, chúng tôi đã thêm một số mẫu cho phép chúng tôi thay đổi giao diện và giảm sự kết hợp giữa máy khách và hệ thống mà họ sử dụng.

Nguyên tắc 00

- Bao gồm những gì thay đổi
- Ưu tiên thành phần hơn là thừa kế
- Chương trình giao diện, không phải triển khai
- Cố gắng thiết kế các đối tượng tư duy
- tư duy cách một cách lồng léo
- Các lớp học nên mở để mở rộng nhưng đóng để sửa đổi
- Phụ thuộc vào sự truu tư duy. Không phụ thuộc vào sự cụ thể
- Chỉ nói chuyện với bạn bè của bạn

Mẫu 00

Khi bạn cần thiết kế một hệ thống có một số thành phần riêng biệt, bạn có thể áp dụng **Định kèm cách** để tạo ra một **Abstract Factory** để quản lý cách tạo ra các thành phần. **Định kèm cách** là một cách tốt để duy trì một cách tiếp cận linh hoạt với các thành phần khác nhau. Tuy nhiên, nó có thể gây ra sự phức tạp và khó khăn trong việc thay đổi.

Để giải quyết vấn đề này, **Facade** là một cách tốt để cung cấp một giao diện đơn giản cho một hệ thống phức tạp. **Facade** định nghĩa giao diện cấp cao hơn cho các điều đó làm cho hệ thống con dễ sử dụng hơn.

ĐIỂM ĐẦU TIÊN

β Khi bạn cần sử dụng một

lớp hiện tại và giao diện của nó không phải là thứ bạn cần, hãy sử dụng bộ điều hợp.

β Khi bạn cần đơn giản hóa

và thống nhất một giao diện lớn hoặc một tập hợp giao diện phức tạp, hãy sử dụng mặt tiền.

β Một bộ điều hợp thay đổi một giao diện thành giao diện mà khách hàng mong đợi.

β Mặt tiền tách biệt khách hàng khỏi hệ thống con phức tạp.

β Việc triển khai một bộ điều hợp có thể yêu cầu ít hoặc nhiều công sức tùy thuộc vào kích thước và độ phức tạp của giao diện mục tiêu.

Để triển khai một facade, chúng ta cần phải kết hợp facade với hệ thống con của nó và sử dụng sự phân quyền để thực hiện công việc của facade.

Có hai dạng của Adapter Pattern: bộ

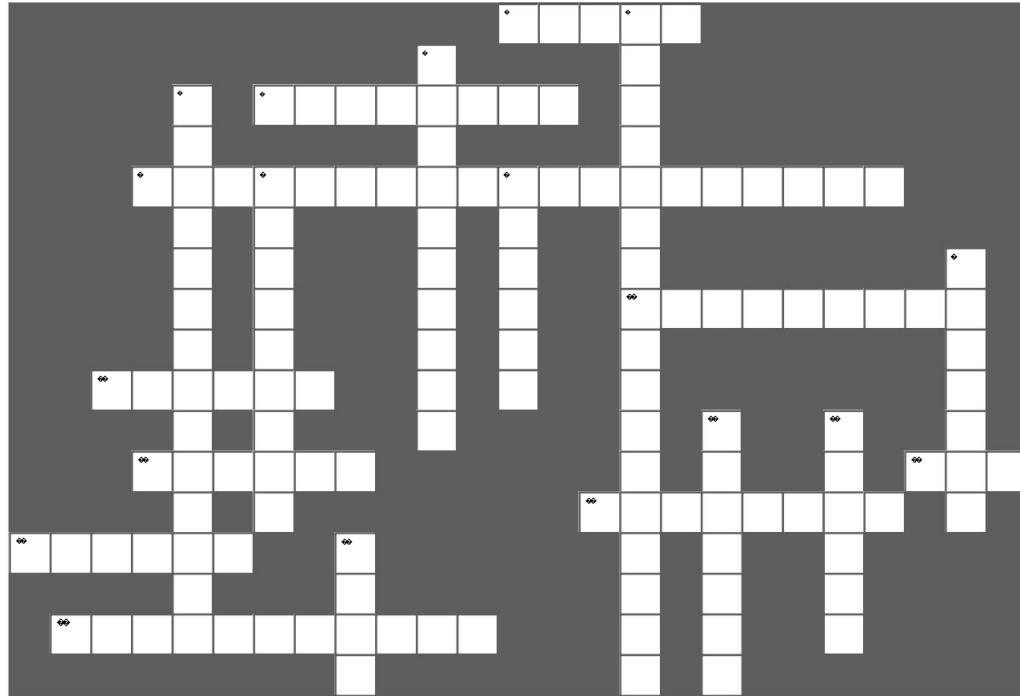
điều hợp đối tư duy và bộ điều hợp lớp. Bộ điều hợp lớp yêu cầu đa kế thừa.

β Bạn có thể triển khai nhiều hơn một mặt tiền cho một hệ thống con.

β Một bộ điều hợp bao bọc một đối tượng để thay đổi giao diện, trình trang trí bao bọc một đối tượng để thêm các hành vi và trách nhiệm mới, và mặt tiền "bao bọc" một tập hợp các đối tượng để đơn giản hóa.



Vâng, đây là một trò chơi ô chữ khác. Tất cả các từ giải đều nằm trong chuỗi này.



?

77 7777 7777 777777 777777 777777 777777 777777
777777

77 77 7777777 777777 77 77777777

77 777777 77 777777 77 777777 77 777777

777 77 77 77 777777 777777 777777 777777 777777

7777 777777

777 777777 777777 777777 777777 777777 777777
777 777777 777777 777777 777777

777 777777 777777 777777 777777
777 7777777777 77 777 7777777 77777777 77 777777
7777777777 777 7777 777777 7777777777
777 7 7777777777 7777777777 77 7777777777

777 777 77777777 777777 777777 777777

3

bạn đang ở đây 4 271

giải bài tập



Giải pháp bài tập



Chuốt bút chì của bạn

Giả sử chúng ta cũng cần một bộ chuyển đổi có thể chuyển đổi Vịt thành Gà tây. Hãy gọi nó là DuckAdapter. Viết lớp đó:

```
lớp công khai DuckAdapter triễn khai Turkey {
    Vịt vịt;
    Rand ngẫu nhiên;

    công khai DuckAdapter(Vịt vịt) {
        this.duck = vịt;
        rand = new Ngẫu nhiên();
    }

    công khai void gobble() {
        vịt.quack();
    }

    công khai void fly() {
        nếu (rand.nextInt(5) == 0) {
            vịt.ruồi();
        }
    }
}
```

Bây giờ chúng ta đang chuyển đổi Gà tây thành Vịt, vì vậy chúng ta triển khai giao diện Gà tây.

Chúng tôi lưu lại một tham chiếu đến chú Vịt mà chúng tôi đang chuyển đổi.

Chúng tôi cũng tạo lại một đối tượng ngẫu nhiên; hãy xem phương thức fly() để xem nó được sử dụng như thế nào.

Một tiếng gobble chỉ trở thành một tiếng quack.

Vì vịt bay lâu hơn gà tây rất nhiều nên chúng tôi quyết định chỉ cho vịt bay trung bình một trong năm lần.



Chuốt bút chì của bạn

Có lớp nào trong số này vi phạm Nguyên tắc ít kiến thức nhất không? Với mỗi trường hợp, tại sao có hoặc không?

```
Nhà công cộng {
    Trạm thời tiết;

    // các phương thức và hàm tạo khác

    công khai float getTemp() {
        trả về.getThermometer().getTemperature();
    }
}

Nhà công cộng {
    Trạm thời tiết;

    // các phương thức và hàm tạo khác

    công khai float getTemp() {
        Nhiệt kế nhiệt kế = station.getThermometer();
        trả về.getTempHelper(nhiệt kế);
    }

    public float getTempHelper(Nhiệt kế nhiệt kế) {
        trả về.nhiệt kế.getTemperature();
    }
}
```

Vì phạm Nguyên tắc ít kiến thức nhất!
Bạn đang gọi phương thức của một đối tượng
được trả về từ một lệnh gọi khác.

Không vi phạm Nguyên tắc ít kiến thức nhất!
Có vẻ như chúng ta đang phá vỡ nguyên tắc.
Có điều gì thực sự thay đổi kể từ khi chúng ta
chuyển lệnh gọi sang phương thức khác không?



Giải pháp bài tập

Bạn đã thấy cách triển khai bộ điều hợp chuyển đổi Enumeration thành Iterator; bây giờ hãy viết bộ điều hợp chuyển đổi Iterator thành Enumeration.

```

lớp công khai IteratorEnumeration thực hiện Enumeration {
    Trình lặp trình lặp;

    công khai IteratorEnumeration(Iterator iterator) {
        this.iterator = trình lặp;
    }

    boolean công khai.hasMoreElements() { trả về
        iterator.hasNext();
    }

    Đối tượng công khai.nextElement() { return
        iterator.next();
    }
}

```

* WHO DOES WHAT? *

Ghép mỗi mẫu với mục đích của nó:

Mẫu

Ý định

Nguồn trang trí

Chuyển đổi một giao diện sang
giao diện khác

Bộ chuyển đổi

Không thay đổi giao diện như ng
thêm trách nhiệm

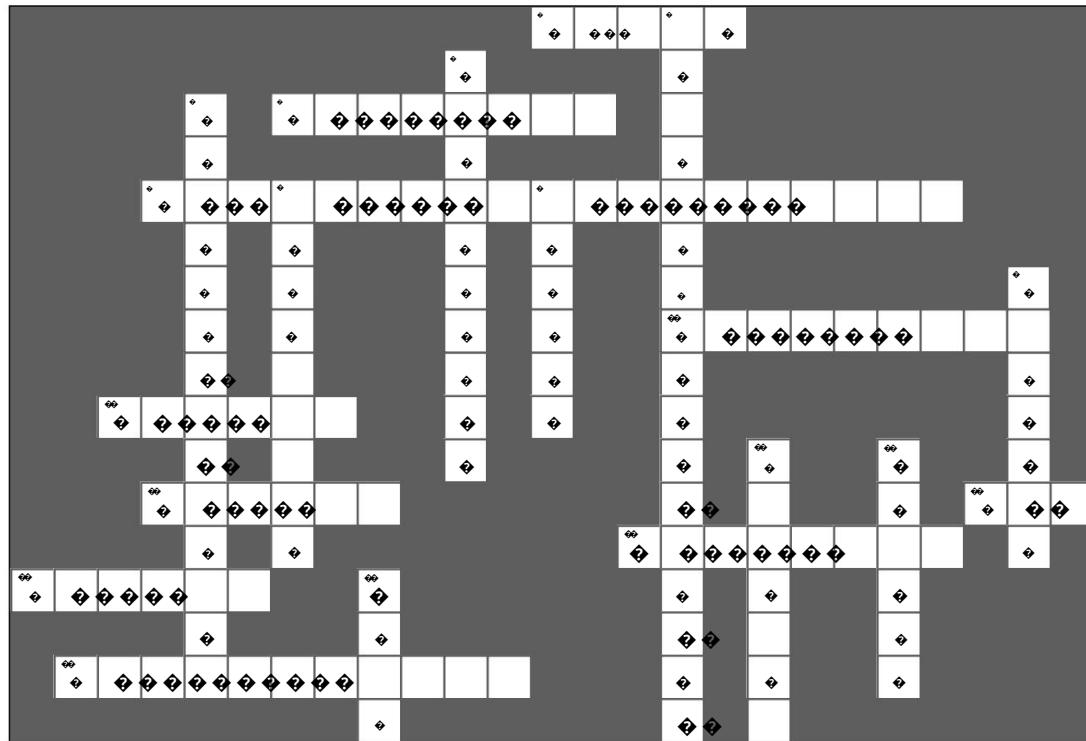
Mặt tiền

Làm cho giao diện đơn giản hơn

giải ô chữ



Giải pháp bài tập



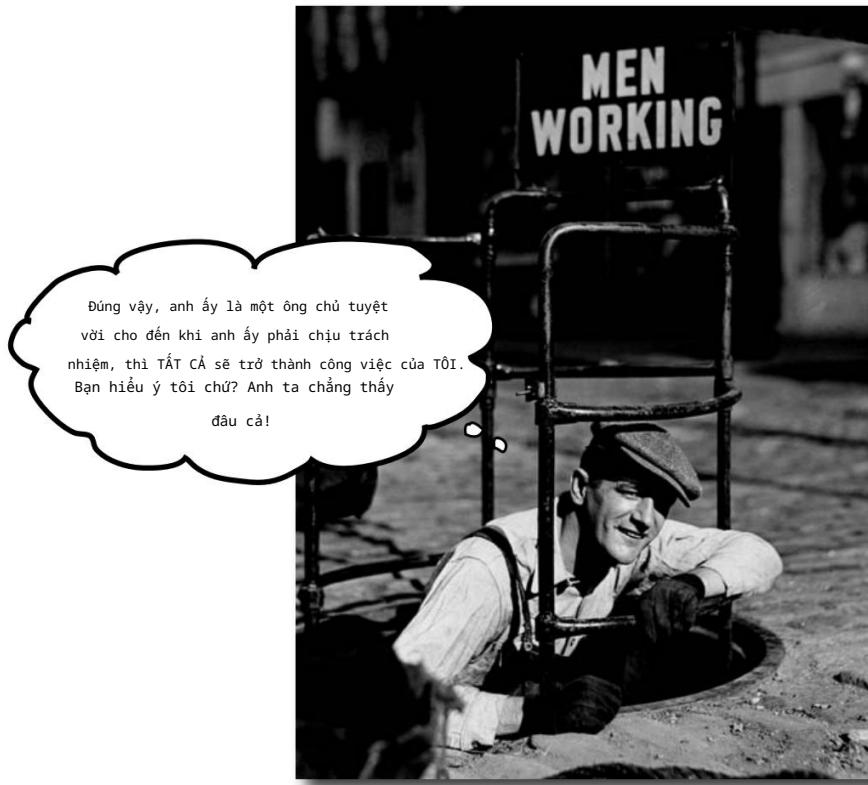
???

????????????????????????????
????? ??????
?? ?? ?????? ????? ????? ?????
????????????????????????????
????????????????????????
????????????????????????
????????????????????????????
????????????????????????
????????????????????????
????????????????????????

???

????????????????????????????
????????????????????????????
?? ?????? ????? ????? ????? ?? ????? ?????
????????????????????????????
????????????????????????????
????????????????????????????
????????????????????????????
????????????????????????????
????????????????????????????
????????????????????????????
????????????????????????????

8 Mẫu Phư ơ ng pháp Mẫu g Đóng gói h Thuật toán g



Chúng tôi đang trong quá trình đóng gói; chúng tôi đã đóng gói việc tạo đối tượng, gọi phư ơ ng

thức, giao diện phức tạp, vịt, pizza... tiếp theo có thể là gì? Chúng tôi sẽ bắt đầu đóng gói

các phần của thuật toán để các lớp con có thể tự mọc nồi vào một phép tính

bắt cứ lúc nào họ muốn. Chúng ta thậm chí sẽ tìm hiểu về một nguyên tắc thiết kế lấy cảm hứng từ

Hollywood.

công thức pha cà phê và trà tự ứng tự nhau

Đã đến lúc cần thêm caffeine

Một số người không thể sống thiếu cà phê; một số người không thể sống thiếu trà. Thành phần phổ biến? Tất nhiên là Caffeine!

Như ng còn hơn thế nữa; trà và cà phê được chế biến theo những cách rất giống nhau. Hãy cùng xem xét:

Sở tay hướng dẫn đào tạo Barista của Starbuzz Coffee

Các nhân viên pha chế! Hãy làm theo chính xác các công thức này khi pha chế đồ uống Starbuzz.

Công thức pha chế cà phê Starbuzz

- (1) Đun sôi một ít nước
- (2) Pha cà phê trong nước sôi
- (3) Đổ cà phê vào cốc
- (4) Thêm đường và sữa

Công thức pha trà Starbuzz

- (1) Đun sôi một ít nước
- (2) Ngâm trà trong nước sôi
- (3) Đổ trà vào tách
- (4) Thêm chanh

Tất cả các công thức đều là bí mật thương mại của Starbuzz Coffee và phải được giữ kín hoàn toàn bảo mật.

Công thức pha cà phê trông rất giống công thức pha trà phải không?

Chuẩn bị một số lớp học pha cà phê và trà (bằng Java)

Hãy cùng chờ i “barista lập trình” và viết mã
để pha chế cà phê và trà.



Đây là cà phê:

Đây là lớp học pha chế cà phê của chúng tôi.

lớp công cộng Cà phê {

```
void chuẩn bị công thức() {
    đun sôi nư óc();
    pha chế cà phê xay();
    đổ vào cốc();
    thêm Đư ờng và Sữa();
}
```

```
công khai void đun sôi nư óc() {
    System.out.println("Nư óc sôi");
}
```

```
công khai void brewCoffeeGrinds() {
    System.out.println("Rót cà phê qua bộ lọc");
}
```

```
công khai void pourInCup() {
    System.out.println("Đổ vào cốc");
}
```

```
công khai void addSugarAndMilk() {
    System.out.println("Thêm đư ờng và sữa");
}
```

}

Sau đây là công thức pha cà phê
của chúng tôi, trích thẳng từ sách hướng dẫn.

Mỗi bước được thực hiện theo phu ơ ng pháp
riêng biệt.

Mỗi phu ơ ng pháp này
thực hiện một bước
của thuật toán. Có
một phu ơ ng pháp đun
sôi nư óc, pha cà
phê, đổ cà phê vào
cốc và thêm đư ờng và sữa.

thực hiện trà

và bây giờ là Trà...

```

lớp công cộng Trà {
    void chuẩn bị công thức() {
        dùn sôi nước();
        Túi trà đúc();
        đổ vào cốc();
        thêmLemon();
    }

    công khai void dùn sôi nước() {
        System.out.println("Nước sôi");
    }

    công khai void steepTeaBag() {
        System.out.println("Đang pha trà");
    }

    công khai void addLemon() {
        System.out.println("Thêm chanh");
    }

    công khai void pourInCup() {
        System.out.println("Đổ vào cốc");
    }
}

```

Điều này trông rất giống với điều chúng ta vừa thực hiện trong Coffee; bước thứ hai và thứ tư thì khác, nhưng về cơ bản thì công thức là giống nhau.



Lưu ý rằng
hai
phương pháp này là
giống hệt
như trong
Coffee! Vậy
nên chúng ta
chắc chắn có
một số mã trùng
lặp ở đây.

Hai phương
pháp này là
chuyên môn hóa để
Trà.



Khi chúng ta có sự trùng
lặp mã, đó là dấu hiệu tốt cho
thấy chúng ta cần dọn dẹp thiết kế. Có
về như ở đây chúng ta nên trao đổi ứng hóa
diễn chung thành một lớp cơ sở vì cà
phê và trà rất giống nhau?



Thiết kế câu đố

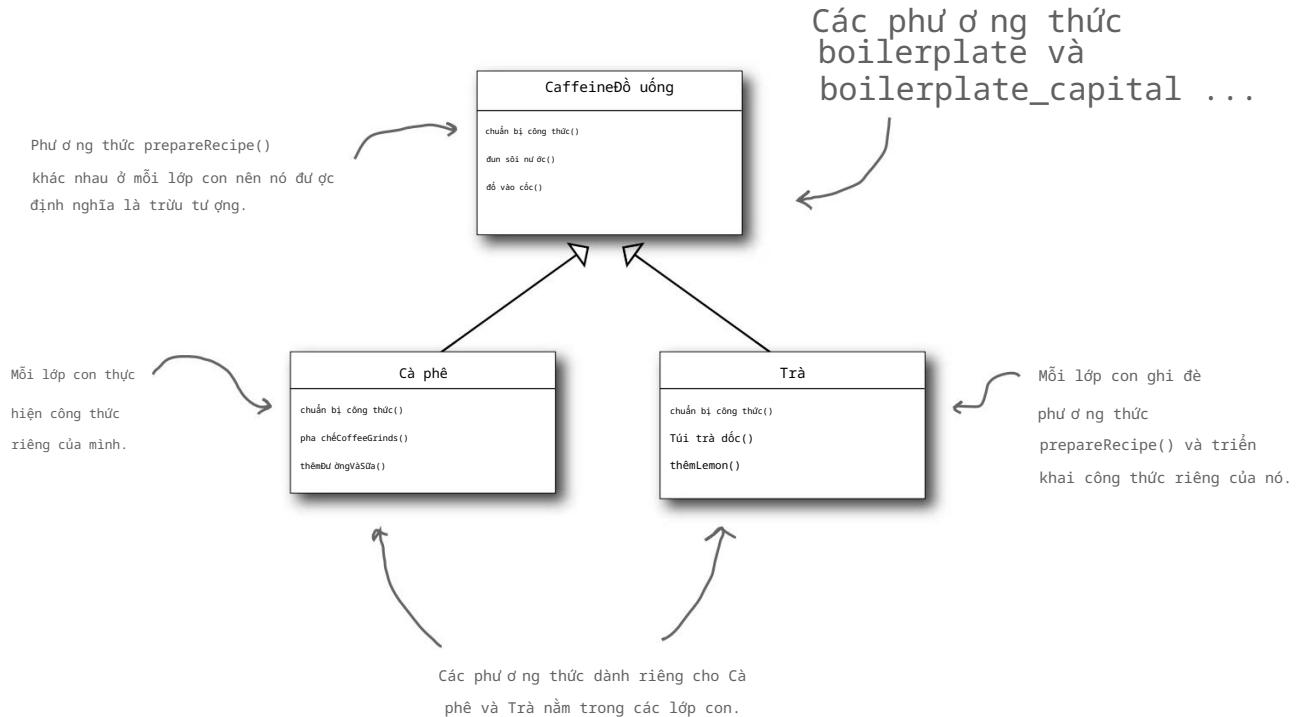
Bạn đã thấy rằng các lớp Coffee and Tea có khá nhiều mã trùng lặp. Hãy xem lại các lớp Coffee and Tea và vẽ sơ đồ lớp cho thấy cách bạn thiết kế lại các lớp để loại bỏ sự trùng lặp:

cắt đầu tiên ở dạng trừu tư ợng

Thư a ông, tôi có thể lấy cà phê hoặc trà của ông đư ợc không?

Có vẻ như chúng ta có một bài tập thiết kế khá đơn giản với lớp Cà phê và Trà.

Bản cắt đầu tiên của bạn có thể trông giống thế này:

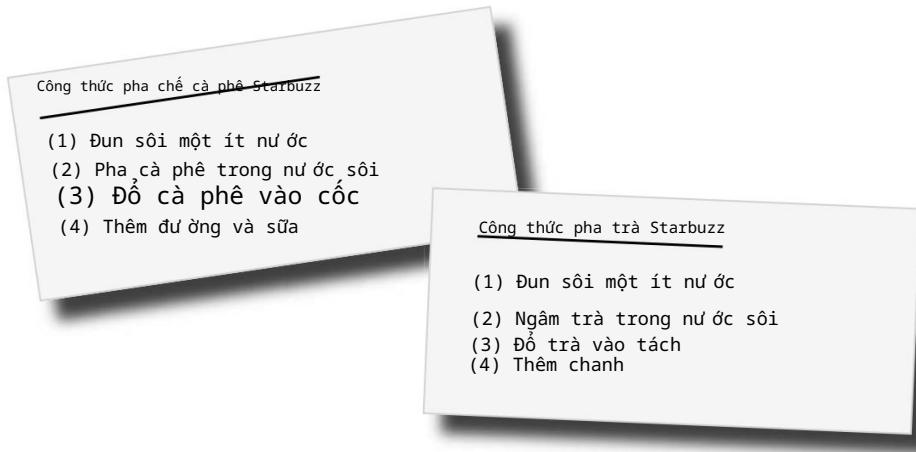


não Apower

Chúng ta đã làm tốt việc thiết kế lại chưa? Hmm, hãy xem lại. Chúng ta có bỏ qua điểm chung nào khác không? Cà phê và Trà có điểm gì giống nhau không?

Đứ a thiέ t kέ tié n xa hơ n...

Vậy cà phê và trà còn có điểm gì chung nữa? Hãy bắt đầu với công thức pha chế.



Lưu ý rằng cả hai công thức đều tuân theo cùng một thuật toán:

- 1 Đun sôi một ít nước.
- 2 Sử dụng nước nóng để chiết xuất cà phê hoặc trà.
- 3 Đổ đồ uống thu được vào cốc.
- 4 Thêm gia vị thích hợp vào đồ uống.

Những điều này không
được trừu tượng hóa,
nhưng chúng giống nhau,
chúng chỉ áp dụng
cho các loại đồ
uống khác nhau.

Hai cái này đã
được trừu tượng hóa
vào lớp cơ sở.

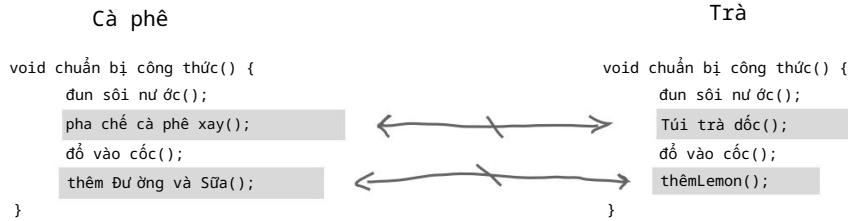
Vậy, chúng ta có thể tìm ra cách trừu tượng hóa `prepareRecipe()` không? Vâng, chúng ta hãy cùng tìm hiểu nhé...

tóm tắt thuật toán

Tóm tắt prepareRecipe()

Chúng ta hãy cùng tìm hiểu cách trừu tượng hóa prepareRecipe() từ mỗi lớp con (tức là các lớp Coffee và Tea)...

- Vấn đề đầu tiên chúng ta gặp phải là Coffee sử dụng các phương thức brewCoffeeGrinds() và addSugarAndMilk() trong khi Tea sử dụng các phương thức steepTeaBag() và addLemon().



Hãy cùng suy nghĩ về điều này: ngâm và pha không khác nhau nhiều lắm; chúng khá giống nhau.

Vậy hãy tạo một tên phương thức mới, chẳng hạn như brew(), và chúng ta sẽ sử dụng cùng một tên cho dù chúng ta đang pha cà phê hay pha trà.

Tự ngưng tự vậy, việc thêm đường và sữa cũng khá giống với việc thêm chanh: cả hai đều là thêm gia vị vào đồ uống. Chúng ta cũng hãy tạo một tên phương thức mới, addCondiments(), để xử lý việc này. Vì vậy, phương thức prepareRecipe() mới của chúng ta sẽ trông như thế này:

```

void chuẩn bị công thức() {
    đun sôi nước();
    pha();
    đổ vào cốc();
    thêm Gia vị();
}

```

- Bây giờ chúng ta có phương thức prepareRecipe() mới, nhưng chúng ta cần phải đưa nó vào trong mã. Để thực hiện điều này, chúng ta sẽ bắt đầu với siêu lớp CaffeineBeverage:

```

        CaffeineBeverage mang tính trừu tượng,
        giống như thiết kế lớp học.

lớp trừu tượng công khai CaffeineBeverage {

    void cuối cùng prepareRecipe() {
        dùn sôi nước();
        pha();
        đổ vào cốc();
        thêm Gia vị();
    }

    trừu tượng void brew();

    tóm tắt void addCondiments();

    void dùn sôi nước() {
        System.out.println("Nước sôi");
    }

    void pourInCup() {
        System.out.println("Đổ vào cốc");
    }
}

```

Bây giờ, phu ơ ng thức prepareRecipe() sẽ được sử dụng để pha cả Trà và Cà phê. prepareRecipe() được khai báo là final vì chúng ta không muốn các lớp con của mình có thể ghi đè phu ơ ng thức này và thay đổi công thức! Chúng ta đã khái quát hóa các bước 2 và 4 để pha chế đồ uống và addCondiments().

Vì Coffee và Tea xử lý các phu ơ ng thức này theo những cách khác nhau nên chúng sẽ phải được khai báo là trừu tượng. Hãy để các lớp con lo về vấn đề đó!

Hãy nhớ rằng chúng ta đã chuyển những thứ này vào lớp CaffeineBeverage (trở lại sơ đồ lớp của chúng ta).

- 3 Cuối cùng chúng ta cần xử lý các lớp Cà phê và Trà. Bây giờ họ dựa vào CaffeineBeverage để xử lý công thức, vì vậy họ chỉ cần xử lý pha chế và gia vị:

```

lớp công khai Trà mở rộng CaffeineBeverage {
    công khai void brew() {
        System.out.println("Đang pha trà");
    }
    công khai void addCondiments() {
        System.out.println("Thêm chanh");
    }
}

```

Theo thiết kế của chúng tôi, Tea and Coffee hiện mở rộng CaffeineBeverage.

Tea cần định nghĩa brew() và addCondiments() – hai phu ơ ng thức trừu tượng từ Beverage.

```

lớp công khai Coffee mở rộng CaffeineBeverage {
    công khai void brew() {
        System.out.println("Cà phê nhỏ giọt qua bộ lọc");
    }
    công khai void addCondiments() {
        System.out.println("Thêm đường và sữa");
    }
}

```

Tương tự với Cà phê, ngoại trừ việc Cà phê bao gồm cà phê, đường và sữa thay vì trà túi lọc và chanh.

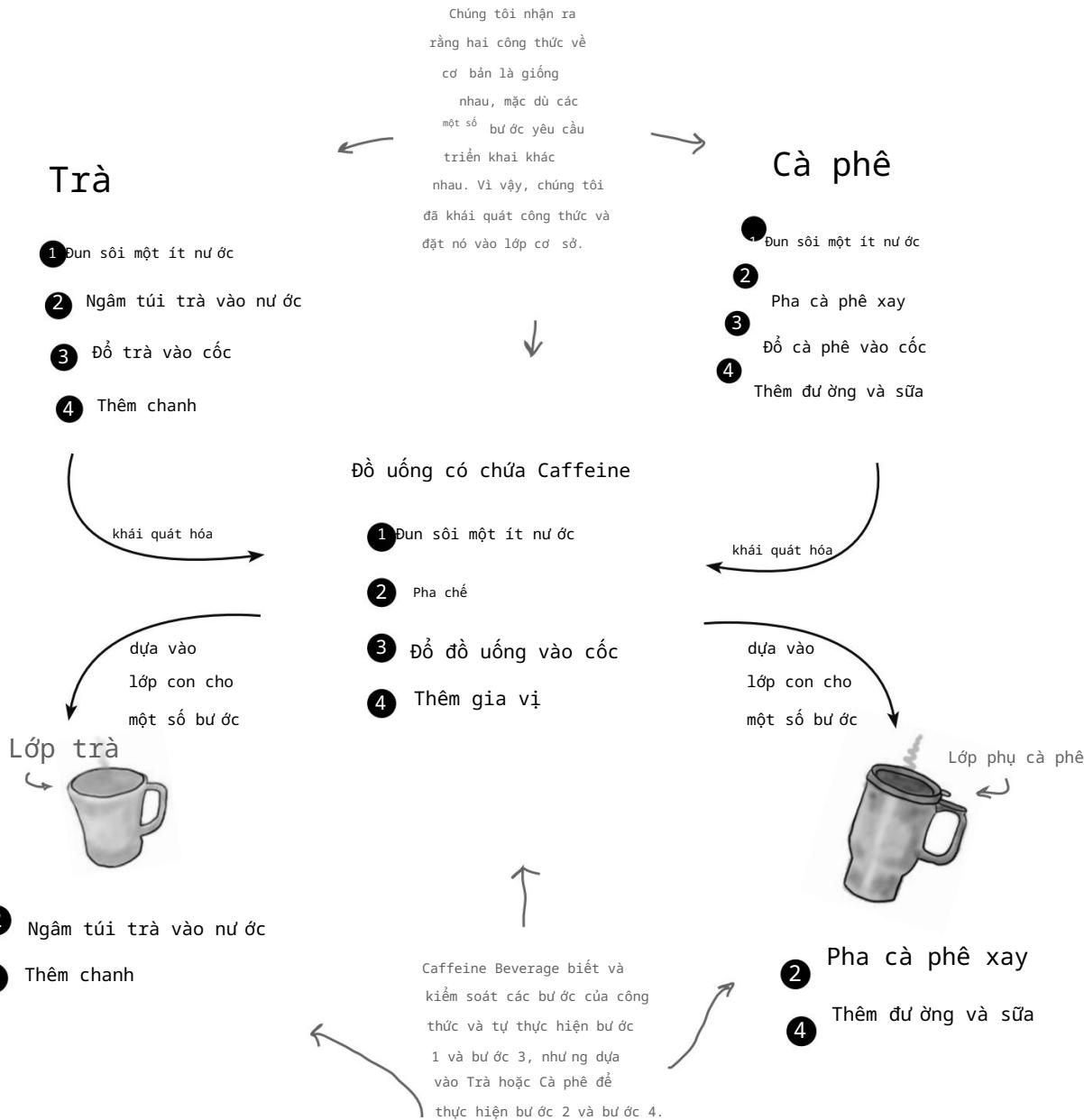
sơ đồ lớp cho đồ uống có chứa caffeine



Chuốt bút chì của bạn

Vẽ sơ đồ lớp mới sau khi chúng ta đã di chuyển phần triển khai `prepareRecipe()` vào lớp `CaffeineBeverage`:

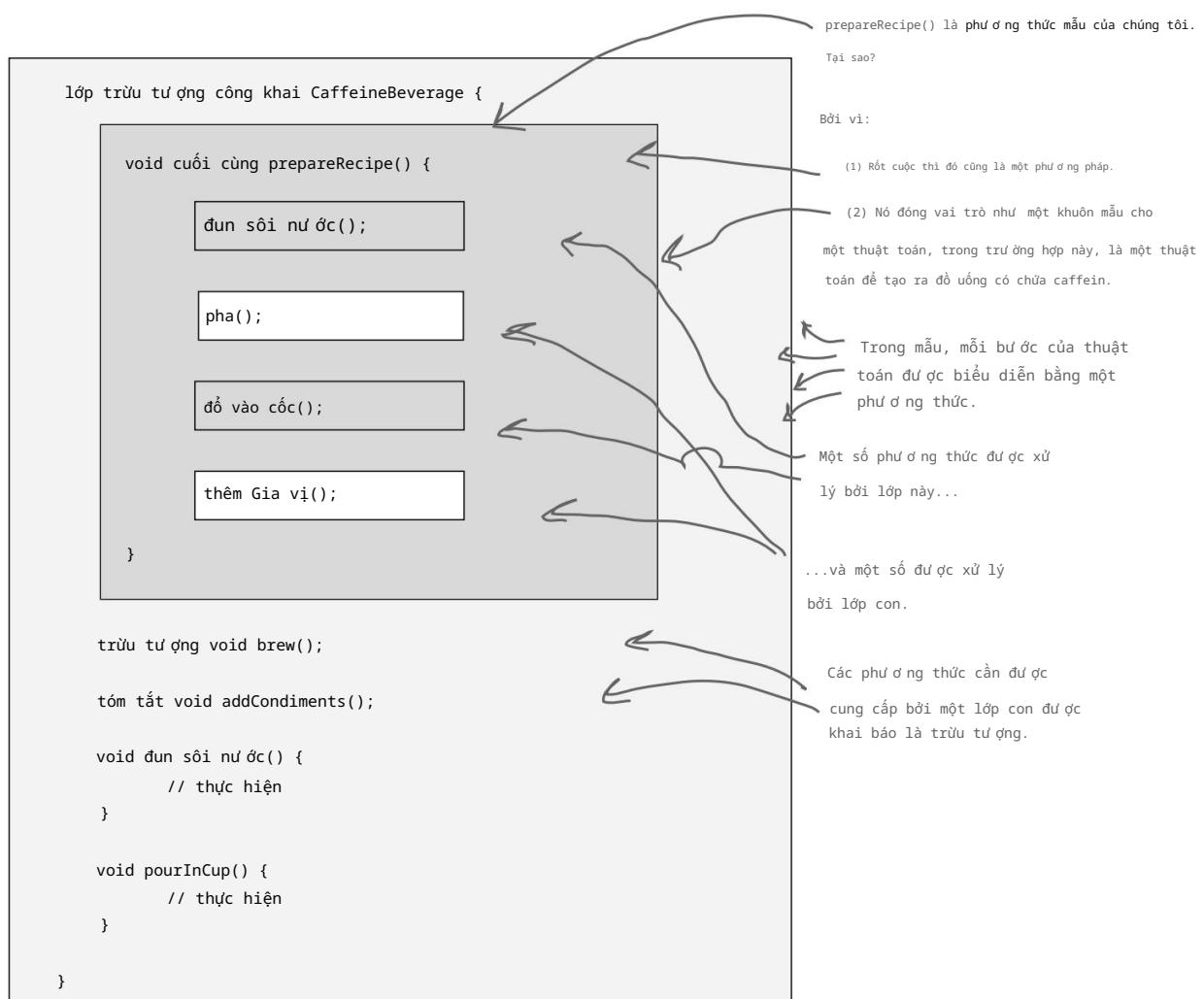
Chúng ta đã làm gì?



đáp ứng mẫu phu ơ ng pháp mẫu

Gặp gỡ Phu ơ ng pháp Mẫu

Về cơ bản, chúng ta vừa triển khai Mẫu phu ơ ng thức mẫu. Đó là gì? Hãy xem cấu trúc của lớp CaffeineBeverage; nó chứa "phu ơ ng thức mẫu" thực tế:



Phu ơ ng pháp mẫu định nghĩa các bước của thuật toán và cho phép các lớp con cung cấp triển khai cho một hoặc nhiều bước.

Chúng ta hãy pha chút trà nhé...

Hậu trư ờng



Hãy cùng thực hiện từng bước pha trà và theo dõi cách thức hoạt động của phư ơ ng thức mẫu. Bạn sẽ thấy rằng phư ơ ng thức mẫu kiểm soát thuật toán; tại một số điểm nhất định trong thuật toán, nó cho phép lớp con cung cấp việc triển khai các bước...

1

Đầu tiên chúng ta cần một đối tượng Tea...

```
Trà myTea = Trà mới();
```

```
đun sôi nước();
pha();
đổ vào cốc();
thêm Gia vị();
```

2

Sau đó chúng ta gọi phư ơ ng thức mẫu:

```
myTea.prepareRecipe();
```

tuân theo thuật toán để tạo ra đồ uống có chứa caffeine...

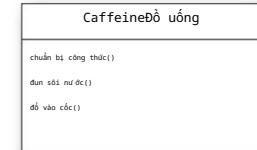
Phư ơ ng thức prepareRecipe() kiểm soát thuật toán, không ai có thể thay đổi điều này và nó dựa vào các lớp con để cung cấp một phần hoặc toàn bộ nội dung triển khai.

3

Đầu tiên chúng ta đun sôi nước:

```
đun sôi nước();
```

điều này xảy ra ở CaffeineBeverage.



4

Tiếp theo chúng ta cần pha trà, việc mà chỉ có lớp con mới biết cách thực hiện:

```
pha();
```

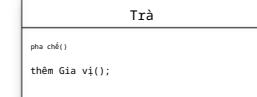
Bây giờ chúng ta rót trà vào cốc; điều này cũng giống với tất cả các loại đồ uống khác nên nó cũng xảy ra trong CaffeineBeverage:

```
đổ vào cốc();
```

5

Cuối cùng, chúng ta thêm các loại gia vị đặc trưng cho từng loại đồ uống, do đó lớp con sẽ thực hiện điều này:

```
thêm Gia vị();
```



Phu ơ ng pháp mău mang lại cho chúng ta điều gì ?

Phu ơ ng pháp Mău mang lại cho chúng ta điều gì?



Triển khai Trà & Cà phê không đủ
năng lực



CaffeineBeverage mới, sành điệu
đư ợc cung cấp bởi Template Method

Cà phê và trà là những ngư ời điều hành; họ kiểm
soát thuật toán.

Lớp CaffeineBeverage chạy chương trình;
nó có thuật toán và bảo vệ thuật toán đó.

Mã đư ợc sao chép trên Coffee and Tea.

Lớp CaffeineBeverage tối đa hóa khả
năng tái sử dụng giữa các lớp con.

Việc thay đổi mã cho thuật toán đòi hỏi
phải mở các lớp con và thực hiện nhiều thay
đổi.

Thuật toán nằm ở một nơi và việc thay đổi mã
chỉ cần đư ợc thực hiện ở đó.

Các lớp học đư ợc tổ chức theo một cấu trúc
đòi hỏi nhiều công sức để thêm một loại đồ
uống có chứa caffeine mới.

Phiên bản Phu ơ ng pháp Mău cung cấp một khuôn khổ
mà các loại đồ uống có chứa caffeine
khác có thể đư ợc đưa vào. Các loại đồ uống có
chứa caffeine mới chỉ cần triển khai một
vài phu ơ ng pháp.

Kiến thức về thuật toán và cách triển khai
thuật toán đư ợc phân bổ trên nhiều lớp.

Lớp CaffeineBeverage tập trung kiến
thức về thuật toán và dựa vào các lớp con
để cung cấp các triển khai hoàn chỉnh.

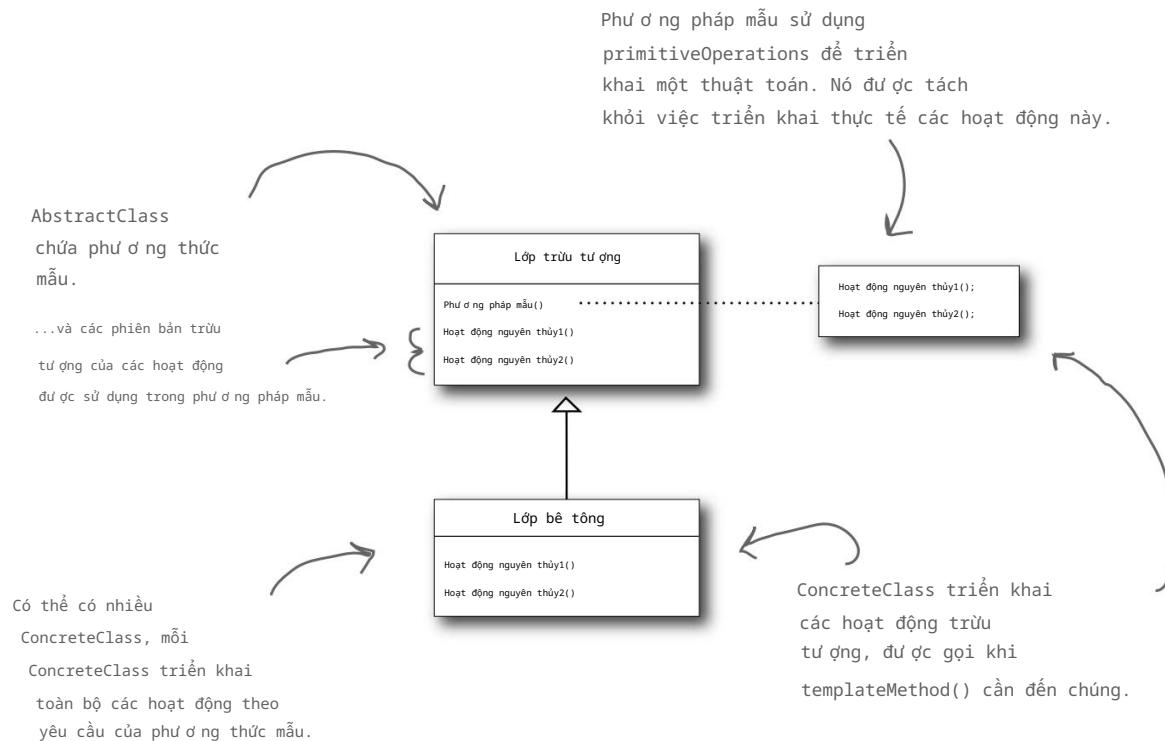
Mẫu Phư ơ ng pháp Mẫu đư ợc xác định

Bạn đã thấy cách thức hoạt động của Mẫu Phư ơ ng pháp trong ví dụ Trà và Cà phê của chúng tôi; bây giờ, hãy xem định nghĩa chính thức và nắm rõ mọi chi tiết:

Mẫu phư ơ ng pháp mẫu định nghĩa bộ khung của một thuật toán trong một phư ơ ng pháp, chuyển một số bù ớc cho các lớp con. Phư ơ ng pháp mẫu cho phép các lớp con định nghĩa lại một số bù ớc nhất định của một thuật toán mà không thay đổi cấu trúc của thuật toán.

Mẫu này là về việc tạo ra một khuôn mẫu cho một thuật toán. Khuôn mẫu là gì? Như bạn đã thấy, nó chỉ là một phư ơ ng pháp; cụ thể hơn, nó là một phư ơ ng pháp định nghĩa một thuật toán như một tập hợp các bù ớc. Một hoặc nhiều bù ớc trong số này đư ợc định nghĩa là trừu tư ợng và đư ợc triển khai bởi một lớp con. Điều này đảm bảo cấu trúc của thuật toán không thay đổi, trong khi các lớp con cung cấp một số phần của quá trình triển khai.

Hãy cùng xem sơ đồ lớp:



mẫu phư ơ ng pháp mẫu cận cảnh



Mã Gần

Chúng ta hãy xem xét kỹ hơn cách AbstractClass được định nghĩa, bao gồm phư ơ ng thức mẫu và các hoạt động nguyên thủy.

Ở đây chúng ta có lớp trừu tư ợng;
lớp này được khai báo là trừu tư ợng
và có nghĩa là được phân lớp theo các
lớp cung cấp triển khai các hoạt động.

```
lớp trừu tư ợng AbstractClass {
    void templateMethod cuối cùng() {
        Hoạt động nguyên thủy1();
        Hoạt động nguyên thủy2();
        Hoạt động cụ thể();
    }

    trừu tư ợng void primitiveOperation1();
    trừu tư Ợng void primitiveOperation2();

    void concreteOperation() {
        // thực hiện ở đây
    }
}
```

Chúng ta cũng có một hoạt động cụ thể được định nghĩa trong lớp trừu tư ợng. Thêm thông tin về các loại phư ơ ng pháp này sau một chút...

Đây là phư ơ ng thức mẫu. Phư ơ ng thức này được khai báo là final để ngăn các lớp con làm lại trình tự các bước trong thuật toán.

Phư ơ ng pháp mẫu xác định trình tự các bước, mỗi bước được biểu diễn bằng một phư ơ ng pháp.

Trong ví dụ này, hai trong số các hoạt động nguyên thủy phải được thực hiện bởi các lớp con bê tông.



Mã Cách Gần Hơ n

Bây giờ chúng ta sẽ xem xét kỹ hơn các loại phu' ơ ng thức có thể có trong lớp trừu tư ợng:

Chúng tôi đã thay đổi
templateMethod() để bao gồm một
lệnh gọi phu' ơ ng thức mới.

```
lớp trừu tư ợng AbstractClass {

    void templateMethod cuối cùng() {
        Hoạt động nguyên thủy1();
        Hoạt động nguyên thủy2();
        Hoạt động cụ thể();
        cái móc();
    }

    trừu tư ợng void primitiveOperation1();

    trừu tư ợng void primitiveOperation2();

    void cuối cùng concreteOperation() {
        // thực hiện ở đây
    }

    void hook() {}
}
```

Chúng ta vẫn có các phu' ơ ng
thức nguyên thủy;
chúng trừu tư ợng và được
triển khai bởi các lớp con cụ thể.

Một hoạt động cụ thể được định nghĩa trong lớp
trừu tư ợng. Hoạt động này được khai báo là
final để các lớp con không thể ghi đè lên nó.
Nó có thể được sử dụng trực tiếp trong phu' ơ ng
thức mẫu hoặc được sử dụng bởi các lớp con.

Một phu' ơ ng pháp cụ thể, như ng
không có tác dụng gì!

Chúng ta cũng có thể có các phu' ơ ng thức cụ thể không làm gì theo
mặc định; chúng ta gọi chúng là "hooks". Các lớp con có thể tự do ghi
để các phu' ơ ng thức này nhưng không nhất thiết phải làm vậy. Chúng
ta sẽ xem chúng hữu ích như thế nào ở trang tiếp theo.

thực hiện một cái móc

Mê mẩn phu ơ ng pháp mẫu...

Hook là một phu ơ ng thức được khai báo trong lớp trừu tư ơng, như ng chỉ được cung cấp một triển khai rỗng hoặc mặc định. Điều này cung cấp cho các lớp con khả năng "móc vào" thuật toán tại nhiều diễm khác nhau, nếu chúng muốn; một lớp con cũng có thể bỏ qua hook.

Có một số cách sử dụng hook; chúng ta hãy xem xét một cách ngay bây giờ. Chúng ta sẽ nói về một số cách sử dụng khác sau:



```

lớp trừu tư ơng công khai CaffeineBeverageWithHook {

    void cuối cùng prepareRecipe() {
        đun sôi nư ớc();
        pha();
        đổ vào cốc();
        nếu (customerWantsCondiments()) {
            thêm Gia vị();
        }
    }

    trùu tư ơng void brew();

    tóm tắt void addCondiments();

    void đun sôi nư ớc() {
        System.out.println("Nú ớc sôi");
    }

    void pourInCup() {
        System.out.println("Đổ vào cốc");
    }

    boolean khách hàng muónCondiments() {
        trả về giá trị đúng;
    }
}

```

Chúng tôi đã thêm một câu lệnh điều kiện nhỏ dựa trên thành công của phu ơ ng thức cụ thể, customerWantsCondiments(). Nếu khách hàng MUỐN gia vị, thì khi đó chúng ta mới gọi addCondiments().

Ở đây chúng tôi đã định nghĩa một phu ơ ng thức với triển khai mặc định (hầu hết) trống. Phu ơ ng thức này chỉ trả về true và không làm gì khác.

Đây là một diễm hấp dẫn vì lớp con có thể ghi đè phu ơ ng thức này như ng không nhất thiết phải làm vậy.

Sử dụng móc

Để sử dụng hook, chúng ta ghi đè nó trong lớp con của mình. Ở đây, hook kiểm soát việc CaffeineBeverage có đánh giá một phần nhất định của thuật toán hay không; nghĩa là, liệu nó có thêm gia vị vào đồ uống hay không.

Làm sao chúng ta biết được khách hàng có muốn loại gia vị đó không? Chỉ cần hỏi!

```

lớp công khai CoffeeWithHook mở rộng CaffeineBeverage {

    public void brew()
        { System.out.println("Cà phê nhỏ giọt qua bộ lọc");
    }

    public void addCondiments()
        { System.out.println("Thêm đường và sữa");
    }

    công khai boolean customerWantsCondiments() {
        Chuỗi câu trả lời = getUserInput();

        nếu (answer.toLowerCase().startsWith("y"))
            trả về true; } else
        { trả về
            false;
        }
    }

    riêng tư String getUserInput() {
        Câu trả lời chuỗi = null;

        System.out.print("Bạn có muốn thêm sữa và đường vào cà phê không (y/n)? ");

        BufferedReader trong = new BufferedReader(new InputStreamReader(System.in));
        thử { câu trả lời = in.readLine(); } bắt
        (IOException ioe) {

            System.err.println("Lỗi IO khi cố gắng đọc câu trả lời của bạn");

            } nếu (câu trả lời == null) { trả
                về "không";
            }
            trả lời lại;
    }
}

```

Đây là nơi bạn ghi đè hook và cung cấp chức năng của riêng bạn.

Nhận thông tin đầu vào của người dùng về quyết định gia vị và trả về giá trị đúng hoặc sai, tùy thuộc vào thông tin đầu vào.

Đoạn mã này hỏi người dùng xem họ có muốn sữa và đường không và lấy thông tin đầu vào từ dòng lệnh.

lái thử

Hãy chạy TestDrive

Đư ợc rồi, nư ớc đang sôi... Đây là mã kiểm tra nơi chúng ta tạo ra một tách trà nóng và một tách cà phê nóng

```
lớp công khai BeverageTestDrive { công
    khai tĩnh void main(String[] args) {

        TeaWithHook teaHook = mới TeaWithHook();
        CoffeeWithHook coffeeHook = new CoffeeWithHook();

        System.out.println("\nPha trà...");
        teaHook.prepareRecipe();

        System.out.println("\nPha cà phê...");
        coffeeHook.prepareRecipe();
    }
}
```

← Pha trà.
← Một tách cà phê.
← ← Và gọi prepareRecipe() trên cả hai!

Và chúng ta hãy thử xem...

```
Cửa sổ chỉnh sửa tệp Trợ giúp send-more-honesttea
%java Đò uốngTestDrive

Đang pha trà...
Đun sôi nư ớc Pha
trà Đỗ vào cốc Bạn có
muốn thêm chanh vào trà
không (y/n)? y Thêm chanh

Đang pha cà phê...
Nư ớc sôi Nhỏ giọt Cà
phê qua bộ lọc Đỗ vào cốc Bạn có muốn thêm sữa và
đu ờng vào cà phê của mình
không (có/không)? n %
```

Một tách trà nóng hổi, và tất nhiên là chúng ta muôn có thêm chanh!

Và một tách cà phê nóng hổi, như ng chúng tôi sẽ bỏ qua các loại gia vị làm tăng vòng eo.



Bạn biết không? Chúng tôi đồng ý với bạn. Nhưng
bạn phải thừa nhận rằng trước khi bạn nghĩ đến điều
đó, đó là một ví dụ khá thú vị về cách sử dụng
hook để kiểm soát có điều kiện luồng của thuật toán
trong lớp trừu tượng. Đúng không?

Chúng tôi chắc chắn rằng bạn có thể nghĩ ra nhiều
tình huống thực tế hơn mà bạn có thể sử dụng
phu ơ ng thức mẫu và móc trong mã của riêng mình.

không có Những câu hỏi ngắn

Q: Khi tôi đang tạo một mẫu
phu ơ ng pháp, làm sao tôi biết khi nào sử
dụng phu ơ ng pháp trừu tượng và khi nào sử dụng
móc?

A: Sử dụng các phu ơ ng pháp trừu tượng khi bạn
lớp con PHẢI cung cấp một triển khai của
phu ơ ng pháp hoặc móc trong thuật toán.
Sử dụng hook khi phần đó của thuật toán là
tùy chọn. Với hook, một lớp con có thể chọn
triển khai hook đó, nhưng không nhất thiết phải
làm vậy.

H: Móc câu thực sự có ý nghĩa gì?
được sử dụng cho mục đích gì?

A: Có một vài cách sử dụng móc. Như
chúng tôi vừa nói, một hook có thể cung cấp một
cách để một lớp con thực hiện một phần tùy chọn

của một thuật toán, hoặc nếu nó không quan
trọng đối với việc triển khai của lớp con, nó
có thể bỏ qua nó. Một cách sử dụng khác là
cung cấp cho lớp con cơ hội phản ứng với
một số bước trong phu ơ ng thức mẫu sắp xảy
ra hoặc vừa xảy ra. Ví dụ, một phu ơ ng thức
hook như justReOrderedList() cho phép lớp con
thực hiện một số hoạt động (chẳng hạn như hiển
thị lại biểu diễn trên màn hình) sau khi danh
sách bên trong được sắp xếp lại. Như bạn đã
thấy, một hook cũng có thể cung cấp cho lớp
con khả năng đưa ra quyết định cho lớp trừu
tương.

Q: Một lớp con có phải
triển khai tất cả các phu ơ ng thức trừu tượng
trong AbstractClass?

A: Có, mỗi lớp con cụ thể xác định
toute bộ tập hợp các phu ơ ng pháp trừu tượng và

cung cấp một triển khai hoàn chỉnh các
bước chia sẻ xác định của thuật toán
phu ơ ng pháp mẫu.

H: Có vẻ như tôi nên giữ
số lượng phu ơ ng thức trừu tượng ít,
nếu không sẽ rất khó khăn để triển khai chúng
trong lớp con.

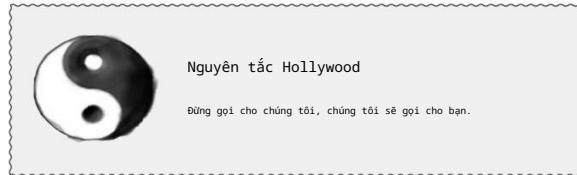
A: Đó là một điều tốt để giữ trong
lưu ý khi bạn viết phu ơ ng thức mẫu.
Đôi khi điều này có thể thực hiện được bằng cách không
làm cho các bước trong thuật toán của bạn quá chi tiết.
Nhưng rõ ràng đây là một sự đánh đổi: càng ít chi tiết
thì tinh linh hoạt càng kém.

Ngoài ra, hãy nhớ rằng một số bước sẽ là
tùy chọn; do đó, bạn có thể triển khai chúng
như các hook thay vì các phu ơ ng thức trừu
tương, giúp giảm bớt gánh nặng cho các
lớp con của lớp trừu tượng.

nguyên tắc hollywood

Nguyên tắc Hollywood

Chúng tôi có một nguyên tắc thiết kế khác dành cho bạn; nó được gọi là Nguyên tắc của Hollywood:



Để nhớ phải không? Như ng nó liên quan gì đến thiết kế OO?

Nguyên lý Hollywood cung cấp cho chúng ta một cách để ngăn chặn "sự phụ thuộc". Sự phụ thuộc xảy ra khi bạn có các thành phần cấp cao phụ thuộc vào các thành phần cấp thấp, phụ thuộc vào các thành phần cấp cao, phụ thuộc vào các thành phần ngang, phụ thuộc vào các thành phần cấp thấp, v.v.

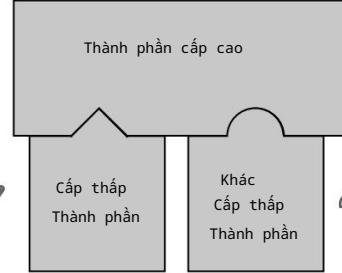
Khi sự mục nát bắt đầu, không ai có thể dễ dàng hiểu được cách một hệ thống đưa ra quyết định.

Với Nguyên tắc Hollywood, chúng tôi cho phép các thành phần cấp thấp tự móc nối vào hệ thống, như ng các thành phần cấp cao sẽ quyết định khi nào cần và như thế nào. Nói cách khác, các thành phần cấp cao sẽ cung cấp cho các thành phần cấp thấp cách xử lý "đừng gọi cho chúng tôi, chúng tôi sẽ gọi cho bạn".

Bạn đã từng nghe tôi nói rồi, và tôi sẽ nói lại lần nữa:
đừng gọi cho tôi, tôi sẽ gọi cho bạn!



Các thành phần cấp thấp
có thể tham gia vào quá
trình tính toán.



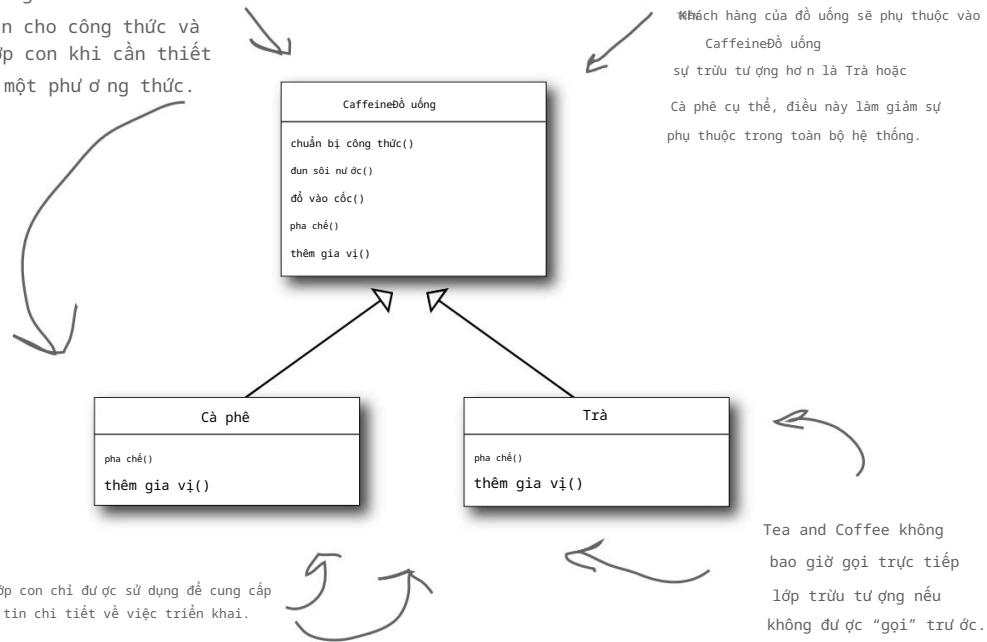
Như ng các thành phần
cấp cao kiểm soát
khi nào và như thế nào.

Một thành phần cấp thấp không
bao giờ gọi trực tiếp một thành
phần cấp cao.

Nguyên tắc Hollywood và Phu ơ ng pháp mẫu

Mối liên hệ giữa Nguyên lý Hollywood và Mẫu phu ơ ng pháp mẫu có lẽ khá rõ ràng: khi chúng ta thiết kế bằng Mẫu phu ơ ng pháp mẫu, chúng ta đang nói với các lớp con rằng "đừng gọi cho chúng tôi, chúng tôi sẽ gọi cho bạn". Làm thế nào? Hãy cùng xem lại thiết kế CaffeineBeverage của chúng ta:

CaffeineBeverage là thành phần cấp cao của chúng tôi. Nó kiểm soát thuật toán cho công thức và chỉ gọi các lớp con khi cần thiết để triển khai một phu ơ ng thức.



não Apower

Những mô hình nào khác sử dụng Nguyên lý Hollywood?

ai làm gì

không có Những câu hỏi ngắn

H: Nguyên tắc Hollywood là gì?
liên quan đến Nguyên lý đảo ngược phụ
thuộc mà chúng ta đã học ở một vài chương
trình không?

A: Sự đảo ngược phụ thuộc
Nguyên tắc dạy chúng ta tránh sử dụng các
lớp cụ thể và thay vào đó làm việc nhiều
nhất có thể với các lớp trừu tượng. Nguyên
tắc Hollywood là một kỹ thuật xây dựng
các khuôn khổ hoặc thành phần để các thành phần
cấp thấp hơn có thể đưa ra mốc nối

vào tính toán, nhưng không tạo ra sự
phụ thuộc giữa các thành phần cấp
thấp hơn và các lớp cấp cao hơn. Vì
vậy, cả hai đều có mục tiêu tách rời,
như ng Nguyên tắc đảo ngược phụ thuộc
đưa ra một tuyên bố mạnh mẽ và chung
hơn nhiều về cách tránh phụ thuộc trong thiết kế không thực sự. Thực tế, một mức độ thấp

Nguyên lý Hollywood cung cấp cho
chúng ta một kỹ thuật để tạo ra các thiết
kế cho phép các cấu trúc cấp thấp tuơng
tác với nhau trong khi ngăn các lớp khác trở
nên quá phụ thuộc vào chúng.

Q: Là một thành phần cấp thấp
không được phép gọi phuơng thức
trong thành phần cấp cao hơn?

Không. Thực tế, một mức độ thấp
thành phần thường sẽ kết thúc bằng việc gọi một
phuơng thức được định nghĩa ở trên nó trong hệ
thống phân cấp kể thừa hoàn toàn thông qua kế
thừa. Nhưng chúng ta muốn tránh tạo ra các phụ
thuộc vòng tròn rõ ràng giữa thành phần cấp
thấp và thành phần cấp cao.



Ghép mỗi mẫu với mô tả của nó:

Mẫu

Sự miêu tả

Phuơng pháp mẫu

Đóng gói các hành vi có thể hoán
đổi cho nhau và sử dụng sự phân quyền
để quyết định hành vi nào sẽ sử dụng

Chiến lược

Các lớp con quyết định cách
thực hiện các bước trong một
thuật toán

Phuơng pháp nhà máy

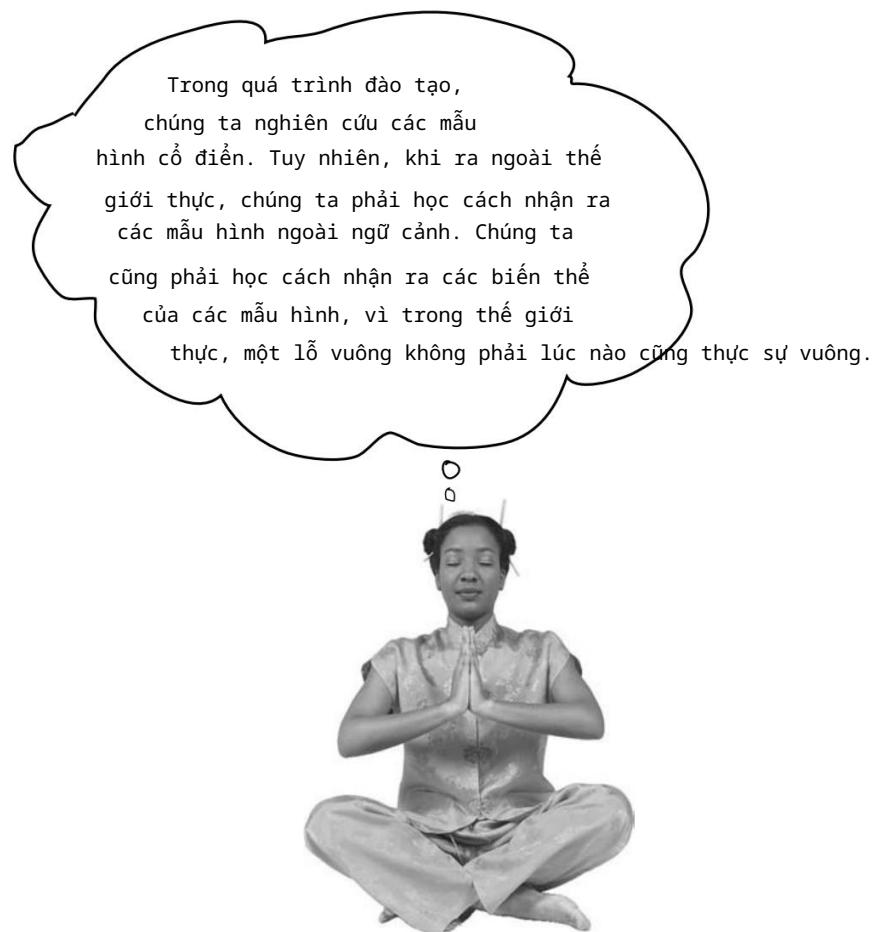
Các lớp con quyết định
lớp cụ thể nào sẽ được tạo

Phư ơ ng pháp mẫu trong tự nhiên

Mẫu phư ơ ng pháp mẫu là một mẫu rất phổ biến và bạn sẽ tìm thấy rất nhiều mẫu trong tự nhiên. Tuy nhiên, bạn phải có con mắt tinh tư ờng vì có nhiều cách triển khai các phư ơ ng pháp mẫu không giống với thiết kế mẫu trong sách giáo khoa.

Mẫu này xuất hiện thường xuyên vì nó là một công cụ thiết kế tuyệt vời để tạo ra các khuôn khổ, trong đó khuôn khổ kiểm soát cách thực hiện một việc gì đó, như ng cho phép bạn (người sử dụng khuôn khổ) tự chỉ định các chi tiết của riêng bạn về những gì thực sự xảy ra ở mỗi bước của thuật toán khuôn khổ.

Chúng ta hãy cùng khám phá một vài cách sử dụng trong thực tế (hay đúng hơn là trong Java API)...



sắp xếp theo phu ơ ng pháp mău

Sắp xếp theo phu ơ ng pháp mău

Chúng ta thư ờng cần phải làm gì với mảng?

Hay phân loại chúng!

Nhận ra điều đó, các nhà thiết kế của lớp Java Arrays đã cung cấp cho chúng ta một phu ơ ng pháp mău tiện dụng để sắp xếp. Hãy cùng xem phu ơ ng pháp này hoạt động như thế nào:

Trên thực tế, chúng ta có hai phu ơ ng pháp ở đây và chúng hoạt động cùng nhau để cung cấp chức năng sắp xếp.



Chúng tôi đã lư ợc bỏ đoạn mã này một chút để dễ giải thích hơn. Nếu bạn muốn xem toàn bộ, hãy lấy mã nguồn từ Sun và kiểm tra nhé...

Phu ơ ng pháp đầu tiên, sort(), chỉ là một phu ơ ng pháp trợ giúp tạo một bản sao của mảng và truyền nó như mảng đích cho phu ơ ng pháp mergeSort(). Nó cũng truyền dọc theo chiều dài của mảng và yêu cầu sắp xếp bắt đầu từ phần tử đầu tiên.

```
public static void sort(Object[] a) {
    Đối tượng aux[] = (Object[])a.clone();
    mergeSort(aux, a, 0, a.length, 0);
}

private static void mergeSort(Object src[], Object dest[],
    int thấp, int cao, int tắt)
{
    đối với (int i=thấp; i<cao; i++){
        đối với (int j=i; j>thấp &&
            ((Có thể so sánh)dest[j-1]).compareTo((Có thể so sánh)dest[j])>0; j--)
        {
            hoán đổi(đích, j, j-1);
        }
    }
    trả lại;
}
```

Phu ơ ng thức mergeSort() chứa thuật toán sắp xếp và dựa vào việc triển khai phu ơ ng thức compareTo() để hoàn thiện thuật toán. Nếu bạn quan tâm đến chi tiết về cách sắp xếp diễn ra, bạn sẽ muốn xem mã nguồn Sun.

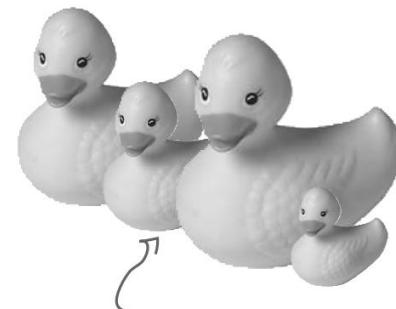
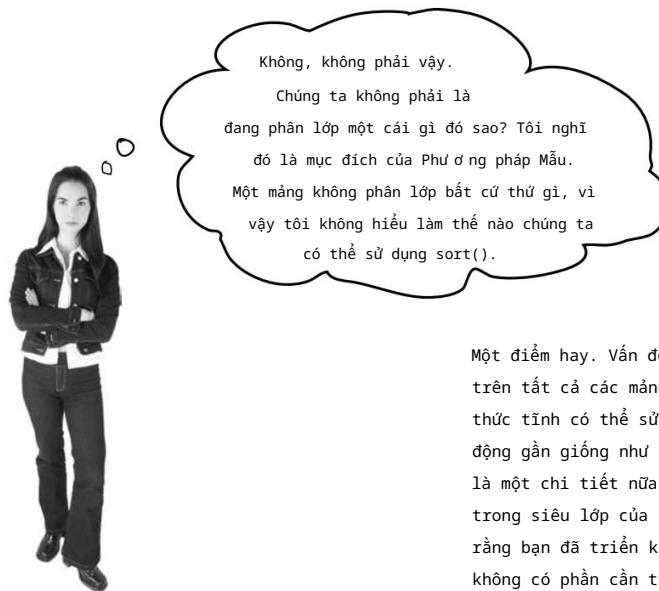
Hãy coi đây là phu ơ ng pháp mău.

Đây là một phu ơ ng thức cụ thể đã được định nghĩa trong lớp Arrays.

compareTo() là phu ơ ng thức chúng ta cần triển khai để "hoàn thiện" phu ơ ng thức mău.

Chúng ta cần phải giải quyết một số vấn đề...

Giả sử bạn có một mảng vịt mà bạn muốn sắp xếp. Bạn làm thế nào? Vâng, phư ơng pháp mẫu sắp xếp trong Mảng cung cấp cho chúng ta thuật toán, nhưng bạn cần cho nó biết cách so sánh vịt, mà bạn thực hiện bằng cách triển khai phư ơng pháp `compareTo()`... Bạn có hiểu không?



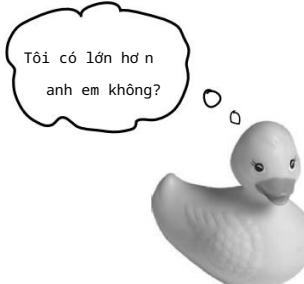
Chúng tôi có một loạt các
Vịt
chúng ta cần phải sắp xếp.

Một điểm hay. Vấn đề là: các nhà thiết kế `sort()` muốn nó hữu ích trên tất cả các mảng, vì vậy họ phải biến `sort()` thành một phư ơng thức tĩnh có thể sử dụng ở bất kỳ đâu. Nhưng không sao, nó hoạt động gần giống như khi nó ở trong một siêu lớp. Bây giờ, đây là một chi tiết nữa: vì `sort()` thực sự không được định nghĩa trong siêu lớp của chúng ta, nên phư ơng thức `sort()` cần biết rằng bạn đã triển khai phư ơng thức `compareTo()`, nếu không, bạn sẽ không có phần cần thiết để hoàn thành thuật toán sắp xếp.

Để xử lý điều này, các nhà thiết kế đã sử dụng giao diện `Comparable`. Tất cả những gì bạn phải làm là triển khai giao diện này, có một phư ơng thức (thật bất ngờ): `compareTo()`.

`compareTo()` là gì?

Phư ơng thức `compareTo()` so sánh hai đối tượng và trả về kết quả xem đối tượng nào nhỏ hơn, lớn hơn hay bằng đối tượng kia. `Sort()` sử dụng điều này làm cơ sở để so sánh các đối tượng trong mảng.



thực hiện so sánh

So sánh Vịt và Việt

Đú ợc rồi, vậy là bạn biết rằng nếu muốn sắp xếp Ducks, bạn sẽ phải triển khai phương thức compareTo() này; bằng cách đó, bạn sẽ cung cấp cho lớp Arrays những gì nó cần để hoàn thiện thuật toán và sắp xếp Ducks của bạn.



Sau đây là cách thực hiện của con vịt:

```

lớp công khai Duck thực hiện Comparable {
    tên chuỗi;
    trọng lượng int;

    public Duck(String name, int weight) {
        this.name = tên;
        this.weight = trọng lượng;
    }

    công khai String toString() {
        trả về tên + "cân nặng" + trọng lượng;
    }

    public int compareTo(Object đối tượng) {
        Việt otherDuck = (Việt)đối tượng;

        nếu (this.weight < otherDuck.weight) {
            trả về -1;
        } nếu không thì nếu (this.weight == otherDuck.weight) {
            trả về 0;
        } else { // this.weight > otherDuck.weight
            trả về 1;
        }
    }
}

```

Hãy nhớ rằng chúng ta cần triển khai giao diện Comparable vì chúng ta không thực sự tạo lớp con.

Vịt của chúng tôi có tên và trọng lượng

Chúng tôi muốn đơn giản hóa vấn đề; tất cả những gì Việt làm là in tên và cân nặng của chúng!

Đú ợc rồi, đây là những gì sort cần...

compareTo() sử dụng một con Duck khác để so sánh con Duck NÀY.

Đây là nơi chúng ta chỉ định cách so sánh Việt. Nếu Việt NÀY nhẹ hơn Việt khác thì chúng ta trả về -1; nếu chúng bằng nhau, chúng ta trả về 0; và nếu Việt NÀY nặng hơn, chúng ta trả về 1.

mẫu phu' ơ ng pháp mẫu

Chúng ta hãy phân loại một số con vịt

Đây là bài kiểm tra phân loại Việt...

```

lớp công khai DuckSortTestDrive { công khai
    tĩnh void main(String[] args) {
        Vịt[] vịt = {
            Vịt mới("Daffy", 8), Vịt
            mới("Dewey", 2), Vịt
            mới("Howard", 7), Vịt
            mới("Louie", 2), Vịt
            mới("Donald", 10), Vịt mới("Huey",
            2)
        };
        Lưu ý rằng chúng ta
        gọi phương thức tĩnh của
        Mảng là sort và truyền cho
        nó Ducks.

        Mảng.sort(vịt);
        System.out.println("\nSau khi sắp xếp:");
        System.out.println("Trước khi sắp xếp:");
        display(ducks);
    }
}

public static void display(Duck[] vịt) {
    đối với (int i = 0; i < ducks.length; i++) {
        System.out.println(vịt[i]);
    }
}
}

```

Lưu ý rằng chúng ta
gọi phương thức tĩnh của
Mảng là sort và truyền cho
nó Ducks.

Mảng.sort(vịt);
System.out.println("\nSau khi sắp xếp:");
System.out.println("Trước khi sắp xếp:");
display(ducks);

Đến giờ phân loại rồi!

Chúng ta cần một loạt
Vịt; trông chúng có vẻ ngon.

Hãy in chúng ra để xem
tên và trọng lượng của chúng.

Hãy in chúng ra (một lần nữa) để xem
tên và trọng lượng của chúng.

Hãy bắt đầu phân loại nhé!

```

Cửa sổ chính sửa tệp Trợ giúp DonaldNeedsToGoOnADiet
%java DuckSortTestDrive

Trước khi phân loại:
Daffy nặng 8
Dewey nặng 2
Howard nặng 7
Louie nặng 2
Donald nặng 10
Huey nặng 2

Những chú vịt không được phân loại
Sau khi sắp xếp:
Dewey nặng 2
Louie nặng 2
Huey nặng 2
Howard nặng 7
Daffy nặng 8
Donald nặng 10 %
Vịt được phân loại

```

bạn đang ở đây 4 303

hậu trư ờng: phân loại vịt

Quá trình chế tạo máy phân loại vịt

Chúng ta hãy cùng theo dõi cách thức hoạt động của phương thức mẫu Arrays sort(). Chúng ta sẽ xem phương thức mẫu kiểm soát thuật toán như thế nào và tại một số điểm nhất định trong thuật toán, cách thức nó yêu cầu Ducks của chúng ta cung cấp triển khai của một bút ớc...

Hậu
trư ờng



1

Đầu tiên, chúng ta cần một mảng Ducks:

```
Vịt[] vịt = {vịt mới("Daffy", 8), ...};
```

2

Sau đó, chúng ta gọi phương thức mẫu sort() trong lớp Array và truyền cho nó các ducks của chúng ta:

```
Mảng.sort(vịt);
```

Phương thức sort() (và trình trợ giúp mergeSort()) kiểm soát quy trình sắp xếp.

3

Để sắp xếp một mảng, bạn cần so sánh từng phần tử một cho đến khi toàn bộ danh sách được sắp xếp theo thứ tự.

Khi so sánh hai con vịt, phương pháp sắp xếp dựa vào phương pháp compareTo() của Duck để biết cách thực hiện điều này. Phương pháp compareTo() được gọi trên con vịt đầu tiên và truyền con vịt để so sánh:

```
vịt[0].compareTo(vịt[1]);
```

Vịt đầu tiên

Vịt để so sánh với

```
đổi với (int i=thấp; i<cao; i++){
    ... so sánh() ...
    ... tráo đổi() ...
}
```

Phương thức sort() kiểm soát thuật toán, không lớp nào có thể thay đổi điều này. sort() dựa vào lớp Comparable để cung cấp triển khai compareTo()

4

Nếu các chú Vịt không được sắp xếp theo thứ tự, chúng sẽ được hoán đổi bằng phương thức swap() cụ thể trong Mảng:

```
tráo đổi()
```

Con vịt
so sánh với() đếnChuỗi()

Không có sự kế thừa, không giống như phương pháp mẫu thông thường.

Mảng
loại() tráo đổi()

5

Phương pháp sắp xếp tiếp tục so sánh và hoán đổi các Vịt cho đến khi mảng có thứ tự đúng!

mẫu phư ơ ng pháp mẫu

không có Những câu hỏi ngớ ngẩn

Q: Đây có thực sự là Mẫu không?

Phư ơ ng pháp mẫu, hay bạn đang cố gắng quá mức?

A: Mẫu này đòi hỏi phải thực hiện một thuật toán và để các lớp con cung cấp việc triển khai các bù ớc - và sắp xếp Mảng xõ ràng không làm điều đó! Nhưng, như chúng ta đã biết, các mẫu trong tự nhiên không phải lúc nào cũng giống như các mẫu trong sách giáo khoa. Chúng phải được sửa đổi để phù hợp với ngữ cảnh và những hạn chế khi thực hiện.

Các nhà thiết kế phư ơ ng thức sort() của Arrays có một vài hạn chế. Nhìn chung, bạn không thể phân lớp một mảng Java và họ muốn sắp xếp được sử dụng trên tất cả các mảng (và mỗi mảng là một lớp khác nhau). Vì vậy, họ đã định nghĩa một phư ơ ng thức tĩnh và hoàn phần so sánh của

thuật toán để sắp xếp các mục.

Vì vậy, mặc dù không phải là phư ơ ng pháp mẫu sách giáo khoa, như việc triển khai này vẫn theo tinh thần của Mẫu phư ơ ng pháp mẫu. Ngoài ra, bằng cách loại bỏ yêu cầu bạn phải phân lớp Mảng để sử dụng thuật toán này, họ đã làm cho việc sắp xếp theo một số cách linh hoạt và hữu ích hơn.

Q: Việc triển khai sắp xếp này

thực sự có vẻ giống với Strategy Pattern hơn là Template Method Pattern. Tại sao chúng ta lại coi nó là Template Method?

A: Có lẽ bạn đang nghĩ rằng

vì Strategy Pattern sử dụng thành phần đối tư ợng. Bạn nói đúng theo một cách nào đó - chúng tôi

sử dụng đối tư ợng Arrays để sắp xếp mảng của chúng ta, vì vậy nó tương tự như Strategy. Nhưng hãy nhớ rằng, trong Strategy, lớp mà bạn soạn thảo sẽ triển khai toàn bộ thuật toán. Thuật toán mà Arrays triển khai để sắp xếp là không đầy đủ; nó cần một lớp để diễn vào phư ơ ng thức compareTo() còn thiếu. Vì vậy, theo cách đó, nó giống với Template Method hơn.

Q: Có ví dụ nào khác không?

phư ơ ng thức mẫu trong Java API?

A: Vâng, bạn sẽ tìm thấy chúng trong một vài nơi. Ví dụ, java.io có phư ơ ng thức read() trong InputStream mà các lớp con phải triển khai và được sử dụng bởi phư ơ ng thức mẫu read(byte b[], int off, int len).

não Apower²

Chúng ta biết rằng chúng ta nên ưu tiên thành phần hơn là kế thừa, đúng không? Vâng, những người triển khai phư ơ ng thức mẫu sort() đã quyết định không sử dụng kế thừa và thay vào đó triển khai sort() như một phư ơ ng thức tĩnh được kết hợp với Comparable khi chạy. Điều này tốt hơn như thế nào? Nó tệ hơn như thế nào? Bạn sẽ tiếp cận vấn đề này như thế nào? Mảng Java có làm cho điều này trở nên đặc biệt khó khăn không?

não Apower²

Hãy nghĩ đến một mẫu khác là chuyên môn hóa của phư ơ ng thức mẫu. Trong chuyên môn hóa này, các hoạt động nguyên thủy được sử dụng để tạo và trả về các đối tượng. Đây là mẫu nào?

móc sơ n

Đu đưa với Khung

Tiếp theo trong chuyến thám hiểm Phư ơng pháp Mẫu của chúng tôi... hãy chú ý đến JFrames xoay!

Nếu bạn chưa từng gặp JFrame, thì đây là container Swing cơ bản nhất và kế thừa phư ơng thức paint().

Theo mặc định, paint() không làm gì cả vì nó là một hook! Bằng cách ghi đè paint(), bạn có thể chèn chính mình vào thuật toán của JFrame để hiển thị khu vực màn hình của nó và có đầu ra đồ họa của riêng bạn được tích hợp vào JFrame. Sau đây là một ví dụ đơn giản đến xấu hổ về việc sử dụng JFrame để ghi đè phư ơng thức hook paint():

```

lớp công khai MyFrame mở rộng JFrame {
    công khai MyFrame(Chuỗi tiêu đề) {
        siêu(tiêu đề);
        này.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        này.setSize(300,300);
        this.setVisible(dùng);
    }

    public void paint(Đồ họa đồ họa) {
        super.paint(đồ họa);
        Chuỗi tin nhắn = "Tôi cai trị!!";
        đồ họa.drawString(tin nhắn, 100, 100);
    }
}

public static void main(String[] args) {
    MyFrame myFrame = new MyFrame("Mẫu thiết kế ưu tiên hàng đầu");
}
}

```

Chúng tôi đang mở rộng JFrame, chứa phư ơng thức update() để điều khiển thuật toán cập nhật màn hình.
Chúng ta có thể sử dụng thuật toán đó bằng cách ghi đè phư ơng thức hook paint().

Đừng nhìn đằng sau bức màn! Chỉ là một số khởi tạo ở đây...

Thuật toán cập nhật của JFrame gọi paint(). Theo mặc định, paint() không làm gì cả... đó là một hook.
Chúng tôi đang ghi đè paint() và yêu cầu JFrame vẽ một thông điệp trong cửa sổ.



I rule!!

Đây là thông điệp được vẽ trong khung vì chúng ta đã nói vào phư ơng thức paint().

Các ứng dụng nhỏ

Điểm dừng chân cuối cùng của chúng tôi trong chuyến đi săn: applet.

Bạn có thể biết applet là một chương trình nhỏ chạy trong trang web. Bất kỳ applet nào cũng phải là lớp con của Applet và lớp này cung cấp một số hook. Chúng ta hãy xem một vài trong số chúng:



```
lớp công khai MyApplet mở rộng Applet {
    Tin nhắn chuỗi;
```

Môc init cho phép applet thực hiện bất kỳ thao tác nào nó muốn để khởi tạo applet lần đầu tiên.

```
công khai void init() {
    message = "Xin chào thế giới, tôi còn sống!";
    sờn lại();
}
```

repaint() là một phương thức cụ thể trong lớp Applet cho phép các thành phần cấp cao hơn biết applet cần được vẽ lại.

```
công khai void bắt đầu() {
    message = "Bây giờ tôi đang bắt đầu...";
    sờn lại();
}
```

Môc bắt đầu cho phép applet thực hiện một thao tác nào đó khi applet sắp được hiển thị trên trang web.

```
công khai void dừng() {
    message = "Ồ, giờ thì tôi bị chặn lại rồi...";
    sờn lại();
}
```

Nếu người dùng chuyển sang trang khác, hook dừng sẽ được sử dụng và applet có thể thực hiện bất kỳ hành động nào cần thiết để dừng các hành động của người dùng.

```
công khai void destroy() {
    // applet sẽ không còn nữa...
}
```

Và hook destroy được sử dụng khi applet sắp bị hủy, ví dụ, khi cửa sổ trình duyệt bị đóng. Chúng ta có thể thử hiển thị một cái gì đó ở đây, nhưng mục đích là gì?

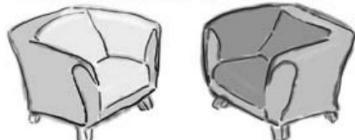
```
}
```

Vâng, hãy nhìn đây! Người bạn cũ của chúng ta là phương thức paint()! Applet cũng sử dụng phương thức này như một hook.

Các applet cụ thể sử dụng rộng rãi các hook để cung cấp hành vi của riêng chúng. Vì các phương pháp này được triển khai dưới dạng hook nên applet không cần phải triển khai chúng.

trò chuyện bên lò sưởi: phư ơ ng pháp và chiến lư ợc mẫu

Fireside Chats

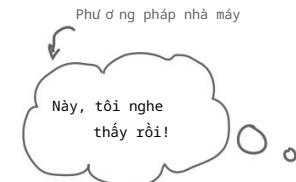


Phư ơ ng pháp mẫu

Này Strategy, anh đang làm gì trong chư ơ ng của tôi
thé? Tôi nghĩ mình sẽ bị mắc kẹt với một ngư ời nhầm chán
như Factory Method.

Bài nói chuyện tối nay: Phư ơ ng pháp mẫu và Chiến lư ợc
so sánh các phư ơ ng pháp.

Chiến lư ợc



Tôi chỉ dùa thôi! Nhưng nghiêm túc mà nói, bạn đang làm gì
ở đây vậy? Chúng tôi đã không nghe tin gì từ bạn trong tám
chư ơ ng rồi!

Không, là tôi đây, như ng hấy cắn thận nhé - bạn và Factory
Method có liên quan đến nhau, đúng không?

Tôi nghe nói bạn đang hoàn thiện bản thảo cuối cùng của
chư ơ ng và tôi nghĩ tôi sẽ ghé qua để xem tiến độ thế nào.
Chúng ta có nhiều điểm chung, vì vậy tôi nghĩ tôi có thể
giúp đư ợc bạn...

Có thể bạn muốn nhắc lại với ngư ời đọc về mục đích của
bạn vì đã lâu rồi.

Tôi không biết, kể từ Chư ơ ng 1, mọi ngư ời đã chặn tôi
trên phố và nói, "Bạn không phải là mẫu đó sao..." Vì vậy,
tôi nghĩ họ biết tôi là ai. Nhưng vì lợi ích của bạn: Tôi
định nghĩa một họ thuật toán và làm cho chúng có thể hoán
đổi cho nhau.
Vi mỗi thuật toán đều đư ợc đóng gói nên khách hàng có thể dễ
dàng sử dụng nhiều thuật toán khác nhau.

Này, nghe có vẻ giống với những gì tôi làm. Nhưng mục đích
của tôi hơi khác một chút so với bạn; công việc của tôi là
định nghĩa phác thảo của một thuật toán, như ng để các
lớp con của tôi thực hiện một số công việc. Theo cách đó,
tôi có thể có các triển khai khác nhau của các bư ớc riêng lẻ
của một thuật toán, như ng vẫn kiểm soát đư ợc cấu trúc của
thuật toán. Có vẻ như bạn phải từ bỏ quyền kiểm soát các
thuật toán của mình.

Tôi không chắc mình có thể diễn đạt như vậy không...
và dù sao thì tôi cũng không bị mắc kẹt khi sử dụng kế
thừa cho việc triển khai thuật toán. Tôi cung cấp cho khách
hang lựa chọn triển khai thuật toán thông qua thành phần đối
tư ợng.

mẫu phu' ơ ng pháp mẫu

Phu' ơ ng pháp mẫu

Tôi nhớ điều đó. Như ng tôi kiềm soát thuật toán của mình nhiều hơn và tôi không sao chép mã. Trên thực tế, nếu mọi phần trong thuật toán của tôi đều giống nhau ngoại trừ, chẳng hạn, một dòng, thì các lớp của tôi hiệu quả hơn nhiều so với lớp của bạn. Tất cả mã sao chép của tôi đều được đưa vào siêu lớp, vì vậy tất cả các lớp con có thể chia sẻ nó.

Vâng, tôi rất vui cho bạn, như ng đừng quên tôi là mẫu ngư ời đư ợc sử dụng nhiều nhất.

Tại sao? Bởi vì tôi cung cấp một phu' ơ ng pháp cơ bản để tái sử dụng mã cho phép các lớp con chỉ định hành vi. Tôi chắc rằng bạn có thể thấy rằng điều này hoàn hảo để tạo các khuôn khổ.

Thế nào? Lớp siêu cấp của tôi là trừu tư ợng.

Như tôi đã nói Strategy, tôi thực sự vui mừng cho bạn. Cảm ơn bạn đã ghé thăm, như ng tôi phải hoàn thành phần còn lại của chư ơ ng này.

Tôi hiểu rồi. Đừng gọi cho chúng tôi, chúng tôi sẽ gọi cho bạn...

Chiến lư ợc

Bạn có thể hiệu quả hơn một chút (chỉ một chút) và yêu cầu ít đối tư ợng hơn. Và bạn cũng có thể ít phức tạp hơn một chút so với mô hình ủy quyền của tôi, nhưng tôi linh hoạt hơn vì tôi sử dụng thành phần đổi tư ợng. Với tôi, khách hàng có thể thay đổi thuật toán của họ khi chạy chỉ bằng cách sử dụng một đối tư ợng chiến lư ợc khác.

Thôi nào, họ không chọn tôi cho Chư ơ ng 1 một cách vô cớ đâu!

Vâng, tôi đoán vậy... như ng còn sự phụ thuộc thì sao?
Bạn phụ thuộc nhiều hơn tôi.

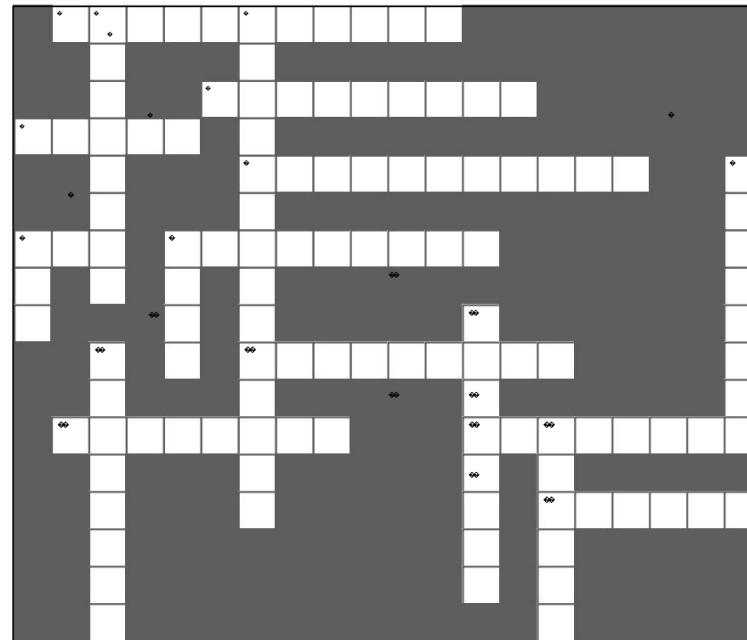
Như ng bạn phải phụ thuộc vào các phu' ơ ng thức đư ợc triển khai trong siêu lớp của bạn, là một phần của thuật toán của bạn. Tôi không phụ thuộc vào bất kỳ ai; tôi có thể tự mình thực hiện toàn bộ thuật toán!

Đư ợc rồi, đư ợc rồi, đừng động chạm. Tôi sẽ để bạn làm việc, như ng hãy cho tôi biết nếu bạn cần các kỹ thuật đặc biệt của tôi, tôi luôn sẵn lòng giúp đỡ.

trò chơi ô chữ

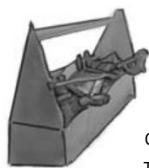


Lại đến thời điểm đó rồi....



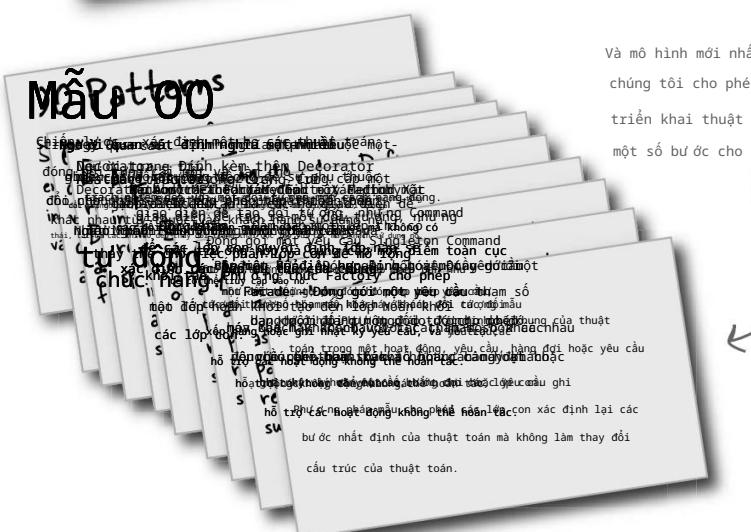
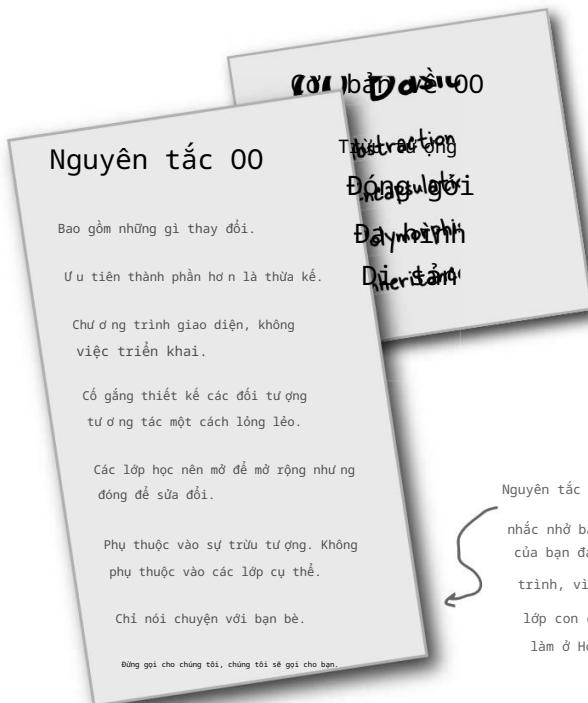
ĐIỂM ĐÓ RỒI
HƯỚNG

mẫu phư ơ ng pháp mẫu



Công cụ cho hộp công cụ thiết kế của bạn

Chúng tôi đã thêm Template Method vào hộp công cụ của bạn. Với Template Method, bạn có thể tái sử dụng mã như một chuyên gia trong khi vẫn kiểm soát được các thuật toán của mình.



Tải xuống tại WoweBook.Com

ĐIỂM ĐẦU TIÊN

"Phuơng pháp mẫu" xác định các
bùớc của thuật toán, chuyển
giao cho các lớp con để thực hiện
các bùớc đó.

Mẫu phư ơ ng pháp mẫu cung cấp
cho chúng ta một kỹ thuật quan trọng để tái sử dụng mã.

B Phư ơ ng pháp mẫu
Lớp trừu tượng có thể định nghĩa các phư ơ ng thức cụ thể, phư ơ ng thức trừu tượng và các hook.

Các phương thức trừu tượng

B Hooks là phương pháp thực hiện không có gì hoặc hành vi mặc định trong lớp trừu tượng, nhưng có thể bị ghi đè trong lớp con.

B Để ngăn các lớp con thay đổi thuật toán trong phương thức mẫu, hãy khai báo phương thức mẫu là final.

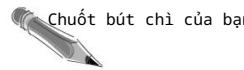
Nguyên tắc Hollywood hứa ứng dẫn chúng ta
đưa việc ra quyết định vào các
mô-đun cấp cao có thể quyết
định cách thức và thời điểm gọi
các mô-đun cấp thấp.

Bạn sẽ thấy rất nhiều cách sử dụng
Mẫu phương pháp mẫu trong mã
thực tế, như ng dung mong
đợi tất cả (giống như bắt kỳ mẫu
nào) đều đư ợc thiết kế "theo sách".

B Phư ơ ng pháp Chiến lư ợc và Mẫu
Các mẫu đều bao gồm các
thuật toán, một theo cách kế
thừa và một theo cách hợp
thành.

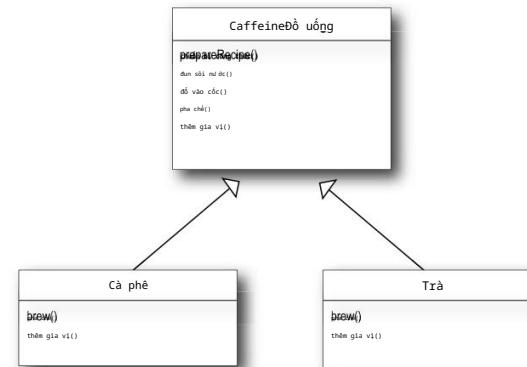
β Phư ơ ng pháp Nhà máy là một chuyên ngành của Phư ơ ng pháp Mẫu.

giải bài tập



Giải pháp bài tập

Vẽ sơ đồ lớp mới sau khi chúng ta đã di chuyển `prepareRecipe()` vào lớp `CaffeineBeverage`:



Ghép mỗi mẫu với mô tả của nó:

Mẫu

Sự miêu tả

Phương pháp mẫu

Đóng gói các hành vi có thể hoán đổi cho nhau và sử dụng sự phân quyền để quyết định hành vi nào sẽ sử dụng

Chiến lược

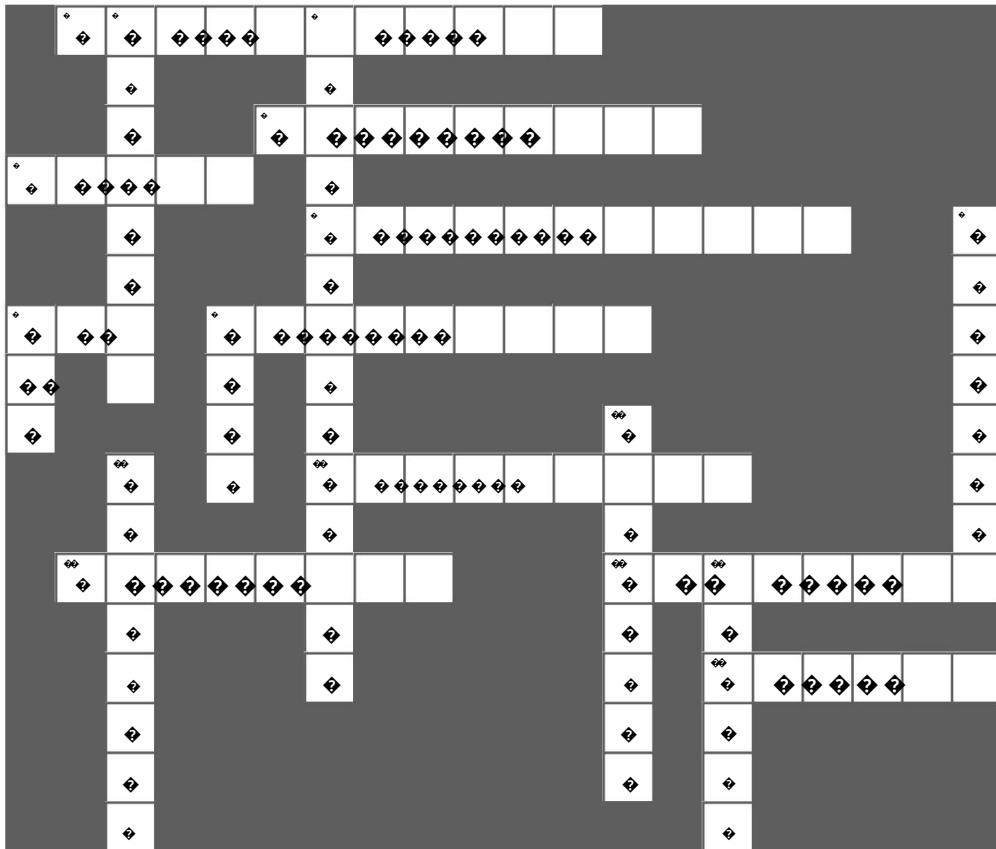
Các lớp con quyết định cách thực hiện các bước trong một thuật toán

Phương pháp nhà máy

Các lớp con quyết định lớp cụ thể nào sẽ được tạo



Giải pháp bài tập



_____ —

_____ _____

_____ _____

_____ _____ _____ _____ _____ _____

_____ _____ _____ _____ _____ _____

_____ _____ _____ _____ _____ _____

_____ _____ _____ _____ _____ _____

_____ _____ _____ _____ _____ _____

_____ _____ _____ _____ _____ _____

bạn đang ở đây 4 313

_____ _____ _____ _____ _____ _____

_____ _____ _____ _____ _____ _____

9 Iterator và các m^{ẫu} t^{ổng} hợp

g

h

Bộ sưu tập đư ợc quản lý tốt

g



Có rất nhiều cách để nhồi nhét các đối tượng vào một bộ sưu tập. Đặt chúng trong một Mảng, một Ngăn xếp, một Danh sách, một Bảng băm, hãy chọn lựa của bạn. Mỗi cái đều có những ưu điểm riêng và sự đánh đổi. Nhưng tại một thời điểm nào đó, khách hàng của bạn sẽ muốn lặp lại các đối tượng đó và khi anh ấy làm vậy, bạn có cho anh ấy xem cách thực hiện của bạn không? Chúng tôi chắc chắn hy vọng là không! Điều đó sẽ không chuyên nghiệp. Vâng, bạn không cần phải mạo hiểm sự nghiệp của mình; bạn sẽ để xem làm thế nào bạn có thể cho phép khách hàng của mình lặp lại các đối tượng của bạn mà không bao giờ nhận đư ợc một cái nhìn thoáng qua về cách bạn lưu trữ các đối tượng của mình. Bạn cũng sẽ học cách tạo ra một số bộ sưu tập siêu đối tượng có thể vươn qua một số cấu trúc dữ liệu ẩn trong một ràng buộc. Và nếu điều đó vẫn chưa đủ, bạn cũng sẽ học đư ợc một hoặc hai điều về đối tượng trách nhiệm.

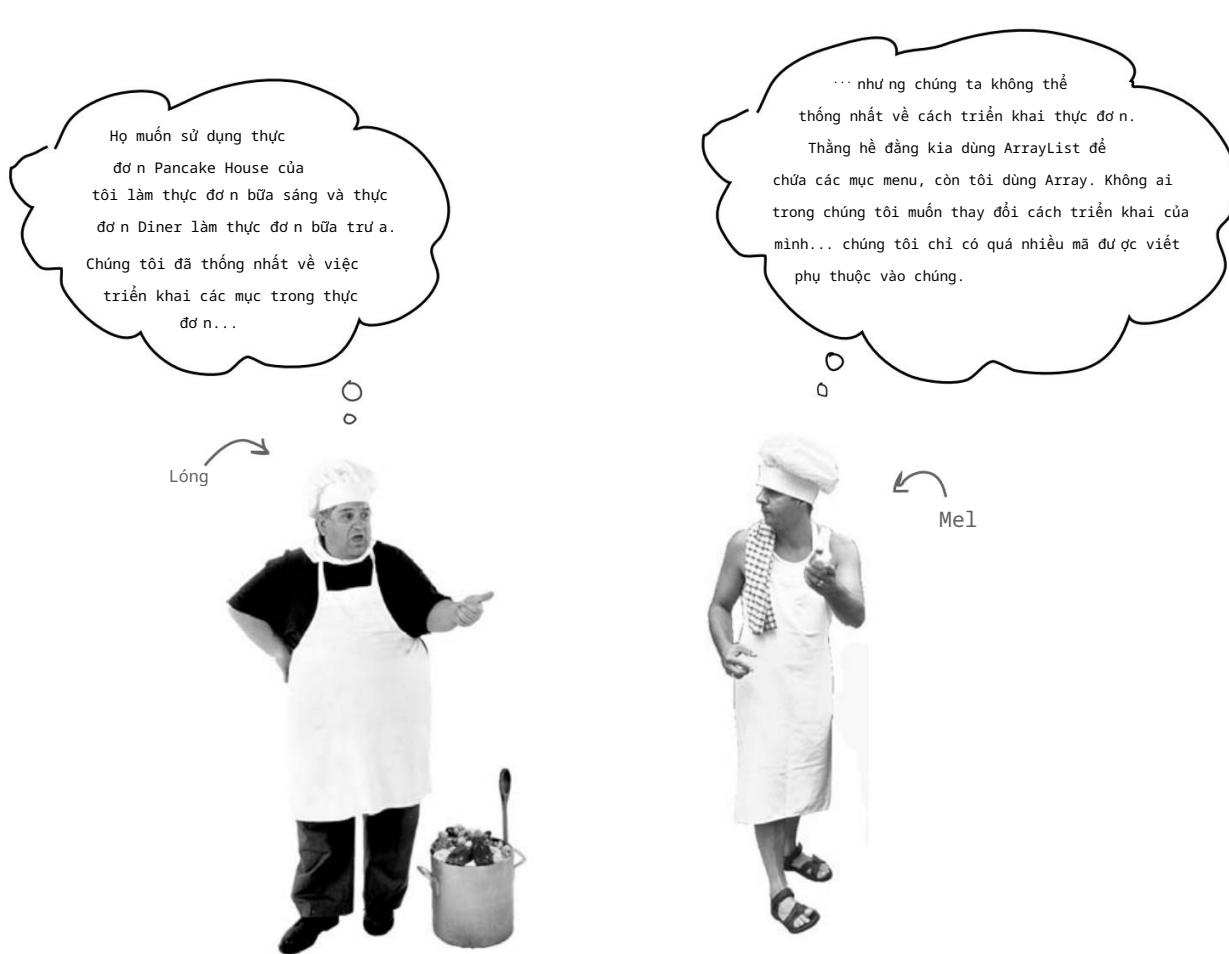
đây là một chương mới

315

tin lớn

Tin tức mới nhất: Quán ăn Objectville và Hợp nhất nhà hàng bánh kếp Objectville

Thật là tin tuyệt vời! Ngày giờ chúng ta có thể thư giãn thức bữa sáng bánh kếp ngon lành tại Pancake House và bữa trưa ngon lành tại Diner ở cùng một nơi.
Nhưng có vẻ như có một vấn đề nhỏ...



trình lặp và các mẫu tổng hợp

Kiểm tra các mục Menu

Ít nhất thì Lou và Mel cũng đồng ý về việc triển khai `MenuItem`s. Chúng ta hãy xem xét các mục trong từng menu và cách thực hiện.

Thực đơn Diner có nhiều món ăn trú a, trong khi Pancake House bao gồm các món ăn sáng. Mỗi món ăn trong thực đơn đều có tên, mô tả và giá

```

lớp công khai MenuItem {
    Tên chuỗi;
    Mô tả chuỗi;
    người ăn chay theo kiểu Boolean;
    giá gấp đôi;

    public MenuItem(String tên,
                    Mô tả chuỗi, boolean
                    vegetarian, giá gấp đôi)

    {
        this.name = tên;
        this.description = mô tả;
        this.vegetarian = ăn chay;
        this.price = giá;
    }

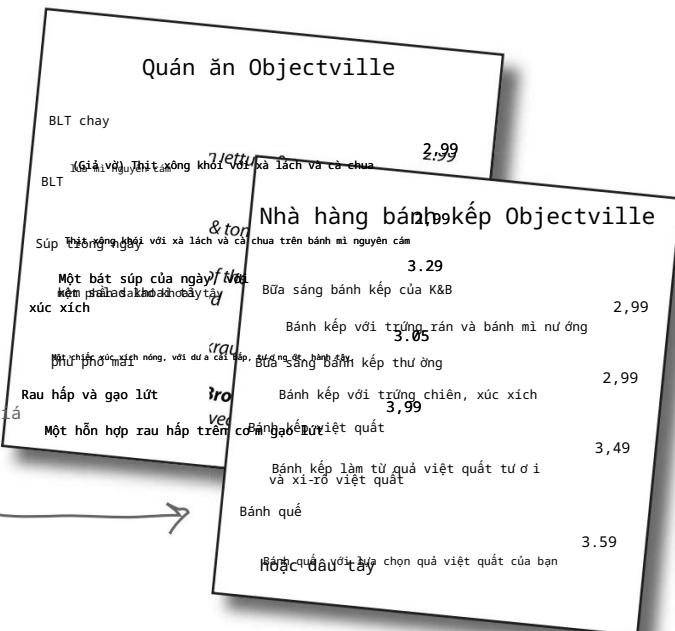
    công khai String getName() {
        trả về tên;
    }

    công khai String getDescription() {
        mô tả trả về;
    }

    công khai double getPrice() {
        giá trả lại;
    }

    công khai boolean isVegetarian() {
        trả lại chế độ ăn chay;
    }
}

```



`MenuItem` bao gồm tên, mô tả, cờ để chỉ ra liệu mục đó có phải là đồ chay không và giá. Bạn truyền tất cả các giá trị này vào hàm tạo để khởi tạo `MenuItem`.

Các phương thức getter này cho phép bạn truy cập vào các trường của mục menu.

hai thực đơn

Thực hiện Menu của Lou và Mel

Bây giờ chúng ta hãy xem Lou và Mel đang tranh cãi về điều gì. Cả hai đều đầu tư nhiều thời gian và mã vào cách lưu trữ các mục menu trong menu, và nhiều mã khác phụ thuộc vào nó.



Đây là cách Lou thực hiện thực
đơn của Pancake House.

```
lớp công khai PancakeHouseMenu thực hiện Menu {
```

```
    Danh sách các mục menu;
```

```
    công khai PancakeHouseMenu() {
        menuItems = ArrayList mới();
```

Lou đang sử dụng ArrayList để lưu trữ các
mục menu của mình

```
        addItem("Bữa sáng bánh kếp của K&B", "Bánh kếp với trứng  
xán và bánh mì nư ống", đúng,  
2,99);
```

Mỗi mục menu được thêm vào
ArrayList ở đây, trong hàm tạo

```
        addItem("Bữa sáng bánh kếp thông thư ống",
            "Bánh kếp với trứng chiên, xúc xích", sai,  
2,99);
```

Mỗi MenuItem đều có
tên, mô tả, có phải là
món chay hay không và giá cả

```
        addItem("Bánh kếp việt quất",
            "Bánh kếp làm từ quả việt quất tươi",
            ĐÚNG VÀY,  
3,49);
```

```
        addItem("Bánh quê",
            "Bánh quê, với lựa chọn việt quất hoặc dâu tây của bạn",
            ĐÚNG VÀY,  
3.59);
```

Để thêm một mục menu, Lou tạo một mục mới
Đối tượng MenuItem, truyền vào mỗi đối số,
sau đó thêm nó vào ArrayList

```
    }
```

```
    public void addItem(String name, String description,
                        boolean chay, giá gấp đôi)
```

```
    {
        MenuItem menuItem = new MenuItem(tên, mô tả, đồ chay, giá);
        menuItems.add(menuItem);
    }
```

```
    công khai ArrayList getMenuItems() {
        trả về menuItems;
```

Phương thức getMenuItems() trả về danh sách các mục menu

```
}
```

```
// các phương pháp menu khác ở đây
```

Lou có một loạt mã menu khác phụ thuộc vào
triển khai ArrayList. Anh ấy không muốn phải
viết lại tất cả mã đó!

trình lặp và các mảng tổng hợp



Haah! Một ArrayList...
Tôi đã sử dụng một Mảng THỰC
để có thể kiểm soát kích thước tối đa
của menu và lấy MenuItem mà không
cần phải sử dụng ép kiểu.

Và đây là cách Mel thực hiện thực đơn Diner.

```
lớp công khai DinerMenu thực hiện Menu {
    int cuối cùng tĩnh MAX_ITEMS = 6;
    int số mục = 0;
    MenuItem[] menuItems;
```

Mel sử dụng một cách tiếp cận khác; anh ấy sử dụng một Mảng để
có thể kiểm soát kích thước tối đa của menu và lấy các mục
menu ra mà không cần phải truyền các đối tượng của mình.

```
công khai DinerMenu() {
    menuItems = MenuItem mới[MAX_ITEMS];
    addItem("BLT chay",
        "Thịt xông khói già với xà lách và cà chua trên bánh mì nguyên cám", đúng, 2,99);
    thêm mục("BLT",
        "Thịt xông khói già với xà lách và cà chua trên bánh mì nguyên cám", sai, 2,99);
    addItem("Súp trong ngày",
        "Súp trong ngày, kèm theo salad khoai tây", sai, 3.29);
    addItem("Hotdog",
        "Một chiếc xúc xích, với dưa cải bắp, tương ớt, hành tây, phủ phô mai",
        sai, 3.05);
    // một vài mục khác trong Menu Diner được thêm vào đây
}
```

Giống như Lou, Mel tạo các mục menu của mình trong
trình xây dựng, sử dụng phương thức trợ giúp addItem().

```
public void addItem(String name, String description,
                    boolean chay, giá gấp đôi)
```

addItem() lấy tất cả các tham số cần
thiết để tạo MenuItem và khởi tạo một
MenuItem. Nó cũng kiểm tra để đảm bảo rằng
chúng ta chưa đạt đến giới hạn kích thước menu.

```
{
```

```
    MenuItem menuItem = new MenuItem(tên, mô tả, đồ chay, giá);
    nếu (số mục >= MAX_ITEMS) {
        System.err.println("Xin lỗi, menu đã đầy! Không thể thêm mục vào menu");
    } khác {
        menuItems[số mục] = menuItem;
        số mục = số mục + 1;
    }
}
```

Mel đặc biệt muốn giữ thực đơn của mình ở một
mức độ nhất định (có lẽ là để anh ấy không phải
nhớ quá nhiều công thức nấu ăn).

```
công khai MenuItem[] getMenuItems() {
    trả về menuItems;
}
```

getMenuItems() trả về mảng của

mục menu.

```
// các phương pháp menu khác ở đây ← Giống như Lou, Mel có một loạt mã phụ thuộc vào việc triển khai menu của anh  
Ấy là một Mảng. Anh ấy quá bận nấu ăn để viết lại tất cả những thứ này.
```

cô hầu bàn có hỗ trợ java

Có vấn đề gì khi có hai cách trình bày menu khác nhau?

Để hiểu tại sao việc có hai cách trình bày menu khác nhau lại làm mọi thứ trở nên phức tạp, chúng ta hãy thử triển khai một ứng dụng khách sử dụng hai menu. Hãy tưởng tượng bạn được thuê bởi công ty mới thành lập sau khi sáp nhập Diner và Pancake House để tạo ra một cô hầu bàn hỗ trợ Java (sau cùng thì đây là Objectville). Thông số kỹ thuật dành cho cô hầu bàn hỗ trợ Java nêu rõ rằng cô ấy có thể in thực đơn tùy chỉnh cho khách hàng theo yêu cầu và thậm chí cho bạn biết món ăn trong thực đơn có phải là món chay hay không mà không cần phải hỏi đầu bếp - đó quả là một sáng kiến!

Chúng ta hãy xem xét thông số kỹ thuật và sau đó tìm hiểu những gì cần thiết để triển khai...

Đặc tả Waitress đư ợc hỗ trợ Java

```
Cô hầu bàn đư ợc hỗ trợ Java: tên mã là "Alice"

inMenu()
    - in mọi mục trên menu

printBreakfastMenu() - chỉ in các mục bữa sáng

in Menu ăn trưa()
    - chỉ in đồ ăn trưa

printVegetarianMenu() - in tất cả các mục menu chay

isItemVegetarian(name) - cho biết tên của một mục, trả về true
    nếu các mặt hàng là đồ chay, nếu không,
    trả về false
```

Cô hầu bàn là
đang bật Java.



Thông số kỹ
thuật cho Waitress

Chúng ta hãy bắt đầu bằng cách tìm hiểu cách triển khai phương thức `printMenu()`:

- Để in tất cả các mục trên mỗi menu, bạn sẽ cần gọi phương thức `getMenuItem()` trên `PancakeHouseMenu` và `DinerMenu` để lấy các mục menu từ ứng dụng của chúng. Lưu ý rằng mỗi phương thức trả về một kiểu khác nhau:

```
PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
ArrayList breakfastItems = pancakeHouseMenu.getMenuItems();

DinerMenu dinerMenu = new DinerMenu();
MenuItem[] bữa trưaItems = dinerMenu.getMenuItems();
```

Phương pháp này trông giống nhau, nhưng các lệnh gọi trả về các kiểu dữ liệu khác nhau.

Việc triển khai được thể hiện rõ ràng, các món ăn sáng nằm trong một `ArrayList`, các món ăn trưa nằm trong một `Mảng`.

- Bây giờ, để in ra các mục từ `PancakeHouseMenu`, chúng ta sẽ lắp qua các mục trên `breakfastItems ArrayList`. Và để in ra các mục `Diner`, chúng ta sẽ lắp qua `Array`.

```
đối với (int i = 0; i < breakfastItems.size(); i++) {
    MenuItem menuItem = (MenuItem)breakfastItems.get(i);
    System.out.print(menuItem.getName() + " ");
    System.out.println(menuItem.getPrice() + " ");
    System.out.println(menuItem.getDescription());
}

đối với (int i = 0; i < lunchItems.length; i++) {
    MenuItem menuItem = lunchItems[i];
    System.out.print(menuItem.getName() + " ");
    System.out.println(menuItem.getPrice() + " ");
    System.out.println(menuItem.getDescription());
}
```

Bây giờ, chúng ta phải triển khai hai vòng lặp khác nhau để thực hiện từng bước hai mục menu...

...một vòng lặp cho `ArrayList`...
và một cái nữa cho `Mảng`.

- Việc triển khai mọi phương thức khác trong `Waitress` sẽ là một biến thể của chủ đề này. Chúng ta sẽ luôn cần lấy cả hai menu và sử dụng hai vòng lặp để lắp qua các mục của chúng. Nếu một nhà hàng khác có triển khai khác được mua thì chúng ta sẽ có ba vòng lặp.

mục tiêu là gì



Chuốt bút chì của bạn

Dựa trên việc triển khai `printMenu()` của chúng ta, điều nào sau đây được áp dụng?

- A. Chúng tôi đang mã hóa cho các triển khai cụ thể của `PancakeHouseMenu` và `DinerMenu`, không phải cho một giao diện.
- B. Nhân viên phục vụ không triển khai Java `Waitress` API nên cô ấy không tuân thủ theo một tiêu chuẩn nào.
- C. Nếu chúng ta quyết định chuyển từ sử dụng `DinerMenu` là một loại menu khác triển khai danh sách các mục menu bằng `Hashtable`, chúng ta sẽ phải sửa đổi rất nhiều mã trong `Waitress`.
- D. Người phục vụ cần biết cách mỗi menu biểu thị tập hợp các mục menu bên trong của nó; điều này vi phạm tính đóng gói.
- E. Chúng tôi có mã trùng lặp: phư ơng thức `printMenu()` cần hai vòng lặp riêng biệt để lặp lại hai loại khác nhau menu. Và nếu chúng ta thêm menu thứ ba, chúng ta sẽ có thêm một vòng lặp nữa.
- F. Việc thực hiện không dựa trên MXML (Menu XML) và do đó không có khả năng tự ứng tác như mong đợi.

Bây giờ thì sao?

Mel và Lou đang đặt chúng ta vào một vị trí khó khăn. Họ không muốn thay đổi các triển khai của họ vì điều đó có nghĩa là phải viết lại rất nhiều mã trong mỗi lớp menu tư ứng. Nhưng nếu một trong số họ không như mong đợi, thì chúng ta sẽ phải thực hiện một `Waitress` khó bảo trì và mở rộng.

Sẽ thật tuyệt nếu chúng ta có thể tìm ra cách cho phép họ triển khai cùng một giao diện cho menu của họ (họ đã gần hoàn thành rồi, ngoại trừ kiểu trả về của phư ơng thức `getMenuItems()`).

Bằng cách đó, chúng ta có thể giảm thiểu các tham chiếu cụ thể trong mã `Waitress` và hy vọng loại bỏ được nhiều vòng lặp cần thiết để lặp lại cả hai menu.

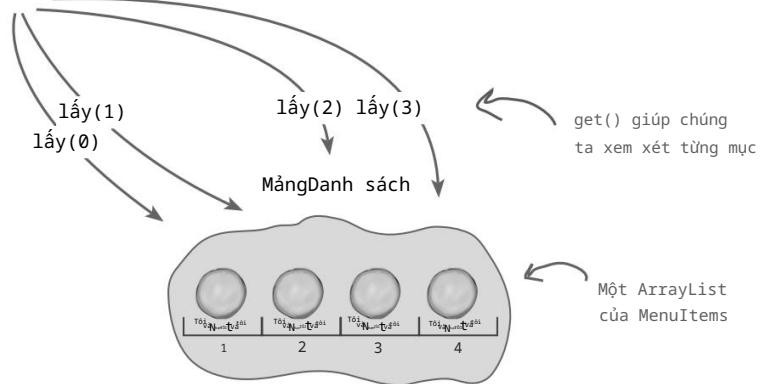
Nghe có vẻ hay phải không? Vậy chúng ta sẽ làm thế nào?

Chúng ta có thể đóng gói quá trình lặp lại không?

Nếu chúng ta học được một điều trong cuốn sách này, đó là hãy đóng gói những gì thay đổi. Rõ ràng là những gì đang thay đổi ở đây: sự lặp lại gây ra bởi các bộ sưu tập đối tượng khác nhau được trả về từ các menu. Nhưng chúng ta có thể đóng gói điều này không? Hãy cùng tìm hiểu ý tưởng này...

- 1 Để lặp qua các mục bữa sáng, chúng ta sử dụng các phương thức `size()` và `get()` trên `ArrayList`:

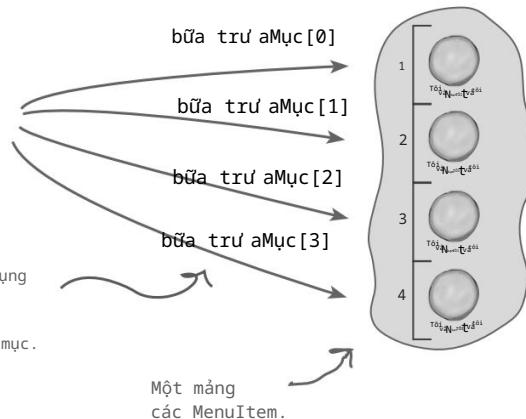
```
đối với (int i = 0; i < breakfastItems.size(); i++) {
    MenuItem menuItem = (MenuItem)breakfastItems.get(i);
}
```



- 2 Và để lặp qua các mục bữa trưa, chúng ta sử dụng truy ẩn Độ dài mảng và ký hiệu chỉ số mảng trên Mảng MenuItem.

```
đối với (int i = 0; i < lunchItems.length; i++) {
    MenuItem menuItem = lunchItems[i];
}
```

Chúng tôi sử dụng
chỉ số mảng để
duyệt qua các mục.



đóng gói lặp lại

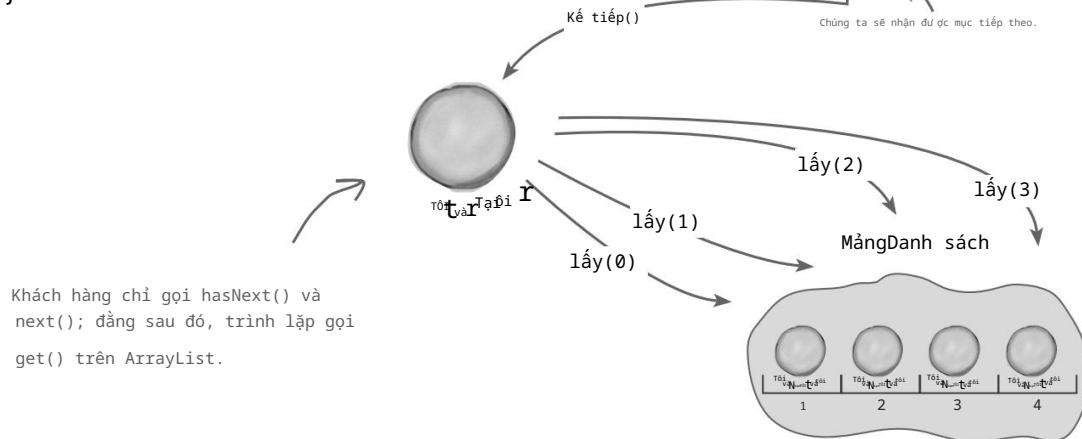
- ③ Bây giờ nếu chúng ta tạo một đối tượng, hãy gọi nó là Iterator, đóng gói cách chúng ta lặp qua một tập hợp các đối tượng thì sao? Hãy thử điều này trên ArrayList

Trình lặp iterator = breakfastMenu.createIterator();

Chúng tôi yêu cầu breakfastMenu
cung cấp một trình lặp cho
các MenuItem của nó.

```
trong khi (iterator.hasNext()) {
    MenuItem menuItem = (MenuItem) iterator.next();
}
```

Và trong khi vẫn còn nhiều mặt hàng còn lại...



- ④ Chúng ta hãy thử điều đó trên Mảng:

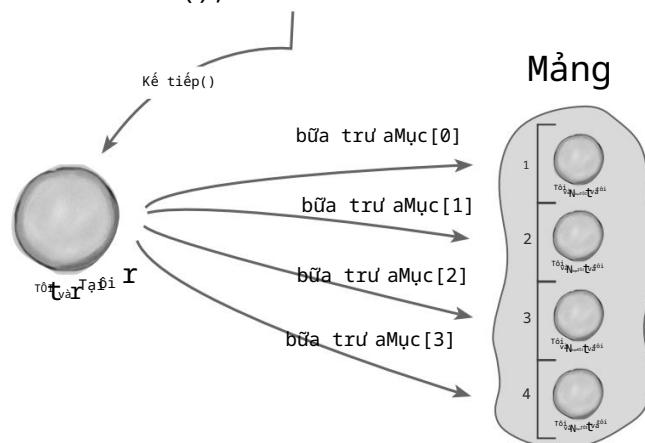
Trình lặp iterator = lunchMenu.createIterator();

```
trong khi (iterator.hasNext()) {
    MenuItem menuItem = (MenuItem) iterator.next();
}
```

Wow, mã này
giống hết mã
breakfastMenu.

Tình huống tương tự ở đây: máy khách
chỉ gọi hasNext() và next(); đằng sau
đó, trình lặp sẽ lập chỉ mục vào Mảng.

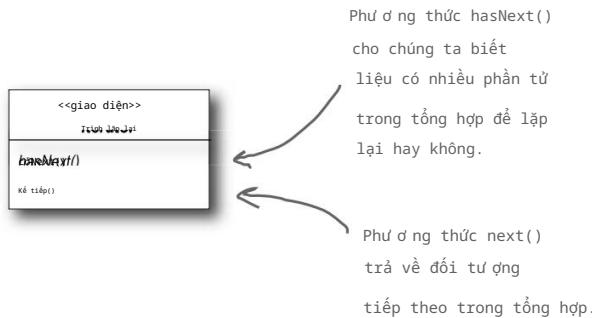
Mảng



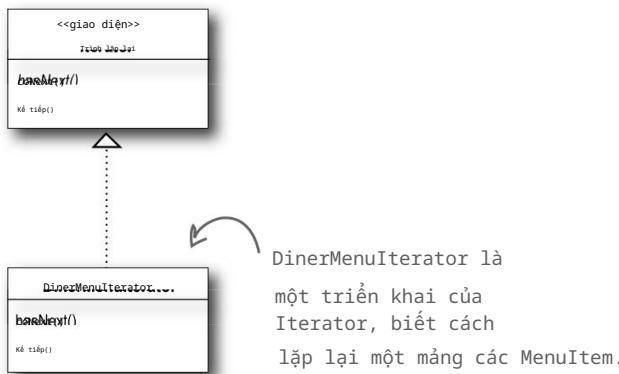
Gặp gỡ mẫu Iterator

Vâng, có vẻ như kế hoạch đóng gói vòng lặp của chúng ta thực sự có thể thành công; và như bạn có thể đã đoán, đó là một Mẫu thiết kế có tên là Mẫu lặp.

Điều đầu tiên bạn cần biết về Iterator Pattern là nó dựa trên một giao diện được gọi là Iterator. Sau đây là một giao diện Iterator có thể có:



Bây giờ, khi chúng ta có giao diện này, chúng ta có thể triển khai Iterator cho bất kỳ loại tập hợp đối tượng nào: mảng, danh sách, bảng băm, ...chọn tập hợp đối tượng yêu thích của bạn. Giả sử chúng ta muốn triển khai Iterator cho Mảng được sử dụng trong DinerMenu. Nó sẽ trông như thế này:



Chúng ta hãy tiếp tục và triển khai Iterator này và kết nối nó vào DinerMenu để xem nó hoạt động như thế nào...

Khi chúng ta nói đến COLLECTION, chúng ta chỉ muốn nói đến một nhóm đối tượng. Chúng có thể được lưu trữ trong các cấu trúc dữ liệu rất khác nhau như danh sách, mảng, bảng băm, nhưng chúng vẫn là các bộ sưu tập. Đôi khi chúng ta cũng gọi chúng là AGGREGATES.



tạo một trình lắp

Thêm Iterator vào DinerMenu

Để thêm Iterator vào DinerMenu, trước tiên chúng ta cần xác định

Giao diện Iterator:

Sau đây là hai phương pháp của chúng tôi

```
Giao diện công cộng Iterator  
    boolean hasNext();  
    Đối tượng next();  
}
```

- Phương thức `hasNext()` trả về một giá trị boolean cho biết có nhiều phần tử để lặp lại hay không...

...và phương thức `next()` trả về phần tử tiếp theo.

Và bây giờ chúng ta cần triển khai một Iterator cụ thể hoạt động cho menu Diner

```
lớp công khai DinerMenuItemIterator triển khai Iterator {
    MenuItem[] mục;
    vị trí int = 0;

    công khai DinerMenuItemIterator(MenuItem[] mục) {
        this.items = các mục;
    }

    Đối tượng công khai tiếp theo() {
        MenuItem menuItem = items[vị trí];
        vị trí = vị trí + 1;
        trả về menuItem;
    }

    boolean công khai hasNext() {
        nếu (vị trí >= items.length || items[vị trí] == null)
            trả về false;
        } khác {
            trả về giá trị đúng;
        }
    }

    ←
    ↑
```

mǎng hay chư a và tr
càn lăp lai.

Chúng tôi triển khai
giao diện Iterator.

Hàm tạo sẽ lấy mảng các mục menu mà chúng ta sẽ lặp lại.

Phuơng thức next() trả về phần tử tiếp theo trong mảng và tăng vị trí.

1

Vì đầu bếp của quán ăn đã phân bổ một mảng có kích thước tối đa, nên chúng ta không chỉ cần kiểm tra xem chúng ta có đang ở cuối mảng hay không mà còn phải kiểm tra xem mục tiếp theo có phải là null hay không, điều này cho biết không còn mục nào nữa.

Làm lại Menu Diner với Iterator

Đến đây rồi, chúng ta đã có trình lặp. Đến lúc đưa nó vào DinerMenu; tất cả những gì chúng ta cần làm là thêm một phương thức để tạo DinerMenuIterator và trả về cho máy khách:

```

lớp công khai DinerMenu thực hiện Menu {
    int cuối cùng tĩnh MAX_ITEMS = 6;
    int số mục = 0;
    MenuItem[] menuItems;

    // hàm tạo ở đây

    // thêm mục ở đây

    công khai MenuItem[] getMenuItems() {
        trả về menuItems;
    }

    công khai Iterator createIterator() {
        trả về DinerMenuIterator(menuItems) mới;
    }

    // các phương pháp menu khác ở đây
}

```

Chúng ta sẽ không cần phương thức ~~getMenuItems()~~
nữa và thực tế là chúng ta không muốn sử dụng nó
vì nó làm lộ phần triển khai nội bộ của chúng ta!

Sau đây là phương thức `createIterator()`.
Nó tạo ra một `DinerMenuIterator` từ
mảng `menuItems` và trả về cho máy
khách.

Chúng tôi đang trả về giao diện Iterator. Client
không cần biết `menuItems` được duy trì như thế nào
trong `DinerMenu`, cũng không cần biết
`DinerMenuIterator` được triển khai như thế nào. Nó chỉ
cần sử dụng các trình lặp để duyệt qua các mục trong menu.



Bài tập

Hãy tự mình triển khai `PancakeHouseIterator` và thực hiện những thay đổi cần thiết để đưa nó vào `PancakeHouseMenu`.

cô hầu bàn lặp lại

Sửa mã Waitress

Bây giờ chúng ta cần tích hợp mã lặp vào Waitress.

Chúng ta có thể loại bỏ một số phần dư thừa trong quy trình.

Việc tích hợp khá đơn giản: trước tiên chúng ta tạo phương thức `printMenu()` để lấy một `Iterator`, sau đó chúng ta sử dụng phương thức `createIterator()` trên mỗi menu để lấy `Iterator` và truyền nó cho phương thức mới.



```

lớp công khai Waitress {
    Thực đơn PancakeHouseThực đơn pancakeHouse;
    Thực đơn nhà hàng Thực đơn nhà hàng;

    công cộng Waitress (PancakeHouseMenu pancakeHouseMenu, DinerMenu dinerMenu) {
        this.pancakeHouseMenu = pancakeHouseMenu;
        this.dinerMenu = dinerMenu;
    }

    công khai void printMenu() {
        Trình lặp pancakeIterator = pancakeHouseMenu.createIterator();
        Trình lặp dinerIterator = dinerMenu.createIterator();
        System.out.println("MENU\n----\nBỮA SÁNG");
        printMenu(pancakeIterator);
        System.out.println("\nBỮA TRƯA");
        printMenu(dinerIterator);
    }

    private void printMenu(Iterator iterator) {
        trong khi (iterator.hasNext()) {
            MenuItem menuItem = (MenuItem)iterator.next();
            System.out.print(menuItem.getName() + ", ");
            System.out.print(menuItem.getPrice() + -- " ");
            System.out.println(menuItem.getDescription());
        }
    }
}

// các phương pháp khác ở đây
}

```

Mới và được cải tiến với Iterator.

Trong trình xây dựng, Người phục vụ sẽ lấy hai thực đơn.

Và sau đó gọi phương thức `printMenu()` quá tải với mỗi trình lặp.

Kiểm tra xem còn mục nào nữa không.

Nhận vật phẩm tiếp theo.

Sử dụng sản phẩm để lấy tên, giá và mô tả rồi in chúng.

Phương thức `printMenu()` phương pháp bây giờ tạo ra hai trình lặp, một cho mỗi thực đơn.

Lưu ý rằng chúng ta chỉ còn một vòng lặp.

trình lắp và các mẫu tổng hợp

Kiểm tra mã của chúng tôi

Đã đến lúc thử nghiệm mọi thứ. Hãy viết một số mã thử nghiệm và xem Waitress hoạt động như thế nào...

```
lớp công khai MenuTestDrive {
    public static void main(String args[]) {
        PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
        DinerMenu dinerMenu = new DinerMenu();

        Cô hầu bàn = new Waitress(pancakeHouseMenu, dinerMenu);

        phục vụ bàn.printMenu();
    }
}
```

Đầu tiên chúng ta tạo menu mới.



Sau đó, chúng ta tạo một cô hầu bàn và đưa cho cô ấy thực đơn.

Sau đó chúng tôi in chúng.

Đây là bản chạy thử nghiệm...

```
Cửa sổ chính sửa tệp Trợ giúp GreenEggs&Ham
% java DinerMenuTestDrive
THỰC ĐƠN
-----
BỮA SÁNG
Bữa sáng bánh kếp của K&B, 2,99 đồ la -- Bánh kếp với trứng rán và bánh mì nư ống
Bữa sáng bánh kếp thư ờng, 2,99 -- Bánh kếp với trứng chiên, xúc xích
Bánh kếp việt quất, 3,49 -- Bánh kếp làm từ việt quất tươi
Bánh qué, 3,59 -- Bánh qué, với lựa chọn việt quất hoặc dâu tây của bạn

BỮA TRƯA
BLT chay, 2,99 -- (Giả) Thịt xông khói với xà lách và cà chua trên bánh mì nguyên cám
BLT, 2,99 -- Thịt xông khói với xà lách và cà chua trên bánh mì nguyên cám
Súp trong ngày, 3.29 -- Súp trong ngày, kèm theo salad khoai tây
Hotdog, 3.05 -- Một chiếc hot dog, với dưa cải bắp, tơ ngớt, hành tây, phủ phô mai
Rau hấp và gạo lứt, 3,99 -- Rau hấp trên gạo lứt
Mì Óng, 3,89 -- Mì Ý với sốt Marinara và một lát bánh mì chua

%
```

Đầu tiên chúng ta lắp lại qua bánh kếp thực đơn.

Và sau đó là thực đơn bữa trưa, tất cả đều có như nhau mã lắp lại.

Ưu điểm của trình lặp

Cho đến nay chúng ta đã làm đư ợc gì?

Để bắt đầu, chúng tôi đã làm cho các đầu bếp Objectville của mình rất vui. Họ giải quyết những khác biệt và giữ lại các triển khai của riêng họ. Khi chúng tôi cung cấp cho họ một PancakeHouseIterator và một DinerMenuIterator, tất cả những gì họ phải làm là thêm một phu ơng thức createIterator() và họ đã hoàn thành.

Chúng tôi cũng đã tự giúp mình trong quá trình này. Waitress sẽ dễ dàng hơn nhiều để duy trì và mở rộng sau này. Hãy cùng xem xét chính xác những gì chúng tôi đã làm và suy nghĩ về hậu quả:



Khó bảo trì

Thực hiện Waitress

Các Menu không đư ợc đóng gói tốt; chúng ta có thể thấy Diner đang sử dụng một Mảng và Pancake House đang sử dụng một ArrayList.

Chúng ta cần hai vòng lặp để lặp qua các MenuItem.

Waitress đư ợc liên kết với các lớp cụ thể (MenuItem[] và ArrayList).

Waitress bị ràng buộc với hai lớp Menu cụ thể khác nhau, mặc dù giao diện của chúng gần như giống hệt nhau.

Mới, Hip

Waitress Đư ợc hỗ trợ bởi Iterator

Các triển khai Menu hiện đã đư ợc đóng gói. Waitress không biết Menu lưu trữ bộ sưu tập các mục menu của họ như thế nào.

Tất cả những gì chúng ta cần là một vòng lặp có thể xử lý đa hình bất kỳ tập hợp các mục nào miễn là nó triển khai Iterator.

Người phục vụ hiện sử dụng giao diện (Iterator).

Giao diện Menu hiện tại hoàn toàn giống nhau và, ôi trời, chúng ta vẫn chưa có giao diện chung, điều đó có nghĩa là Waitress vẫn bị ràng buộc với hai lớp Menu cụ thể. Chúng ta nên sửa lỗi đó.

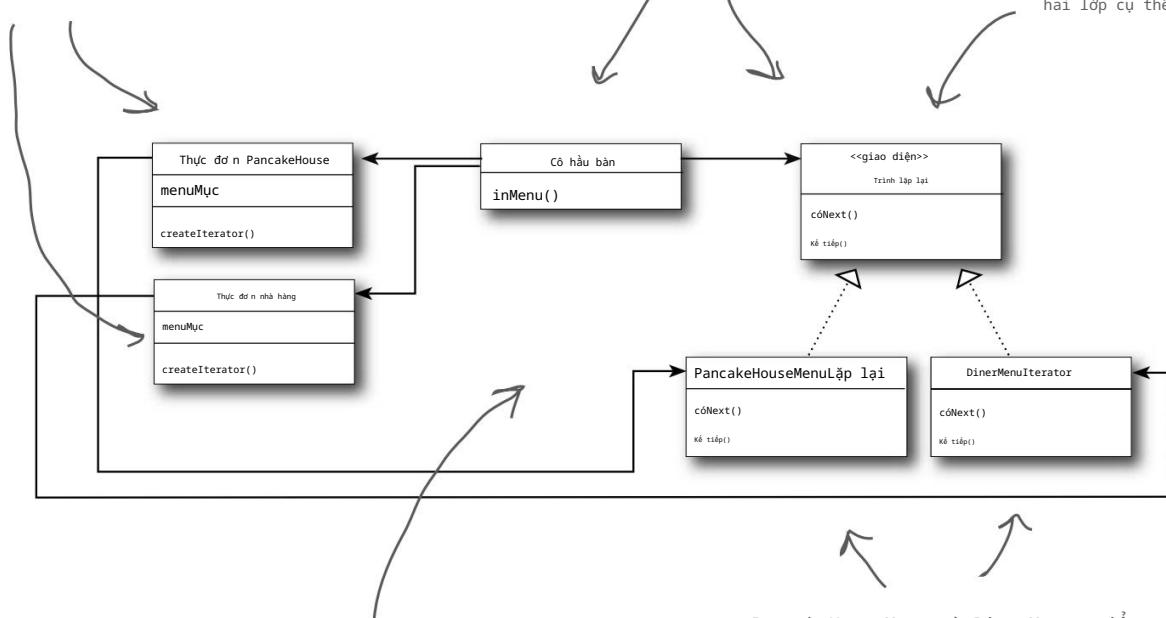
Những gì chúng ta có cho đến nay...

Trước khi dọn dẹp mọi thứ, hãy cùng xem qua thiết kế hiện tại của chúng ta.

Hai menu này triển khai cùng một tập hợp chính xác các phương thức, nhưng chúng không triển khai cùng một Giao diện. Chúng tôi sẽ sửa lỗi này và giải phóng Waitress khỏi mọi sự phụ thuộc vào Menu cụ thể.

Iterator cho phép Waitress tách khỏi việc triển khai thực tế của các lớp cụ thể. Cô ấy không cần biết liệu Menu có được triển khai bằng ArrayList hay ghi chú PostIt hay không. Tất cả những gì cô ấy quan tâm là có thể có Iterator để thực hiện việc lặp lại của mình.

Hiện tại chúng tôi đang sử dụng giao diện Iterator chung và đã triển khai hai lớp cụ thể.



Lưu ý rằng iterator cung cấp cho chúng ta một cách để bước qua các phần tử của một tổng hợp mà không buộc tổng hợp phải làm lộn xộn giao diện của chính nó bằng một loạt các phương thức để hỗ trợ việc duyệt qua các phần tử của nó. Nó cũng cho phép việc triển khai iterator tồn tại bên ngoài tổng hợp; nói cách khác, chúng ta đã đóng gói sự tương tác.

PancakeHouseMenu và DinerMenu triển khai phương thức createIterator() mới; chúng chịu trách nhiệm tạo trình lặp cho các mục menu tương ứng của chúng.

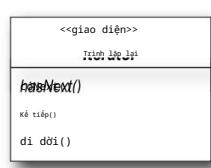
cải thiện trình lặp

Đang thực hiện một số cải tiến...

Được rồi, chúng ta biết giao diện của PancakeHouseMenu và DinerMenu hoàn toàn giống nhau như chúng ta vẫn chưa định nghĩa được giao diện chung cho chúng. Vì vậy, chúng ta sẽ làm điều đó và đơn dẹp Waitress thêm một chút.

Bạn có thể thắc mắc tại sao chúng tôi không sử dụng giao diện Java Iterator - chúng tôi đã làm điều đó để bạn có thể thấy cách xây dựng một trình lặp từ đầu. Bây giờ chúng tôi đã làm điều đó, chúng tôi sẽ chuyển sang sử dụng giao diện Java Iterator, vì chúng tôi sẽ có được nhiều đòn bẩy bằng cách triển khai giao diện đó thay vì giao diện Iterator do chúng tôi tự phát triển. Đòn bẩy như thế nào? Bạn sẽ sớm thấy thôi.

Đầu tiên, chúng ta hãy kiểm tra giao diện java.util.Iterator:



Điều này trông giống hệt như định nghĩa trước của chúng tôi.

Ngoài trừ việc chúng ta có một phương thức bỏ sung cho phép chúng ta xóa phần tử cuối cùng được trả về bởi phương thức next() khỏi tổng hợp.

Đây sẽ là một miếng bánh: Chúng ta chỉ cần thay đổi giao diện mà cả PancakeHouseIterator và DinerIterator đều mở rộng, đúng không? Gần như ... thực ra, thậm chí còn dễ hơn thế nữa. Không chỉ java.util có giao diện Iterator riêng, mà ArrayList còn có phương thức iterator() trả về một iterator. Nói cách khác, chúng ta không bao giờ cần phải triển khai iterator riêng cho ArrayList. Tuy nhiên, chúng ta vẫn cần triển khai cho DinerMenu vì nó dựa trên một Array, không hỗ trợ phương thức iterator() (hoặc bất kỳ cách nào khác để tạo một iterator mảng).

không có Những câu hỏi ngớ ngẩn

Q: Nếu tôi không muốn cung cấp thì sao?
khả năng loại bỏ một thứ gì đó khỏi
tập hợp các đối tượng cơ bản?

A: Phương thức remove() được xem xét tùy chọn. Bạn không phải cung cấp chức năng xóa. Nhưng rõ ràng là bạn cần cung cấp phương thức vì nó là một phần của giao diện Iterator. Nếu bạn không cho phép remove() trong trình lặp của mình, bạn sẽ muôn ném

ngoại lệ thời gian chạy java.lang.UnsupportedOperationException. Tài liệu API Iterator chỉ rõ rằng ngoại lệ này có thể được đưa ra từ remove() và bất kỳ máy khách nào là công dân tốt sẽ kiểm tra ngoại lệ này khi gọi phương thức remove().

Q: remove() hoạt động như thế nào trong nhiều luồng có thể sử dụng các trình lặp khác nhau trên cùng một tập hợp các đối tượng?

A: Hành vi của remove() là không xác định nếu bộ sưu tập thay đổi trong khi bạn đang lặp lại nó. Vì vậy, bạn nên cẩn thận khi thiết kế mã đa luồng của riêng mình khi truy cập đồng thời vào bộ sưu tập.

Dọn dẹp mọi thứ bằng java.util.Iterator

Chúng ta hãy bắt đầu với PancakeHouseMenu, việc thay đổi nó thành java.util.Iterator sẽ dễ dàng.

Chúng ta chỉ cần xóa lớp PancakeHouseIterator, thêm lệnh import java.util.Iterator vào đầu PancakeHouseMenu và thay đổi một dòng của PancakeHouseMenu:

```
công khai Iterator createIterator() {
    trả về menuItems.iterator();
}
```



Thay vì tạo trình lắp riêng ngay bây giờ, chúng ta chỉ cần gọi phương thức iterator() trên ArrayList của menuItems.

Và thế là PancakeHouseMenu đã hoàn thành.

Bây giờ chúng ta cần thực hiện các thay đổi để cho phép DinerMenu hoạt động với java.util.Iterator.

```
nhập java.util.Iterator;
```



Đầu tiên chúng ta import java.util.Iterator, giao diện mà chúng ta sẽ triển khai.

```
lớp công khai DinerMenuIterator triển khai Iterator {
    MenuItem[] danh sách;
    vị trí int = 0;

    công khai DinerMenuIterator(MenuItem[] danh sách) {
        this.list = danh sách;
    }

    Đối tượng công khai tiếp theo() {
        //thực hiện ở đây
    }

    boolean công khai hasNext() {
        //thực hiện ở đây
    }

    công khai void remove() {
        nếu (vị trí <= 0) {
            ném IllegalStateException mới
                ("Bạn không thể xóa một mục cho đến khi bạn đã thực hiện ít nhất một mục tiếp theo()");
        }
        nếu (danh sách[vị trí-1] != null) {
            đổi với (int i = vị trí-1; i < (danh sách.chiều dài-1); i++) {
                danh sách[i] = danh sách[i+1];
            }
            danh sách[danh sách.length-1] = null;
        }
    }
}
```

Không có thay đổi nào trong quá trình triển khai hiện tại của chúng tôi...

...nhưng chúng ta cần phải triển khai remove().
Ở đây, vì đầu bếp đang sử dụng một Mảng có kích thước cố định nên chúng ta chỉ cần dịch chuyển tất cả các phần tử lên một phần tử khi gọi remove().



tách cõi hầu bàn ra khỏi thực đơn

Chúng ta sắp tới nơi rồi...

Chúng ta chỉ cần cung cấp cho Menus một giao diện chung và chỉnh sửa Waitress một chút. Giao diện Menu khá đơn giản: chúng ta có thể muốn thêm một vài thứ gì đó nữa vào đó sau này, như addItem(), nhưng hiện tại chúng ta sẽ để các đầu bếp kiểm soát menu của họ bằng cách giữ phuơng thức đó ngoài giao diện công khai:

```
giao diện công cộng Menu {
    công khai Iterator createIterator();
}
```

Đây là một giao diện đơn giản cho phép khách hàng có được trình lặp cho các mục trong menu.

Bây giờ chúng ta cần thêm một Menu thực hiện vào cả hai
Định nghĩa lớp PancakeHouseMenu và DinerMenu và cập nhật Waitress:

```
nhập java.util.Iterator;
```

Bây giờ Waitress cũng sử dụng java.util.Iterator.

```
lớp công khai Waitress {
    Thực đơn pancakeHouseMenu;
    Thực đơn thực đơn;

    công cộng Waitress (Menu pancakeHouseMenu, Menu dinerMenu) {
        this.pancakeHouseMenu = pancakeHouseMenu;
        this.dinerMenu = dinerMenu;
    }

    công khai void printMenu() {
        Trình lặp pancakeIterator = pancakeHouseMenu.createIterator();
        Trình lặp dinerIterator = dinerMenu.createIterator();
        System.out.println("MENU\n----\nBỮA SÁNG");
        printMenu(pancakeIterator);
        System.out.println("\nBỮA TRƯA");
        printMenu(dinerIterator);
    }

    private void printMenu(Iterator iterator) {
        trong khi (iterator.hasNext()) {
            MenuItem menuItem = (MenuItem)iterator.next();
            System.out.print(menuItem.getName() + ", ");
            System.out.print(menuItem.getPrice() + " - ");
            System.out.println(menuItem.getDescription());
        }
    }

    // các phuơng pháp khác ở đây
}
```

Chúng ta cần thay thế các lớp Menu cụ thể bằng Giao diện Menu.

Không có gì thay đổi đây.

trình lặp và các mẫu tổng hợp

Điều này mang lại cho chúng ta điều gì?

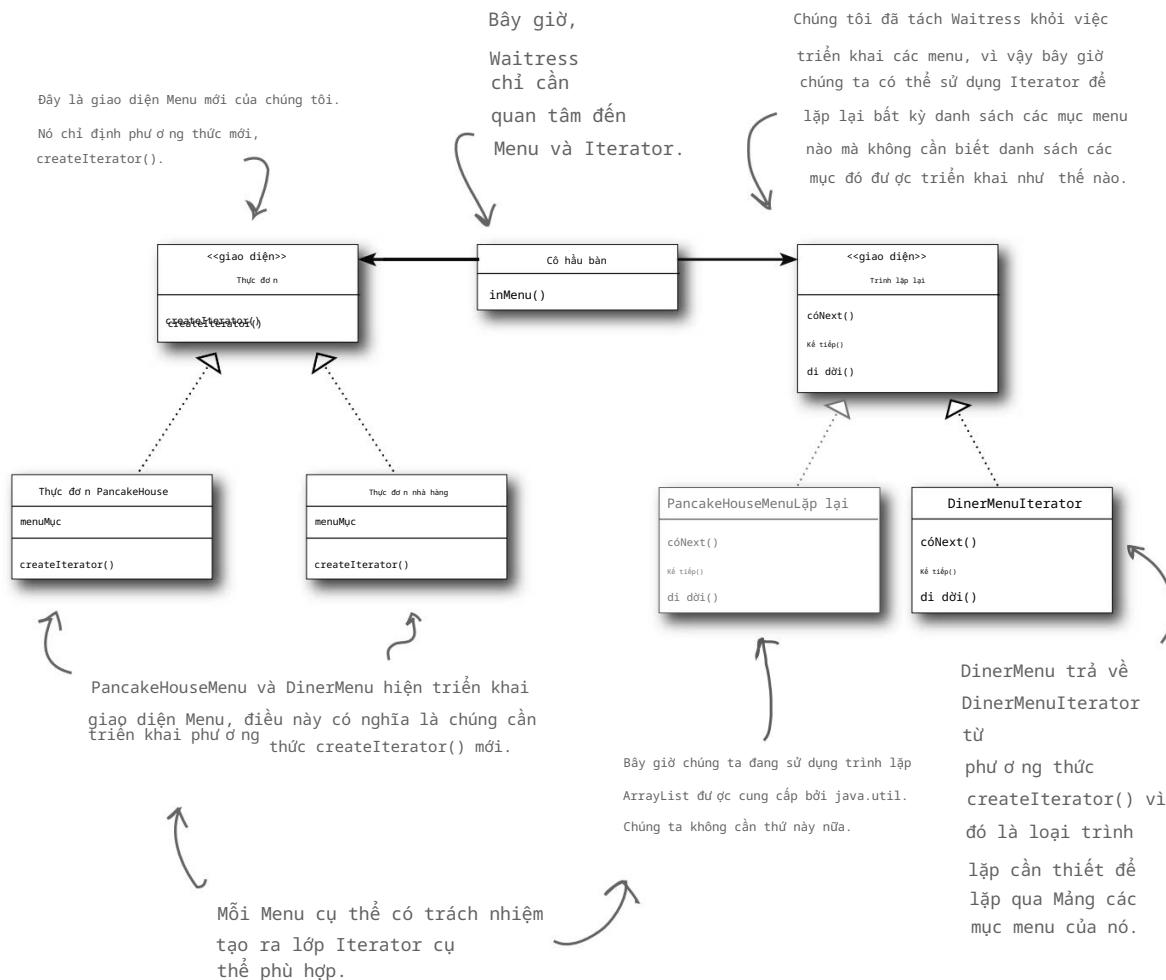
Các lớp PancakeHouseMenu và DinerMenu triển khai một giao diện, Menu.

Waitress có thể tham chiếu đến từng đối tượng menu bằng giao diện thay vì lớp cụ thể. Vì vậy, chúng ta đang giảm sự phụ thuộc giữa Waitress và các lớp cụ thể bằng cách "lập trình vào một giao diện, không phải là một triển khai".

Giao diện Menu mới có một phương thức, `createIterator()`, được triển khai bởi PancakeHouseMenu và DinerMenu. Mỗi lớp menu đảm nhiệm trách nhiệm tạo một Iterator cụ thể phù hợp với việc triển khai nội bộ các mục menu của nó.

Điều này giải quyết vấn đề của Người phục vụ phụ thuộc vào Thực đơn cụ thể.

Điều này giải quyết vấn đề của Waitress tùy thuộc vào cách triển khai MenuItem.



mẫu lặp đư ợc xác định

Mẫu Iterator đư ợc định nghĩa

Bạn đã thấy cách triển khai Iterator Pattern với iterator của riêng bạn. Bạn cũng đã thấy cách Java hỗ trợ iterator trong một số lớp hưu ứng tập hợp của nó (ArrayList). Đây giờ là lúc để kiểm tra định nghĩa chính thức của mẫu:

Mẫu Iterator cung cấp một cách để truy cập các phần tử của một đối tượng tổng hợp theo trình tự mà không cần phải hiển thị biểu diễn cơ bản của nó.

Điều này rất có ý nghĩa: mẫu cung cấp cho bạn một cách để bù ớc qua các phần tử của một tổng hợp mà không cần phải biết cách mọi thứ được biểu diễn bên dưới. Bạn đã thấy điều đó với hai triển khai của Menus. Nhưng hiệu ứng của việc sử dụng trình lặp trong thiết kế của bạn cũng quan trọng không kém: khi bạn có một cách thống nhất để truy cập các phần tử của tất cả các đối tượng tổng hợp của mình, bạn có thể viết mã đa hình hoạt động với bất kỳ

của các tập hợp này - giống như phương thức printMenu(), không quan tâm đến việc các mục menu được giữ trong một Mảng hay ArrayList (hoặc bất kỳ thứ gì khác có thể tạo ra một Iterator), miễn là nó có thể giữ đư ợc một Iterator.

Tác động quan trọng khác lên thiết kế của bạn là Iterator Pattern đảm nhiệm việc duyệt qua các phần tử và trao trách nhiệm đó cho đối tượng iterator, không phải đối tượng tổng hợp. Điều này không chỉ giúp giao diện và triển khai tổng hợp đơn giản hơn mà còn loại bỏ trách nhiệm lặp lại khỏi tổng hợp và giữ cho tổng hợp tập trung vào những thứ mà nó cần tập trung vào (quản lý một tập hợp các đối tượng), không phải vào việc lặp lại.

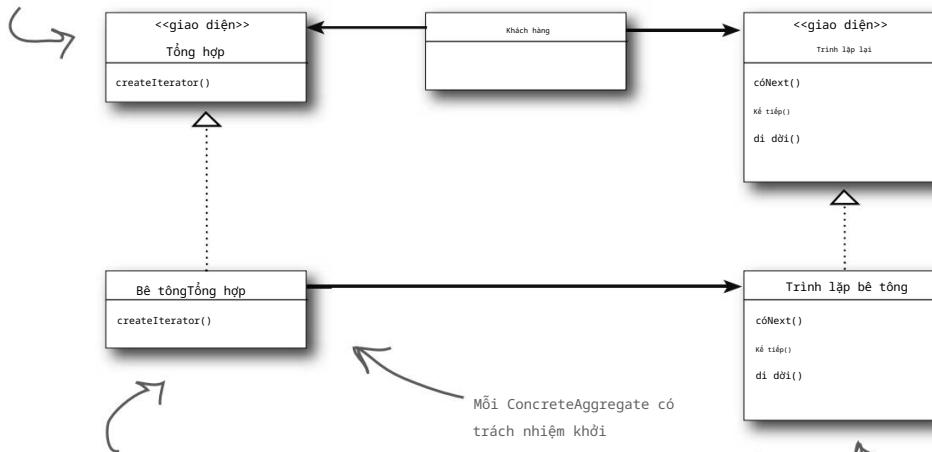
Chúng ta hãy kiểm tra sơ đồ lớp để đưa a tất cả các phần vào bối cảnh...

Mẫu Iterator cho phép duyệt qua các phần tử của một tổng hợp mà không làm lộ phần triển khai cơ bản.

Nó cũng đặt nhiệm vụ duyệt qua cho đối tượng lặp, không phải cho tổng hợp, điều này giúp đơn giản hóa giao diện và triển khai tổng hợp, đồng thời đặt trách nhiệm vào đúng nơi cần thiết.

Việc có một giao diện chung cho các tổng hợp của bạn rất tiện lợi cho khách hàng của bạn; nó tách biệt khách hàng của

bạn khỏi việc triển khai bộ sưu tập đối tượng của bạn.



ConcreteAggregate có
một tập hợp các
đối tượng và triển
khai phong
thức trả về
Iterator cho tập hợp đó.

Mỗi ConcreteAggregate có
trách nhiệm khởi
tạo một
ConcreteIterator có thể
lặp lại bộ sưu tập các
đối tượng của nó.

ConcreteIterator chịu
trách nhiệm quản lý
vị trí hiện tại của
lần lặp.

Giao diện Iterator cung cấp

giao diện mà tất cả các
trình lặp phải triển
khai và một tập hợp các
phóng thức để

duyệt qua các phần tử
của một tập hợp.

Ở đây chúng ta sử dụng
java.util.Iterator. Nếu bạn
không muốn sử dụng giao diện
Iterator của Java, bạn luôn
có thể tạo
sở hữu.

não Apower

Biểu đồ lớp cho Iterator Pattern trông rất giống với một Pattern khác mà bạn đã học; bạn có thể nghĩ ra nó là gì không? Gợi ý: Một lớp con quyết định đối tượng nào sẽ được tạo.

hỏi đáp về iterator

không có Những câu hỏi ngắn

H: Tôi đã thấy những cuốn sách khác cho thấy Sơ đồ lớp Iterator với các phương thức `first()`, `next()`, `isDone()` và `currentItem()`. Tại sao các phương pháp này lại khác nhau?

A: Đó là phương pháp "cổ điển" tên đã được sử dụng. Những tên này đã thay đổi theo thời gian và hiện tại chúng ta có `next()`, `hasNext()` và thậm chí `remove()` trong `java.util.Iterator`.

Chúng ta hãy xem xét các phương pháp cổ điển. `next()` và `currentItem()` đã được hợp nhất thành một phương thức trong `java.util`. Phương thức `isDone()` rõ ràng đã trở thành `hasNext()`; nhưng chúng ta không có phương thức nào tương ứng với `first()`. Đó là vì trong Java, chúng ta có xu hướng chỉ lấy một trình lặp mới bắt đầu khi nào chúng ta cần bắt đầu duyệt lại. Tuy nhiên, bạn có thể thấy có rất ít sự khác biệt trong các giao diện này. Trên thực tế, có một loạt các hành vi mà bạn có thể cung cấp cho các trình lặp của mình. Phương thức `remove()` là một ví dụ về phần mở rộng trong `java.util.Iterator`.

H: Tôi đã nghe nói về "nội bộ" các trình lặp và các trình lặp "bên ngoài". Cái gì chúng là gì? Chúng ta đã triển khai loại nào trong ví dụ?

A: Chúng tôi đã triển khai một trình lặp bên ngoài, có nghĩa là khách hàng kiểm soát lặp lại bằng cách gọi `next()` để lấy phần tử tiếp theo. Một trình lặp nội bộ được kiểm soát bởi chính trình lặp. Trong trường hợp đó, vì trình lặp là trình bù ốc qua các phần tử, bạn phải cho trình lặp biết phải làm gì với các phần tử đó khi nó đi qua chúng. Điều đó có nghĩa là bạn cần một cách để truyền một thao tác cho trình lặp. Trình lặp nội bộ kém linh hoạt hơn trình lặp bên ngoài vì máy khách không có quyền kiểm soát việc lặp lại. Tuy nhiên, một số người có thể lập luận

rằng chúng dễ sử dụng hơn vì bạn chỉ cần đưa cho chúng một thao tác và bảo chúng lặp lại, và chúng sẽ làm mọi công việc thay bạn.

bộ sưu tập miễn là nó hỗ trợ `Iterator`. Chúng tôi không quan tâm đến cách thu thập dữ liệu triển khai, chúng ta vẫn có thể viết mã để lặp lại nó.

Q: Tôi có thể triển khai một `Iterator` không? có thể đi lùi cũng như đi tới không?

A: Chắc chắn rồi. Trong trường hợp đó, bạn sẽ có lẽ muốn thêm hai phương thức, một để đến phần tử trước đó và một để cho bạn biết khi nào bạn đang ở đầu tập hợp các phần tử. Bộ sưu tập của Java

Framework cung cấp một loại giao diện lặp khác được gọi là `ListIterator`. Trình lặp này thêm `previous()` và một vài phương thức khác vào giao diện `Iterator` chuẩn. Nó là được hỗ trợ bởi bất kỳ Bộ sưu tập nào triển khai giao diện `Danh sách`.

H: Nếu tôi sử dụng Java, tôi sẽ không phải luôn luôn muốn sử dụng giao diện `java.util.Iterator` để tôi có thể sử dụng

các triển khai trình lặp của riêng mình với các lớp đang sử dụng trình lặp Java không?

A: Có lẽ vậy. Nếu bạn có một điểm chung Giao diện `Iterator`, chắc chắn sẽ giúp bạn dễ dàng kết hợp và khớp các tập hợp của riêng bạn với các tập hợp Java như `ArrayList` và `Vector`. Nhưng hãy nhớ rằng, nếu bạn cần thêm chức năng vào giao diện `Iterator` cho các tập hợp của mình, bạn luôn có thể mở rộng giao diện `Iterator`.

Q: Ai định nghĩa thứ tự của lặp lại trong một bộ sưu tập như `Hashtable`, vốn không có thứ tự?

A: Các trình lặp không có thứ tự. Các bộ sưu tập cơ bản có thể không được sắp xếp như trong một `Hashtable` hoặc trong một `bag`; chúng thậm chí có thể chứa các bản sao. Vì vậy, việc sắp xếp có liên quan đến cả các thuộc tính của bộ sưu tập cơ bản và việc triển khai. Nhìn chung, bạn không nên đưa ra bất kỳ giả định nào về việc sắp xếp trừ khi tài liệu Collection chỉ ra điều ngược lại.

H: Bạn đã nói chúng ta có thể viết "mã da hình" sử dụng trình lặp; bạn có thể giải thích thêm không?

A: Khi chúng ta viết các phương pháp thực hiện Các trình lặp như các tham số, chúng tôi đang sử dụng phép lặp đa hình. Điều đó có nghĩa là chúng tôi đang tạo mã có thể lặp qua bất kỳ

Q: Tôi đã thấy một `Enumeration` giao diện trong Java; liệu nó có triển khai `Mẫu lặp lại` không?

A: Chúng tôi đã nói về điều này trong Bộ chuyên đề `Chương`. Nhớ không? `java.util`. `Enumeration` là một triển khai cũ hơn của `Iterator` đã được thay thế bằng `java.util.Iterator`. `Enumeration` có hai phương thức, `hasMoreElements()`, tương ứng với `hasNext()` và `nextElement()`, tương ứng với `next()`. Tuy nhiên, bạn có thể muốn sử dụng `Iterator` thay vì `Enumeration` vì có nhiều lớp Java hơn hỗ trợ nó. Nếu bạn cần chuyển đổi từ cái này sang cái khác, hãy xem lại `Chương` Bộ điều hợp, nơi bạn đã triển khai bộ điều hợp cho `Enumeration` và `Iterator`.

trình lắp và các mẫu tổng hợp

Trách nhiệm duy nhất

Nếu chúng ta cho phép các tổng hợp của mình triển khai các tập hợp nội bộ và các hoạt động liên quan VÀ các phư ơng thức lắp thì sao? Vâng, chúng ta đã biết rằng điều đó sẽ mở rộng số lượng các phư ơng thức trong tổng hợp, như ng vậy thì sao? Tại sao điều đó lại tệ đến vậy?

Vâng, để hiểu lý do, trước tiên bạn cần nhận ra rằng khi chúng ta cho phép một lớp không chỉ tự xử lý công việc của mình (quản lý một số loại tổng hợp) mà còn đảm nhận nhiều trách nhiệm hơn (như lắp lại) thì chúng ta đã đưa ra cho lớp đó hai lý do để thay đổi.

Hai? Vâng, hai: nó có thể thay đổi nếu bộ sưu tập thay đổi theo một cách nào đó, và nó có thể thay đổi nếu cách chúng ta lắp lại thay đổi. Vì vậy, một lần nữa, người bạn THAY ĐỔI của chúng ta lại là trung tâm của một nguyên tắc thiết kế khác:



Nguyên lý thiết kế

Mỗi lớp chỉ nên có một lý do để thay đổi.

Chúng tôi biết rằng chúng tôi muốn tránh thay đổi trong một lớp như tránh bệnh dịch hạch - việc sửa đổi mã tạo ra đủ mọi cơ hội để các vấn đề phát sinh. Có hai cách để thay đổi làm tăng khả năng lớp sẽ thay đổi trong tương lai và khi thay đổi, nó sẽ ảnh hưởng đến hai khía cạnh trong thiết kế của bạn.

Giải pháp là gì? Nguyên tắc hướng dẫn chúng ta giao mỗi trách nhiệm cho một lớp và chỉ một lớp mà thôi.

Đúng vậy, dễ như vậy, nhưng cũng không phải vậy: việc phân chia trách nhiệm trong thiết kế là một trong những việc khó khăn nhất. Bộ não của chúng ta quá giỏi trong việc nhìn thấy một tập hợp các hành vi và nhóm chúng lại với nhau ngay cả khi thực tế có hai hoặc nhiều trách nhiệm. Cách duy nhất để thành công là phải siêng năng kiểm tra các thiết kế của bạn và chú ý đến các tín hiệu cho thấy một lớp đang thay đổi theo nhiều cách khi hệ thống của bạn phát triển.

Mỗi trách nhiệm của một lớp là một lĩnh vực có khả năng thay đổi. Nhiều hơn một trách nhiệm có nghĩa là nhiều hơn một lĩnh vực có khả năng thay đổi.

Nguyên tắc này hướng dẫn chúng ta giao mỗi lớp một trách nhiệm duy nhất.



Sự gắn kết là một thuật ngữ bạn sẽ nghe thấy được sử dụng như một thư ớc do mức độ chặt chẽ mà một lớp hoặc một mô-đun hỗ trợ cho một mục đích hoặc trách nhiệm duy nhất.

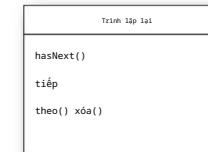
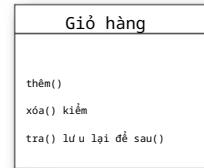
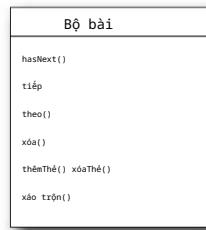
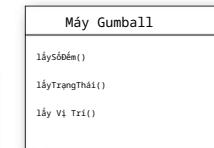
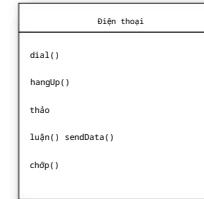
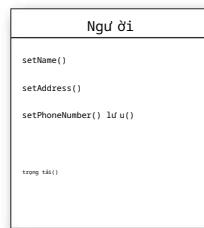
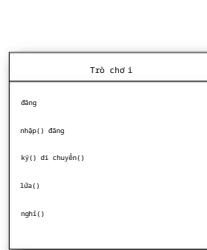
Chúng ta nói rằng một mô-đun hoặc lớp có tính gắn kết cao khi nó được thiết kế xung quanh một tập hợp các chức năng có liên quan, và chúng ta nói rằng nó có tính gắn kết thấp khi nó được thiết kế xung quanh một tập hợp các chức năng không liên quan.

Sự gắn kết là một khái niệm tổng quát hơn. Nguyên tắc trách nhiệm duy nhất, nhưng cả hai có liên quan chặt chẽ với nhau. Các lớp học tuân thủ nguyên tắc này có xu hướng có tính gắn kết cao và dễ duy trì hơn các lớp học đảm nhiệm nhiều trách nhiệm và có tính gắn kết thấp.

nhiều trách nhiệm

não Apower

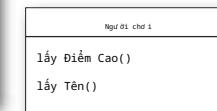
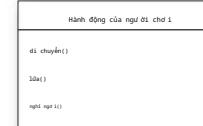
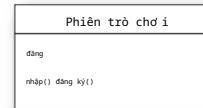
Kiểm tra các lớp này và xác định lớp nào có nhiều trách nhiệm.

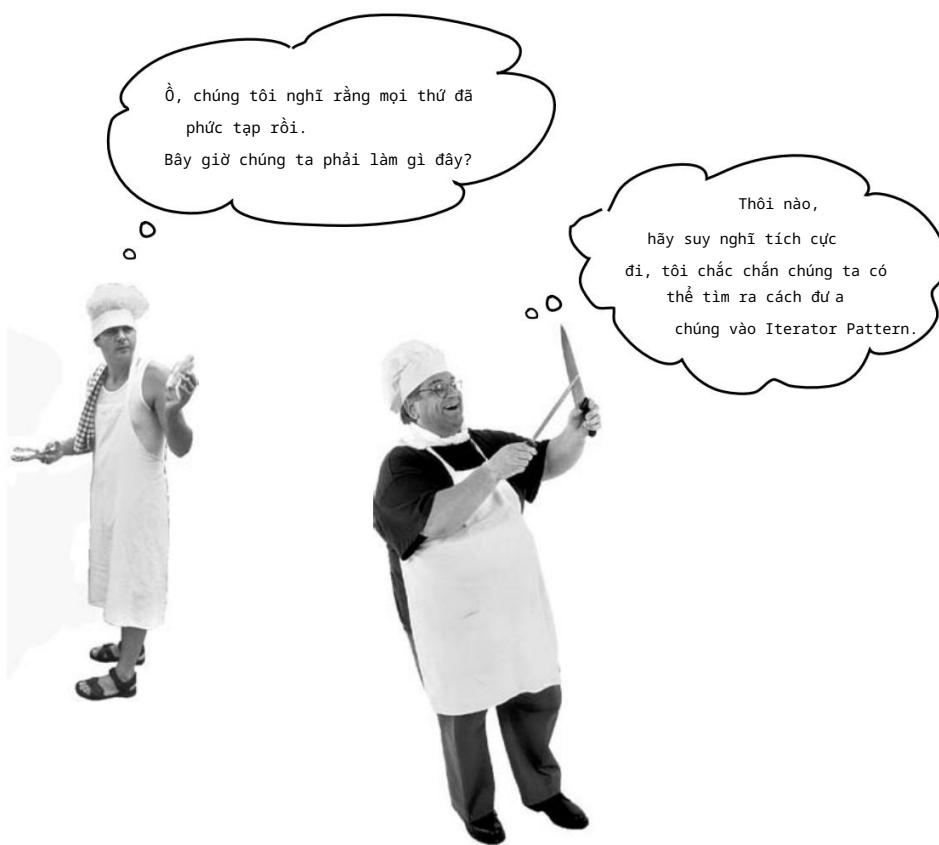
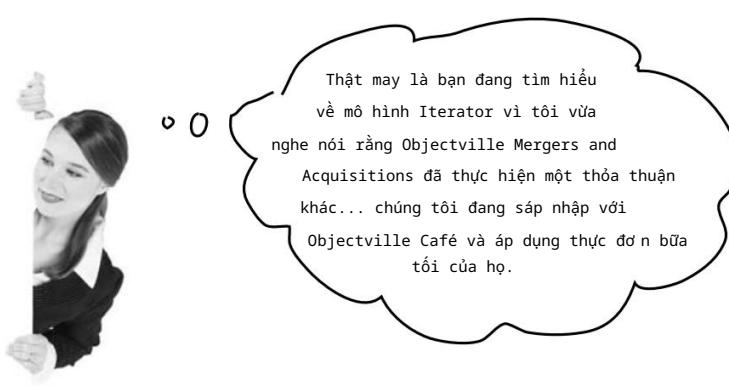


Khu vực mũ cứng, hãy cẩn thận
cho các già định giàm

não Apower²

Xác định xem các lớp này có tính gắn kết thấp hay cao.





một thực đơn mới

Nhìn vào Menu của Café

Đây là Menu Café. Có vẻ như không quá khó để tích hợp Menu Café vào khuôn khổ của chúng ta... hãy cùng xem thử nhé.

```

CafeMenu không triển khai giao diện Menu mới của chúng
tôi, nhưng vấn đề này có thể dễ dàng khắc phục.

lớp công khai CafeMenu thực hiện{Menu {
    menuItems của bảng băm = new Hashtable();

    công khai CafeMenu() {
        addItem("Veggie Burger và Khoai tây chiên",
            "Bánh mì kẹp chay trên bánh mì nguyên cám, rau diếp, cà chua và khoai tây chiên",
            đúng, 3,99);
        addItem("Súp trong ngày",
            "Một cốc súp trong ngày, kèm theo một đĩa salad",
            sai, 3,69);
        addItem("Burrito",
            "Một chiếc burrito lớn, với đậu pinto nguyên hạt, sốt salsa, guacamole",
            đúng, 4.29);
    }

    public void addItem(String name, String description,
                        boolean chay, giá gấp đôi)
    {
        MenuItem menuItem = new MenuItem(tên, mô tả, đồ chay, giá);
        menuItems.put(menuItem.getName(), menuItem);
    }
}

công khai Hashtable getItems() {
    trả về menuItems;
}
}

Quán cà phê đang lưu trữ các mục trong thực đơn của mình trong Hashtable.
Nó có hỗ trợ Iterator không? Chúng ta sẽ xem xét ngay thôi...

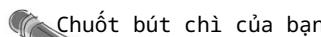
```

Giống như các Menu khác, các mục menu
được khởi tạo trong hàm tạo.

Đây là nơi chúng ta tạo một MenuItem mới và thêm
nó vào bảng băm menuItems.

chìa khóa là tên mục. giá trị là đối tượng menuItem.

Chúng ta sẽ không cần thử này nữa.



Chuốt bút chì của bạn

Trước khi xem trang tiếp theo, hãy nhanh chóng ghi ra ba điều chúng ta phải làm với
đoạn mã này để phù hợp với khuôn khổ của chúng ta:

1.

2.

3.

Làm lại mã Menu Café

Tích hợp Cafe Menu vào framework của chúng tôi rất dễ. Tại sao? Bởi vì Hashtable là một trong những bộ sưu tập Java hỗ trợ Iterator. Nhưng nó không giống hoàn toàn với ArrayList...

```

lớp công khai CafeMenu thực hiện Menu {
    MenuItem[] menuItems = new Hashtable();
    công khai CafeMenu() {
        // mã xây dựng ở đây
    }
    public void addItem(String name, String description,
                        boolean chay, giá gấp đôi)
    {
        MenuItem menuItem = new MenuItem(tên, mô tả, đồ chay, giá);
        menuItems.put(menuItem.getName(), menuItem);
    }
    công khai Hashtable getItems() {
        trả về menuItems;
    }
    công khai Iterator createIterator() {
        trả về menuItems.values().iterator();
    }
}

```

CafeMenu triển khai giao diện Menu, do đó Nhân viên phục vụ có thể sử dụng nó giống như hai Menu kia.

Chúng tôi sử dụng Hashtable vì đây là cấu trúc dữ liệu phổ biến để lưu trữ giá trị; bạn cũng có thể sử dụng HashMap mới hơn.

Giống như trước, chúng ta có thể loại bỏ getItems() để không hiển thị việc triển khai menuItems cho Waitress.

Và đây là nơi chúng ta triển khai phương thức createIterator(). Lưu ý rằng chúng ta không nhận được Iterator cho toàn bộ Hashtable, mà chỉ cho các giá trị.



Mã Gắn

Hashtable phức tạp hơn một chút so với ArrayList vì nó hỗ trợ cả khóa và giá trị, nhưng chúng ta vẫn có thể có được Iterator cho các giá trị (là các MenuItem).

```

công khai Iterator createIterator() {
    trả về menuItems.values().iterator();
}

```

Đầu tiên, chúng ta lấy các giá trị của Hashtable, đây chỉ là tập hợp tất cả các đối tượng trong hashtable.

May mắn thay, bộ sưu tập đó hỗ trợ phương thức iterator(), trả về một đối tượng có kiểu java.util.Iterator.

thử nghiệm menu mới

Thêm Menu Quán Cà Phê vào Waitress

Thật dễ dàng; tại sao không thay đổi Waitress để hỗ trợ Menu mới của chúng ta?

Bây giờ Waitress mong đợi Iterators, điều đó cũng dễ dàng.

```
lớp công khai Waitress {
    Thực đơn pancakeHouseMenu;
    Thực đơn thực đơn;
    Thực đơn cafeMenu;
```

Menu Café được truyền vào Waitress trong hàm
khởi tạo cùng với các menu khác và chúng tôi
lưu trữ nó trong một biến thể hiện.

```
công cộng Waitress (Menu pancakeHouseMenu, Menu dinerMenu, Menu cafeMenu) {
    this.pancakeHouseMenu = pancakeHouseMenu;
    this.dinerMenu = dinerMenu;
    this.cafeMenu = cafeMenu;
}
```

```
công khai void printMenu() {
    Trình lặp pancakeIterator = pancakeHouseMenu.createIterator();
    Trình lặp dinerIterator = dinerMenu.createIterator();
    Trình lặp cafeIterator = cafeMenu.createIterator();
    System.out.println("MENU\n---\nBỮA SÁNG");
    printMenu(pancakeIterator);
    System.out.println("\nBỮA TRƯA");
    printMenu(dinerIterator);
    System.out.println("\nBỮA TỐI");
    printMenu(cafeIterator);
}
```

Chúng tôi sử dụng quán Café
thực đơn cho thực đơn bữa tối.

Tất cả những gì chúng ta phải làm
để in nó là tạo trình lặp và
truyền nó vào printMenu().
Vậy là xong!

```
private void printMenu(Iterator iterator) {
    trong khi (iterator.hasNext()) {
        MenuItem menuItem = (MenuItem)iterator.next();
        System.out.print(menuItem.getName() + ", ");
        System.out.print(menuItem.getPrice() + " - ");
        System.out.println(menuItem.getDescription());
    }
}
```

Không có gì thay đổi ở đây

trình lặp và các mẫu tổng hợp

Bữa sáng, bữa trưa và bữa tối

Hãy cập nhật bản thử nghiệm để đảm bảo mọi thứ đều hoạt động tốt.

```
lớp công khai MenuTestDrive { công khai
    tĩnh void main(String args[]) {
        PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
        DinerMenu dinerMenu = new DinerMenu();
        CafeMenu cafeMenu = mới CafeMenu(); ← Tạo một CafeMenu...
        Cô hầu bàn = new Waitress(pancakeHouseMenu, dinerMenu, cafeMenu); ← ... và đưa nó cho cô hầu bàn.

        phục vụ bàn.printMenu(); ← Bay giờ, khi in chúng ta sẽ thấy cả ba menu.
    }
}
```

Đây là bản thử nghiệm; hãy xem thực đơn bữa tối mới của Café!

```
Cửa sổ chính sửa tệp Trợ giúp Kathy&BertLikePancakes
% java DinerMenuTestDrive
THỰC ĐƠN
-----
BỮA SÁNG
Bữa sáng bánh kếp của K&B, 2,99 đô la -- Bánh kếp với trứng rán và bánh mì nướng Bữa sáng bánh kếp
thông thường, 2,99 đô la -- Bánh kếp với trứng rán, xúc xích Bánh kếp việt quất, 3,49 đô la --
Bánh kếp làm từ việt quất tươi Bánh quê, 3,59 đô la -- Bánh quê, với việt quất hoặc dâu
tây tùy chọn ← Đầu tiên chúng ta lặp
lại qua bánh kếp
thực đơn. ← Và sau đó là
quán ăn
thực đơn.

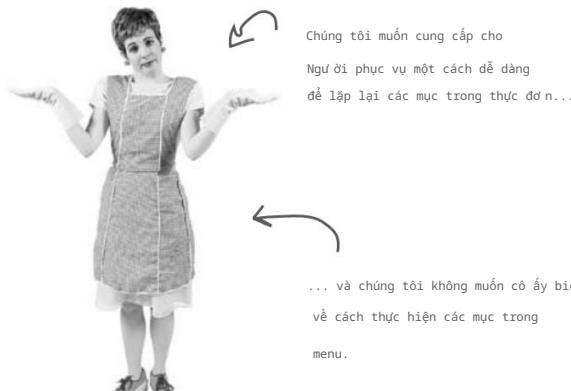
BỮA TRƯA
BLT chay, 2,99 -- (Giả) Thịt xông khói với xà lách & cà chua trên bánh mì nguyên cám BLT, 2,99 -- Thịt
xông khói với xà lách & cà chua trên bánh mì nguyên cám Súp trong ngày, 3,29
-- Súp trong ngày, ăn kèm với salad khoai tây Hotdog, 3,05 -- Một chiếc xúc xích, với dưa cải
bắp, tương ớt, hành tây, phủ phô mai Rau hấp và gạo lứt, 3,99 -- Rau hấp trên gạo lứt Mì ống, 3,89 -- Mì Ý
sốt Marinara và một lát bánh mì chua ← Và cuối cùng là quán
cà phê mới, thực đơn,
tất cả đều sử
dụng cùng một mã
lập.

BỮA TỐI
Súp trong ngày, 3,69 đô la -- Một cốc súp trong ngày, kèm theo một ổ bánh Burrito, 4,29 đô la -- Một chiếc
burrito lớn, với đậu pinto nguyên hạt, sốt salsa, sốt guacamole Veggie Burger và Air Fries, 3,99 đô la -- Veggie
burger trên một chiếc bánh mì nguyên cám,
rau diếp, cà chua và khoai tây chiên % ← Và cuối cùng là quán
cà phê mới, thực đơn,
tất cả đều sử
dụng cùng một mã
lập.
```

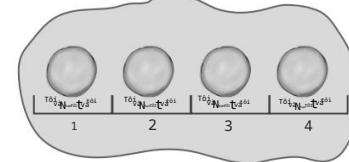
chúng tôi đã làm gì?

Chúng tôi đã làm gì?

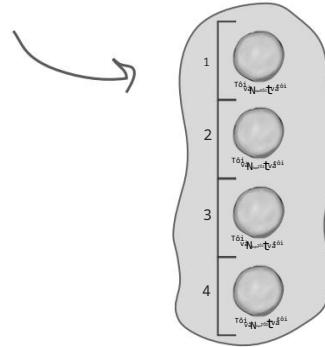
MảngDanh sách



Các mục menu của chúng tôi
có hai cách triển khai khác
nhau và hai giao diện
lặp lại khác nhau.



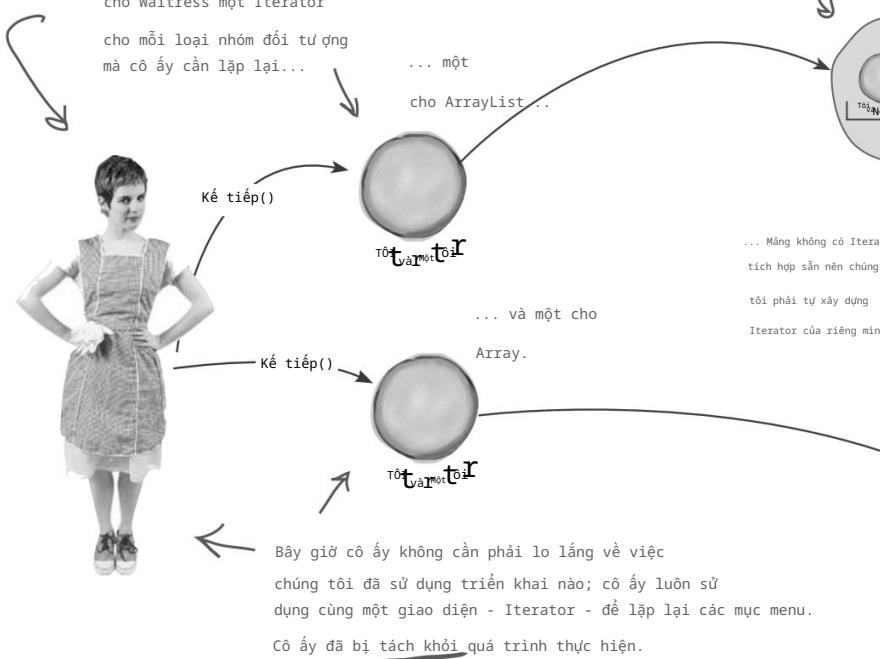
Mảng



Chúng tôi đã tách cô hầu bàn ra....

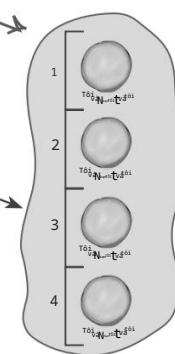
ArrayList có trình
lập tích hợp sẵn...

MảngDanh sách

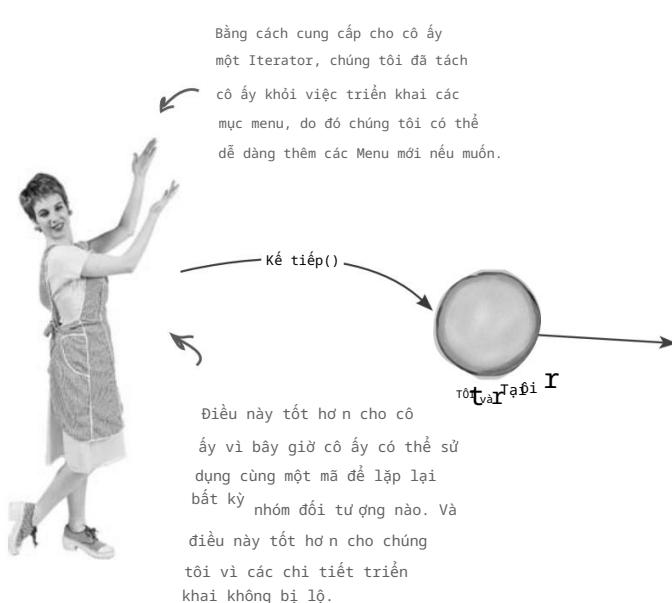


... Mảng không có Iterator
tích hợp sẵn nên chúng
tôi phải tự xây dựng
Iterator của riêng mình.

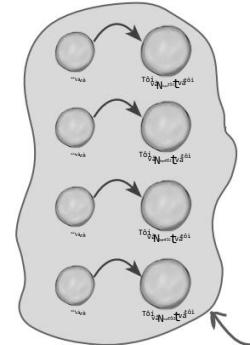
Mảng



... và chúng tôi đã làm cho Waitress có thể mở rộng hơn



Chúng tôi dễ dàng thêm một triển khai khác cho các mục menu và vì chúng tôi cung cấp Iterator nên Waitress biết phải làm gì.



Việc tạo một Iterator
cho các giá trị

Hashtable rất dễ dàng;
khi bạn gọi
values.iterator() bạn
sẽ nhận được
một Iterator.

Như ng vẫn còn hơ n thế nữa!

Java cung cấp cho bạn
rất nhiều lớp "bộ sưu tập"
cho phép bạn lưu trữ và
truy xuất các nhóm đối tượng.
Ví dụ: `Vector` và `LinkedList`.

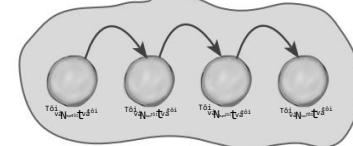
1

Hầu hết đều có giao diện
khác nhau.

Như ng hầu hết
chúng đều hỗ
trợ cách để c
đư ợc Iterator

The diagram shows a vesicle-like structure containing four copies of the T6SS protein complex. Each complex is composed of a central hexagonal head and a long tail ending in a needle. The complexes are labeled 1, 2, 3, and 4.

Danh sách liên kết



và nhiều hơn nữa.

Và nếu họ không hỗ trợ Iterator
thì cũng không sao, vì bây giờ bạn đã
biết cách tự xây dựng Iterator của riêng mình

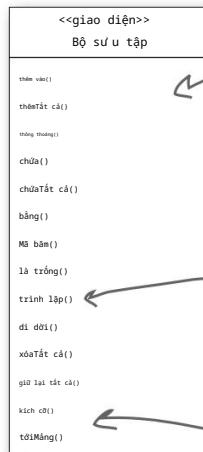
trình lặp và bộ sưu tập

Trình lặp và Bộ sưu tập

Chúng tôi đã sử dụng một số lớp là một phần của Java Collections Framework.

"Khung" này chỉ là một tập hợp các lớp và giao diện, bao gồm ArrayList, mà chúng ta đã sử dụng, và nhiều lớp khác như Vector, LinkedList, Stack và PriorityQueue. Mỗi lớp này triển khai giao diện java.util.Collection, chứa một loạt các phương thức hữu ích để thao tác các nhóm đối tượng.

Chúng ta hãy xem nhanh giao diện:



Như bạn thấy, có đủ mọi thứ tốt ở đây. Bạn có thể thêm và xóa các phần tử khỏi bộ sưu tập của mình mà thậm chí không cần biết cách triển khai.

Đây là người bạn cũ của chúng ta, phuơng thức iterator(). Với phuơng thức này, bạn có thể lấy Iterator cho bất kỳ lớp nào triển khai giao diện Collection.

Các phuơng pháp tiện dụng khác bao gồm size() để lấy số lượng phần tử và toArray() để biến bộ sưu tập của bạn thành một mảng.



Watch it!

Hashtable là một trong số ít lớp giàn tiếp hỗ trợ Iterator. Như bạn đã thấy khi chúng tôi triển khai CafeMenu, bạn có thể lấy Iterator từ nó, nhưng chỉ bằng cách đầu tiên lấy Collection được gọi là values. Nếu bạn nghĩ về nó, điều này có ý nghĩa: Hashtable giữ hai

tập hợp các đối tượng: khóa và giá trị. Nếu chúng ta muốn lặp lại các giá trị của nó, trước tiên chúng ta cần lấy chúng từ Hashtable và sau đó lấy trình lặp.

Điểm tuyệt vời của Collections và Iterator là mỗi đối tượng Collection đều biết cách tạo Iterator riêng của mình.
Gọi iterator() trên ArrayList sẽ trả về một Iterator cụ thể được tạo cho ArrayList, nhưng bạn không bao giờ cần phải xem hoặc lo lắng về lớp cụ thể mà nó sử dụng; bạn chỉ cần sử dụng giao diện Iterator.



Iterator và Collections trong Java 5

Java 5 bao gồm một dạng câu lệnh for mới, được gọi là for-in, cho phép bạn lặp lại một bộ sưu tập hoặc một mảng mà không cần tạo một trình lặp một cách rõ ràng.

Để sử dụng for/in, bạn sử dụng câu lệnh for trông như sau:

Lặp lại qua obj được gán cho phần
từng đối tượng tử tiếp theo trong
trong bộ sưu tập. bộ sưu tập mỗi lần lặp.

```
đối với (Đối tượng obj: bộ sưu tập) {  
    ...  
}
```

Sau đây là cách bạn lặp lại ArrayList bằng cách sử dụng for/in

```
Các mục ArrayList = new ArrayList();
items.add(new MenuItem("Bánh kép", "bánh kép ngon", đúng, 1,59));
items.add(new MenuItem("Bánh qué", "bánh qué ngon", đúng, 1,99));
items.add(new MenuItem("Toast", "perfect toast", true, 0,59));
```

```
cho (Mục MenuItem: mục) {  
    System.out.println("Món ăn sáng: " + mục)  
}
```

Lắp lại danh sách và in mỗi mục.



Tải một
ArrayList
các MenuItem



Watch it!

Bạn cần sử dụng tính năng chung mới
của Java 5 để đảm bảo cho/
trong an toàn kiểu. Hãy đảm bảo bạn
đọc kỹ thông tin chi tiết trước khi
sử dụng generic và for/in.

nam châm mă

Mă Nam Châm



Các đầu bếp đã quyết định rằng họ muốn có thể thay đổi các mục trong thực đơn bữa trưa của mình; nói cách khác, họ sẽ cung cấp một số mục vào Thứ Hai, Thứ Tư, Thứ Sáu và Chủ Nhật, và các mục khác vào Thứ Ba, Thứ Năm và Thứ Bảy. Ai đó đã viết mã cho một DinerMenu Iterator "Alternating" mới để nó thay đổi các mục trong thực đơn, nhưng họ đã xáo trộn nó và đặt nó trên tủ lạnh trong Diner như một trò đùa. Bạn có thể lắp lại nó không? Một số đầu ngoặc nhọn rơi xuống sàn và chúng quá nhỏ để nhặt, vì vậy hãy thoải mái thêm nhiều đầu ngoặc nhọn như bạn cần.

```
MenuItem menuItem = items[vị trí]; vị trí = vị trí + 2; trả về menuItem;
```

```
nhập java.util.Iterator;
nhập java.util.Calendar;
```

Đối tượng công khai tiếp theo() {

```
public AlternatingDinerMenuIterator(MenuItem[] items)
```

```
this.items = items;
Lịch rightNow = Calendar.getInstance(); vị
trí = rightNow.get(Calendar.DAY_OF_WEEK) % 2;
```

thực hiện Iterator

công khai void remove() {

```
MenuItem[] mục; int
vị trí;
```

}

lớp công khai AlternatingDinerMenuIterator

```
boolean công khai hasNext() {
```

```
ném UnsupportedOperationException mới(
    "Trình lắp menu diner thay thế không hỗ trợ remove()");
```

```
nếu (vị trí >= items.length || items[vị trí] == null) {
    trả về false; } else
{ trả về true;
}
```

}



Cô hầu bàn đã sẵn sàng cho giờ vàng chư a?

Waitress đã có nhiều cải tiến, nhưng bạn phải thừa nhận rằng ba lệnh gọi `printMenu()` trông có vẻ hơi i xấu.

Thực tế mà nói, mỗi lần chúng ta thêm một menu mới, chúng ta sẽ phải mở triển khai Waitress và thêm nhiều mã hơn. Bạn có thể nói "vi phạm Nguyên tắc Mở Đóng" không?

Ba lệnh gọi `createIterator()`.

```
công khai void printMenu() {
    Trinh lặp pancakeIterator = pancakeHouseMenu.createIterator();
    Trinh lặp dinerIterator = dinerMenu.createIterator();
    Trinh lặp cafeIterator = cafeMenu.createIterator();

    System.out.println("MENU\n---\nBỮA SÁNG");
    printMenu(pancakeIterator);

    System.out.println("\nBỮA TRƯA");
    printMenu(dinerIterator);

    System.out.println("\nBỮA TỐI");
    printMenu(cafeIterator);
}
```

Ba lần gọi tới `printMenu()`.

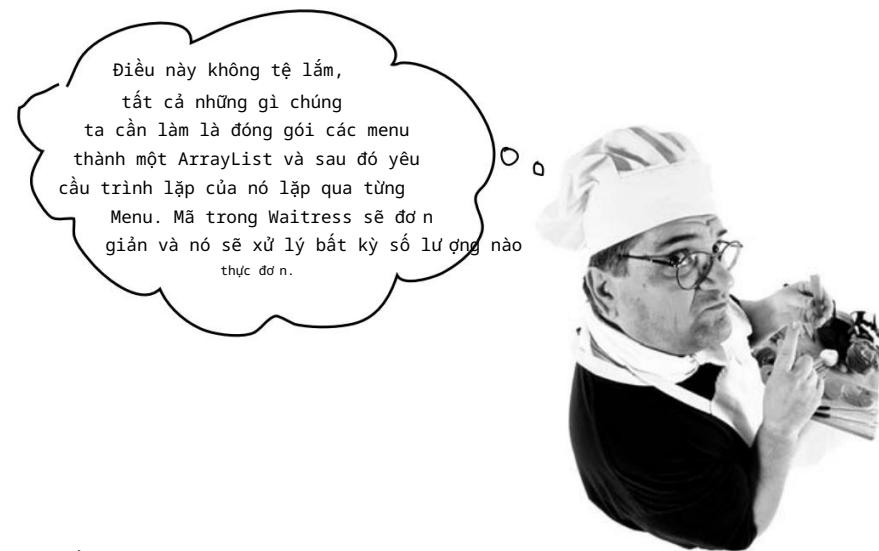
Mỗi khi thêm hoặc xóa menu, chúng ta sẽ phải mở mã này để thay đổi.

Không phải lỗi của Waitress. Chúng tôi đã làm rất tốt việc tách rời việc triển khai menu và trích xuất lặp lại thành một trình lặp. Nhưng chúng tôi vẫn đang xử lý các menu bằng các đối tượng riêng biệt, độc lập - chúng tôi cần một cách để quản lý chúng cùng nhau.

não Apower

Waitress vẫn cần thực hiện ba lệnh gọi `printMenu()`, một lệnh cho mỗi menu. Bạn có thể nghĩ ra cách kết hợp các menu sao cho chỉ cần thực hiện một lệnh gọi không? Hoặc có thể là một Iterator được chuyển cho Waitress để lặp lại tất cả các menu?

một thiết kế mới?



Nghe có vẻ như đầu bếp đã tìm ra cách. Hãy thử xem:

```

lớp công khai Waitress {
    Menu ArrayList;
}

công khai Waitress(ArrayList menus) {
    this.menus = menu;
}

công khai void printMenu() {
    Menu lặpIterator = menus.iterator();
    trong khi(menuIterator.hasNext()) {
        Thực đơn menu = (Menu)menuIterator.next();
        printMenu(menu.createIterator());
    }
}

void printMenu(Trình lặp iterator) {
    trong khi (iterator.hasNext()) {
        MenuItem menuItem = (MenuItem)iterator.next();
        System.out.print(menuItem.getName() + ", ");
        System.out.print(menuItem.getPrice() + -- " ");
        System.out.println(menuItem.getDescription());
    }
}

```

Bây giờ chúng ta chỉ cần lấy một
ArrayList của thực đơn.

Và chúng ta lặp
qua các
menu, truyền trình
lặp của từng
menu cho phuơng
thức printMenu() quá tải.

Không có
thay đổi mã nào ở đây.

Trong có vẻ khá ổn, mặc dù chúng ta đã mất tên của các menu, như ng chúng ta có thể
thêm tên vào từng menu.

trình lặp và các mảng tổng hợp

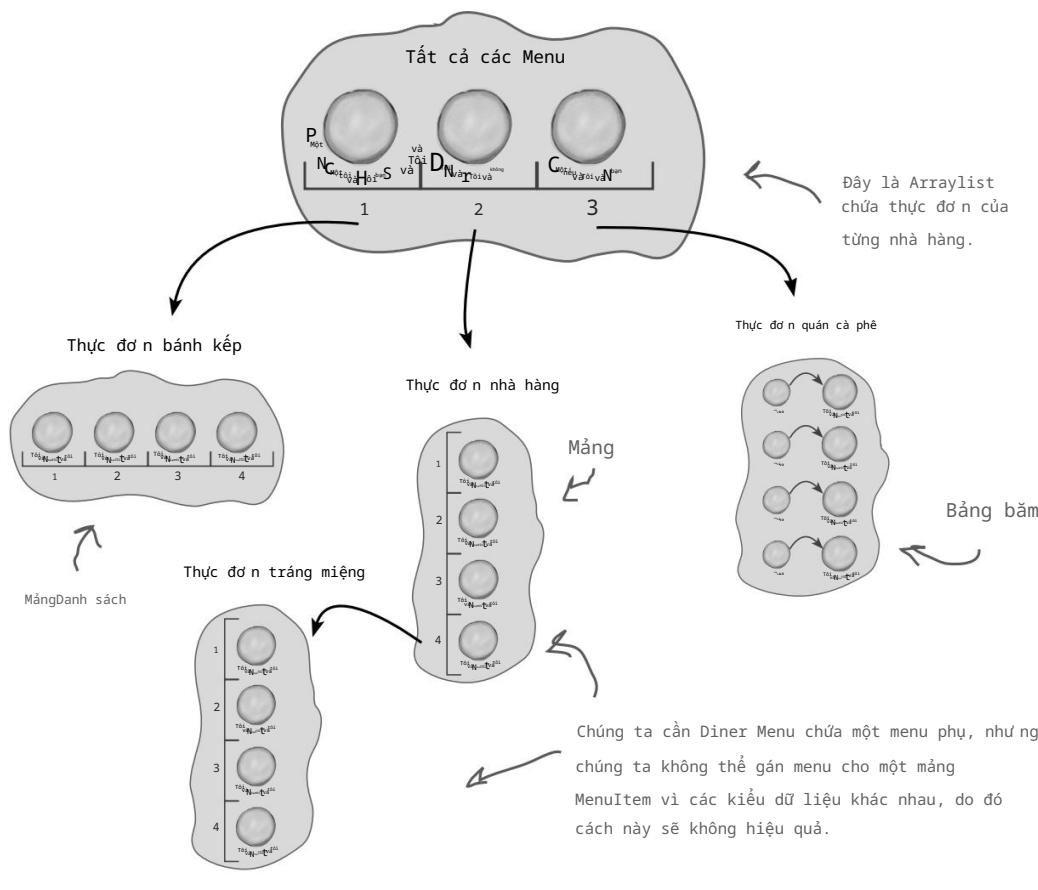
Đúng lúc chúng ta nghĩ rằng đã an toàn...

Bây giờ họ muốn thêm một thực đơn tráng miệng.

Được rồi, bây giờ thì sao? Bây giờ chúng ta phải hỗ trợ không chỉ nhiều menu mà còn cả menu trong menu.

Sẽ thật tuyệt nếu chúng ta có thể biến thực đơn tráng miệng thành một phần tử của bộ sưu tập DinerMenu, như ng điều đó sẽ không khả thi vì hiện tại nó đã được triển khai.

Những gì chúng tôi muốn (tự述 ng tự như thế này):



Như ng điều
này không hiệu quả!

Chúng ta không thể gán thực đơn tráng miệng
vào mảng MenuItem.

Đã đến lúc thay đổi!

thời gian để tái cấu trúc

Chúng ta cần gì?

Đã đến lúc đưa ra quyết định điều hành để làm lại việc thực hiện của đầu bếp thành một cái gì đó đủ chung chung để làm việc trên tất cả các menu (và bây giờ là các menu phụ). Đúng vậy, chúng tôi sẽ nói với các đầu bếp rằng đã đến lúc chúng tôi phải thực hiện lại thực đơn.

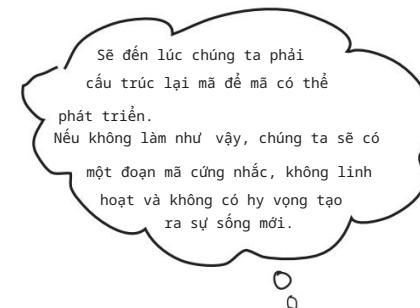
Thực tế là chúng ta đã đạt đến mức độ phức tạp đến mức nếu chúng ta không thiết kế lại ngay bây giờ, chúng ta sẽ không bao giờ có được một thiết kế có thể chứa được các chức năng bổ sung hoặc menu phụ sau này.

Vậy, chúng ta thực sự cần gì ở thiết kế mới?

Chúng ta cần một loại cấu trúc hình cây nào đó có thể chứa các menu, menu phụ và mục menu.

β Chúng ta cần đảm bảo rằng chúng ta duy trì một cách để duyệt qua các mục trong mỗi menu sao cho tiện lợi nhất có thể như những gì chúng ta đang làm với trình lặp.

β Chúng ta có thể cần có khả năng duyệt qua các mục theo cách linh hoạt hơn. Ví dụ, chúng ta có thể cần lặp lại chỉ qua thực đơn tráng miệng của Diner hoặc chúng ta có thể cần lặp lại toàn bộ thực đơn của Diner, bao gồm cả thực đơn phụ tráng miệng.

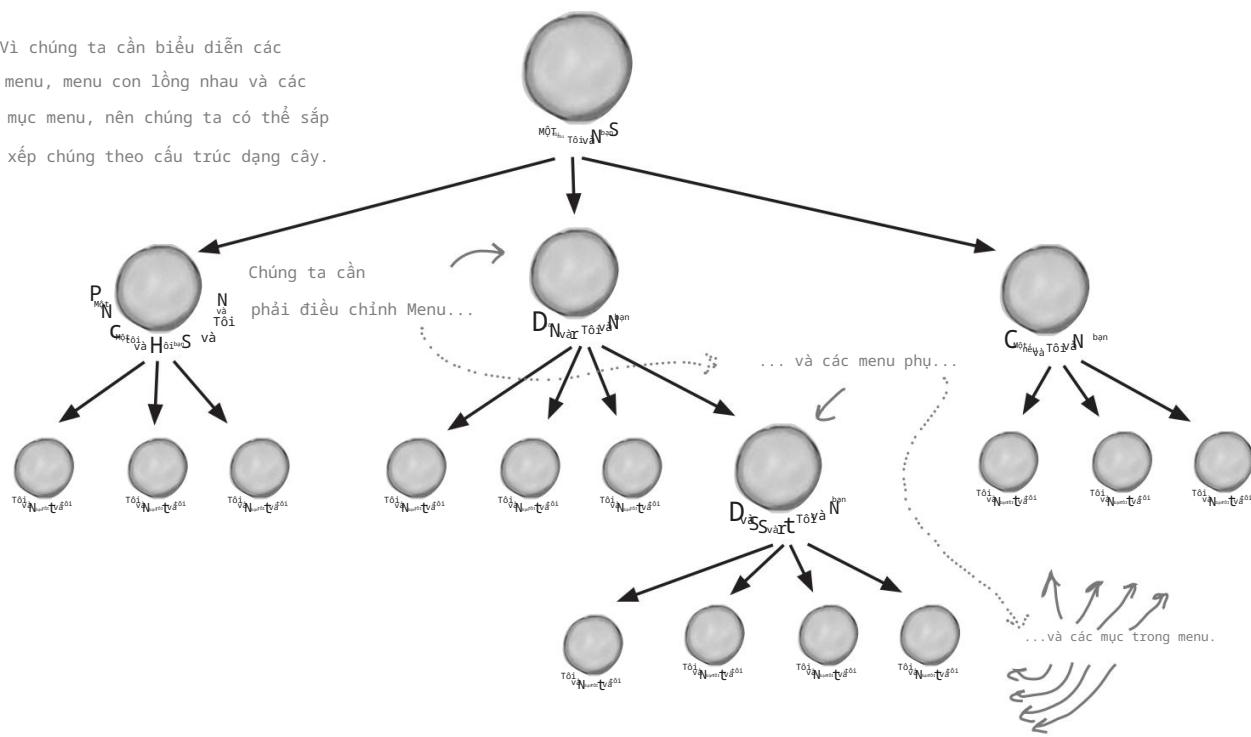


Sẽ đến lúc chúng ta phải cấu trúc lại mã để mã có thể phát triển.
Nếu không làm như vậy, chúng ta sẽ có một đoạn mã cứng nhắc, không linh hoạt và không có hy vọng tạo ra sự sống mới.



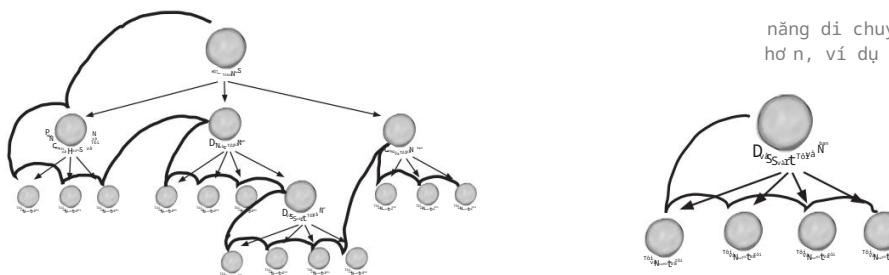
trình lắp và các mău tống hợp

Vì chúng ta cần biểu diễn các menu, menu con lồng nhau và các mục menu, nên chúng ta có thể sắp xếp chúng theo cấu trúc dạng cây.



Chúng ta vẫn cần có thể duyệt qua tất cả các mục trong cây.

Chúng ta cũng cần có khả năng di chuyển linh hoạt hơn, ví dụ như qua một menu.



não Apower

Bạn sẽ xử lý nếp nhăn mới này đối với yêu cầu thiết kế của chúng tôi như thế nào? Hãy suy nghĩ trước khi lật trang.

mẫu tổng hợp đư ợc xác định

Mẫu tổng hợp đư ợc định nghĩa

Đúng vậy, chúng ta sẽ giới thiệu một mẫu khác để giải quyết vấn đề này. Chúng tôi không từ bỏ Iterator - nó vẫn sẽ là một phần trong giải pháp của chúng tôi - tuy nhiên, vấn đề quản lý menu đã có một chiều hướng mới mà Iterator không giải quyết đư ợc. Vì vậy, chúng ta sẽ lùi lại và giải quyết nó bằng Composite Pattern.

Chúng tôi sẽ không vòng vo về mô hình này, chúng tôi sẽ tiếp tục và đưa ra định nghĩa chính thức

Hiện nay:

Mẫu Composite cho phép bạn kết hợp các đối tượng thành các cấu trúc cây để biểu diễn các hệ thống phân cấp một phần-toàn thể. Composite cho phép khách hàng xử lý các đối tượng riêng lẻ và các thành phần của đối tượng một cách thống nhất.

Hãy nghĩ về điều này theo khía cạnh menu của chúng ta: mẫu này cung cấp cho chúng ta một cách để tạo cấu trúc cây có thể xử lý một nhóm menu và mục menu lồng nhau trong cùng một cấu trúc. Bằng cách đặt menu và mục trong cùng một cấu trúc, chúng ta tạo ra một hệ thống phân cấp bộ phận-toàn bộ; nghĩa là, một cây đối tượng đư ợc tạo thành từ các bộ phận (menu và mục menu) nhưng có thể đư ợc xử lý như một tổng thể, giống như một menu über lớn.

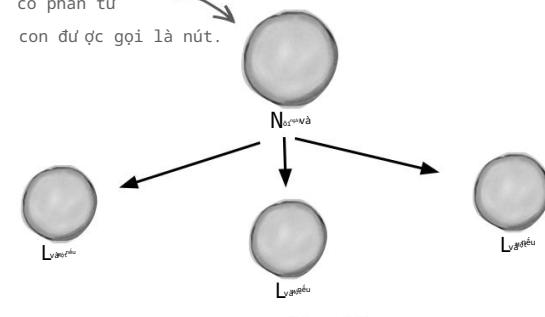
Khi đã có über menu, chúng ta có thể sử dụng mẫu này để xử lý "các đối tượng và thành phần riêng lẻ một cách thống nhất". Điều đó có nghĩa là gì? Nghĩa là nếu chúng ta có cấu trúc cây gồm các menu, menu con và lẻ là các menu con cùng với các mục menu, thì bất kỳ menu nào cũng là một "composition" vì nó có thể chứa cả các menu và mục menu khác. Các đối tượng riêng lẻ chỉ là các mục menu - chúng không chứa các đối tượng khác. Như bạn sẽ thấy, sử dụng thiết kế theo Mẫu tổng hợp sẽ cho phép chúng ta viết một số mã đơn giản có thể áp dụng cùng một thao tác (như in!) trên toàn bộ cấu trúc menu.

Đây là cấu trúc cây

Các phần tử

có phần tử

con đư ợc gọi là nút.

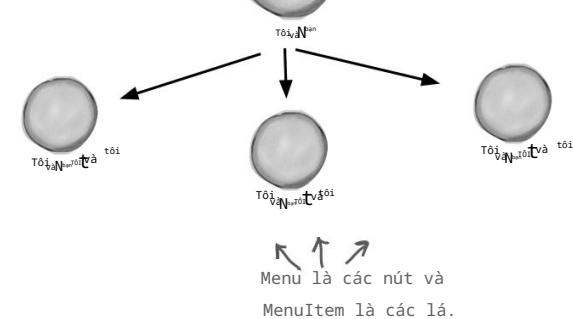


Các phần tử không có phần tử
con đư ợc gọi là lá.

Chúng ta có thể
biểu diễn

Menu và

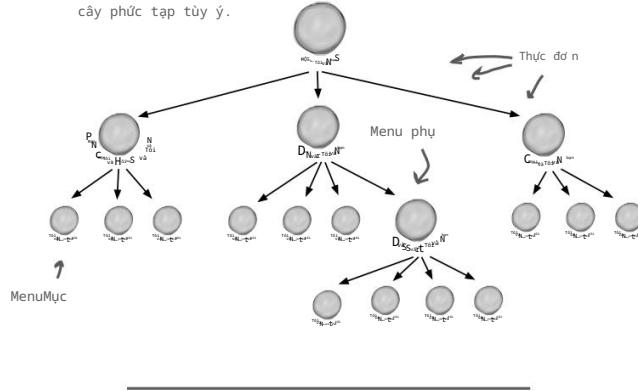
MenuItem theo cấu trúc cây.



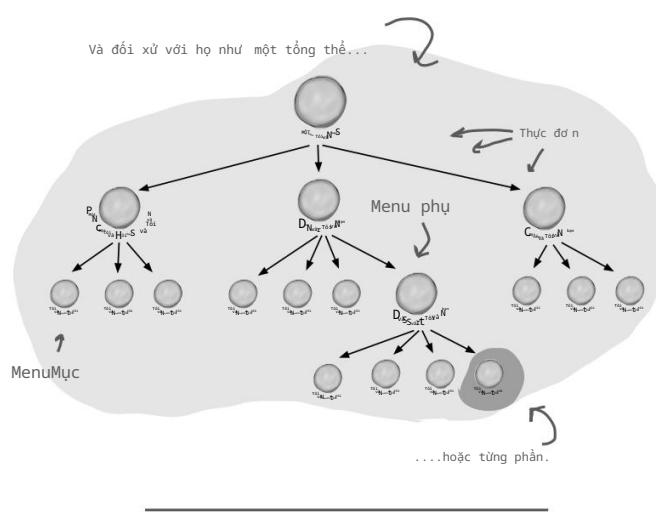
Menu là các nút và
MenuItem là các lá.

trình lặp và các mẫu tổng hợp

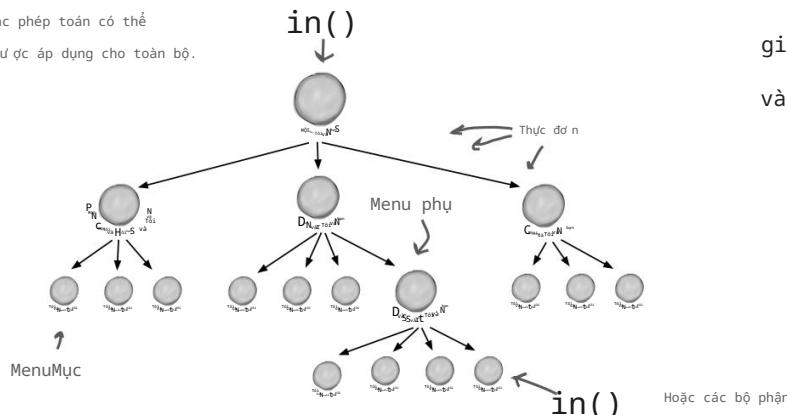
Chúng ta có thể tạo ra những cây phức tạp tùy ý.



Và đổi xử với họ như một tổng thể...



Các phép toán có thể
được áp dụng cho toàn bộ.



Mẫu tổng hợp cho phép chúng ta xây dựng các cấu trúc đối tư ợng

dư ới dạng cây chứa cả các thành phần của đối tư ợng và

các đối tư ợng riêng lẻ dư ới dạng các nút.

Sử dụng cấu trúc tổng hợp, chúng ta có thể áp dụng các thao tác giống nhau trên cả các đối tư ợng riêng lẻ. Nói cách khác, trong hầu hết các trường hợp, chúng ta có thể bỏ qua sự khác biệt

giữa các thành phần của đối tư ợng và các đối tư ợng riêng lẻ.

sơ đồ lớp mẫu tổng hợp

Máy khách sử dụng giao diện Thành phần để thao tác các đối tượng trong thành phần.

Thành phần này định nghĩa một giao diện cho tất cả các đối tượng trong thành phần: cả nút hợp thành và nút lá.

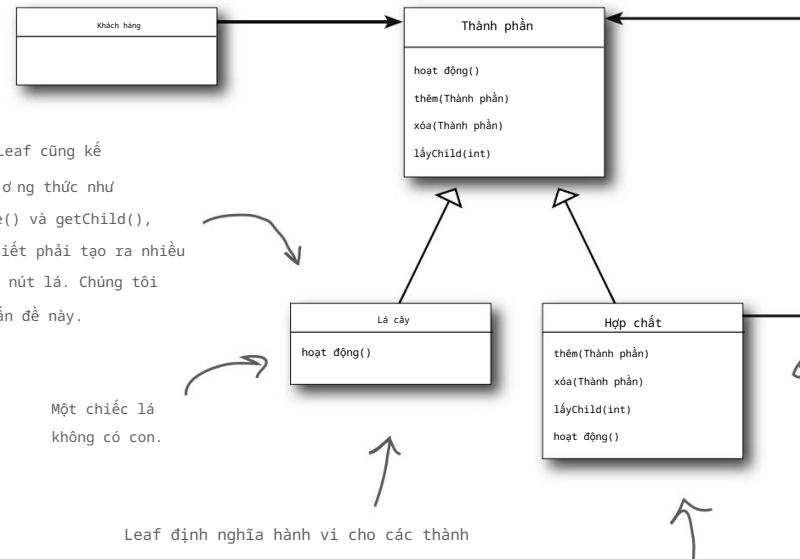
Thành phần có thể triển khai hành vi mặc định cho add(), remove(), getChild() và các hoạt động của nó.

Lưu ý rằng Leaf cũng kế thừa các phương thức như add(), remove() và getChild(), không nhất thiết phải tạo ra nhiều giác quan cho một nút lá. Chúng tôi sẽ quay lại vấn đề này.

Một chiếc lá không có con.

Leaf định nghĩa hành vi cho các thành phần trong thành phần. Nó thực hiện điều này bằng cách triển khai các hoạt động mà Composite hỗ trợ.

Composite cũng triển khai các hoạt động liên quan đến Leaf. Lưu ý rằng một số trong số này có thể không có ý nghĩa trong Composite, do đó trong trường hợp đó, có thể tạo ra ngoại lệ.



không có Những câu hỏi ngớ ngẩn

H: Thành phần, Hợp chất, Cây?
Tôi bối rối.

A: Một hợp chất bao gồm các thành phần. Các thành phần có hai loại: thành phần tổng hợp và thành phần lá. Nghe có vẻ đẽ quy? Đúng vậy. Thành phần tổng hợp chứa một tập hợp các thành phần con, các thành phần con đó có thể là các thành phần tổng hợp hoặc thành phần lá khác.

Khi bạn sắp xếp dữ liệu theo cách này, bạn sẽ có một cấu trúc cây (thực chất là một cấu trúc cây ngược) với một hợp chất ở gốc và các nhánh hợp chất phát triển lên đến các nút lá.

H: Điều này liên quan thế nào đến trình lặp?

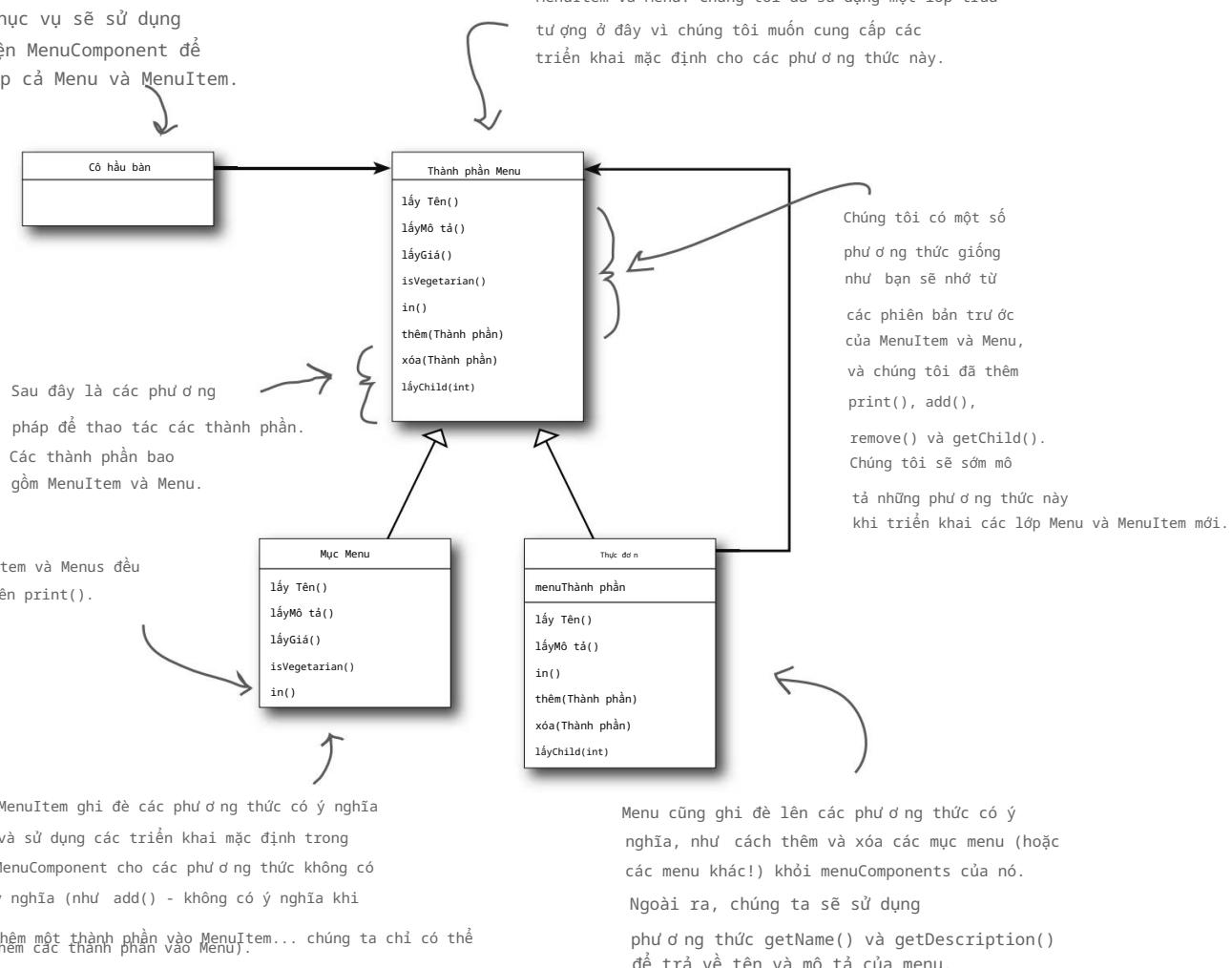
A: Hãy nhớ rằng chúng ta đang thực hiện một điều mới cách tiếp cận. Chúng tôi sẽ triển khai lại các menu bằng một giải pháp mới: Mẫu tổng hợp. Vì vậy, đừng tìm kiếm một số chuyển đổi kỳ diệu từ một trình lặp sang một composite. Nói như vậy, cả hai hoạt động rất tốt với nhau. Bạn sẽ sớm thấy rằng chúng ta có thể sử dụng trình lặp theo một số cách trong triển khai composite.

Thiết kế Menu với Composite

Vậy, làm thế nào để áp dụng Composite Pattern vào menu của chúng ta? Để bắt đầu, chúng ta cần tạo một giao diện thành phần; giao diện này hoạt động như giao diện chung cho cả menu và mục menu và cho phép chúng ta xử lý chúng một cách thống nhất. Nói cách khác, chúng ta có thể gọi cùng một phương thức trên menu hoặc mục menu.

Bây giờ, có thể không hợp lý khi gọi một số phương thức trên một mục menu hoặc một menu, nhưng chúng ta có thể xử lý vấn đề đó, và chúng ta sẽ làm ngay sau đây. Như vậy, hãy cùng xem qua bản phác thảo về cách các menu sẽ phù hợp với cấu trúc Mẫu tổng hợp:

Người phục vụ sẽ sử dụng giao diện MenuComponent để truy cập cả Menu và MenuItem.



thực hiện menu tổng hợp

Triển khai thành phần Menu

Để rõ ràng, chúng ta sẽ bắt đầu với lớp trừu tượng MenuComponent; hãy nhớ rằng, vai trò của thành phần menu là cung cấp giao diện cho các nút lá và các nút hợp thành. Vậy giờ bạn có thể hỏi, "MenuComponent không phải đang đóng hai vai trò sao?" Có thể là như vậy và chúng ta sẽ quay lại vấn đề đó.

Tuy nhiên, hiện tại chúng ta sẽ cung cấp một triển khai mặc định cho các phương thức để nếu MenuItem (lá) hoặc Menu (hợp thành) không muốn triển khai một số phương thức (như getChild() cho một nút lá), chúng có thể quay lại một số hành vi cơ bản:

MenuComponent cung cấp các triển khai mặc định cho mọi phương thức.



```

lớp trừu tượng công khai MenuComponent {

    công khai void add(MenuComponent menuComponent) {
        ném UnsupportedOperationException mới();
    }
    công khai void remove(MenuComponent menuComponent) {
        ném UnsupportedOperationException mới();
    }
    công khai MenuComponent getChild(int i) {
        ném UnsupportedOperationException mới();
    }

    công khai String getName() {
        ném UnsupportedOperationException mới();
    }
    công khai String getDescription() {
        ném UnsupportedOperationException mới();
    }
    công khai double getPrice() {
        ném UnsupportedOperationException mới();
    }
    công khai boolean isVegetarian() {
        ném UnsupportedOperationException mới();
    }

    công khai void print() {
        ném UnsupportedOperationException mới();
    }
}

```

Tất cả các thành phần phải triển khai giao diện MenuComponent; tuy nhiên, vì các nút và lá có vai trò khác nhau nên chúng ta không thể luôn xác định được một triển khai mặc định có ý nghĩa cho từng phương thức.

Đôi khi điều tốt nhất bạn có thể làm là đưa ra một ngoại lệ thời gian chạy.

Vì một số phương thức này chỉ có ý nghĩa đối với MenuItems, và một số chỉ có ý nghĩa đối với Menus, nên triển khai mặc định là UnsupportedOperationException. Theo cách đó, nếu MenuItem hoặc Menu không hỗ trợ một hoạt động, chúng không phải làm gì cả, chúng chỉ có thể kết thúc triển khai mặc định.



Chúng tôi đã nhóm các phương thức "tổng hợp" lại với nhau - tức là các phương thức để thêm, xóa và lấy MenuComponents.



Sau đây là các phương thức "hoạt động"; chúng được sử dụng bởi các MenuItem. Thực ra chúng ta cũng có thể sử dụng một vài trong số chúng trong Menu, như bạn sẽ thấy trong một vài trang khi chúng tôi hiển thị mã Menu.

print() là phương thức "hoạt động" mà cả Menu và MenuItem của chúng ta sẽ triển khai, như chúng tôi cung cấp một hoạt động mặc định tại đây.

trình lắp và các mẫu tổng hợp

Triển khai mục Menu

Đư ợc rồi, hãy thử lớp MenuItem. Hãy nhớ rằng, đây là lớp lá trong sơ đồ Composite và nó triển khai hành vi của các phần tử trong composite.

```

lớp công khai MenuItem mở rộng MenuComponent {
    Tên chuỗi;
    Mô tả chuỗi;
    ngữ điệu ăn chay theo kiểu Boolean;
    giá gấp đôi;

    public MenuItem(String tên,
                    Mô tả chuỗi, boolean
                    vegetarian, giá gấp đôi)

    {
        this.name = tên;
        this.description = mô tả;
        this.vegetarian = ăn chay;
        this.price = giá;
    }

    công khai String getName() {
        trả về tên;
    }

    công khai String getDescription() {
        mô tả trả về;
    }

    công khai double getPrice() {
        giá trả lại;
    }

    công khai boolean isVegetarian() {
        trả lại chế độ ăn chay;
    }

    công khai void print() {
        System.out.print(" " nếu      + lấy Tên());
        (isVegetarian()) {
            Hệ thống.out.print("(v)");
        }
        Hệ thống.out.println(" " Hệ      + lấy giá());
        thống.out.println("          --      + lấy Mô tả());
    }
}

```

Tôi rất vui vì chúng ta
đang đi theo hướng này, tôi nghĩ điều
này sẽ mang lại cho tôi sự linh hoạt cần
thiết để thực hiện thực đơn bánh crepe
mà tôi luôn mong muốn.



Đầu tiên chúng ta cần
mở rộng giao diện
MenuComponent.

Hàm tạo chỉ lấy tên, mô tả,
v.v. và giữ tham chiếu đến tất cả
chúng.
Điều này khá giống với cách triển
khai mục menu cũ của chúng tôi.

Đây là phương thức lấy dữ liệu của chúng
ta - giống như cách triển khai trước đó.

Điều này khác với cách thực hiện trước đây.
Ở đây chúng ta ghi đè phương thức print()
trong lớp MenuComponent. Đối với MenuItem,
phương thức này in ra mục menu đầy đủ: tên, mô
tả, giá và có phải là món chay hay không.

cấu trúc tổng hợp

Triển khai Menu tổng hợp

Bây giờ chúng ta đã có MenuItem, chúng ta chỉ cần lớp tổng hợp, mà chúng ta gọi là Menu. Hãy nhớ rằng, lớp tổng hợp có thể chứa MenuItem hoặc các Menu khác. Có một số phương thức từ MenuComponent mà lớp này không triển khai: getPrice() và isVegetarian(), vì chúng không có nhiều ý nghĩa đối với Menu.

```
Menu cũng là một MenuComponent,
giống như MenuItem.

lớp công khai Menu mở rộng MenuComponent {
    MenuComponents của ArrayList = new ArrayList();
    Tên chuỗi;
    Mô tả chuỗi;

    public Menu(Tên chuỗi, Mô tả chuỗi) {
        this.name = tên;
        this.description = mô tả;
    }

    công khai void add(MenuComponent menuComponent) {
        menuComponents.add(menuComponent);
    }

    công khai void remove(MenuComponent menuComponent) {
        menuComponents.remove(menuComponent);
    }

    công khai MenuComponent getChild(int i) {
        trả về (MenuComponent)menuComponents.get(i);
    }

    công khai String getName() {
        trả về tên;
    }

    công khai String getDescription() {
        mô tả trả về;
    }

    công khai void print() {
        System.out.print("\n" + getName());
        System.out.println(", " + getDescription());
        System.out.println("-----");
    }
}
```

Menu có thể có bất kỳ số lượng phần tử con nào thuộc kiểu MenuComponent, chúng ta sẽ sử dụng ArrayList nội bộ để lưu trữ những phần tử này.

Điều này khác với cách triển khai cũ của chúng tôi: chúng tôi sẽ đặt tên và mô tả cho từng Menu. Trước đây, chúng tôi chỉ dựa vào việc có các lớp khác nhau cho mỗi menu.

Sau đây là cách bạn thêm MenuItems hoặc Menus khác vào Menu. Vì cả MenuItems và Menus đều là MenuComponents, chúng ta chỉ cần một phương pháp để thực hiện cả hai.

Bạn cũng có thể xóa MenuComponent hoặc lấy MenuComponent.

Sau đây là các phương thức lấy tên và mô tả.

Lưu ý, chúng ta không ghi đè getPrice() hoặc isVegetarian() vì những phương thức đó không có ý nghĩa (mặc dù bạn có thể lập luận rằng isVegetarian() có thể có ý nghĩa). Nếu ai đó cố gắng gọi những phương thức đó trên Menu, họ sẽ nhận được UnsupportedOperationException.

Để in Menu, chúng ta in tên Menu và mô tả.



Bất tuyệt vời. Vì menu là một hợp chất và chứa cả Menu Items và các Menu khác, phương thức print() của nó sẽ in mọi thứ mà nó chứa. Nếu không, chúng ta sẽ phải lắp lại toàn bộ hợp chất và tự in từng mục. Điều đó làm mất đi mục đích của việc có một hợp chất kết cấu.

Như bạn sẽ thấy, việc triển khai print() một cách chính xác rất dễ dàng vì chúng ta có thể dựa vào từng thành phần để có thể tự in ra. Tất cả đều đẽ quy và tuyệt vời. Hãy xem thử:

Sửa phương thức print()

```

lớp công khai Menu mở rộng MenuComponent {
    MenuComponents của ArrayList = new ArrayList();
    Tên chuỗi;
    Mô tả chuỗi;

    // mã xây dựng ở đây

    // các phương pháp khác ở đây

    công khai void print() {
        System.out.print("\n" + getName());
        System.out.println(", " + getDescription());
        System.out.println("-----");
    }
}

```

```

Trình lắp iterator = menuComponents.iterator();
trong khi (iterator.hasNext()) {
    MenuComponent menuComponent =
        (MenuComponent) iterator.next();
    menuComponent. print();
}
}

```

LƯU Ý: Nếu trong quá trình lắp lại này, chúng ta gặp một đối tượng Menu khác, phương thức print() của đối tượng đó sẽ bắt đầu một lần lắp lại khác, v.v.

Tất cả những gì chúng ta cần làm là thay đổi phương thức print() để in không chỉ thông tin về Menu này mà còn tất cả các thành phần của Menu này: các Menu và MenuItem khác.

Nhìn này! Chúng ta có thể sử dụng Iterator. Chúng ta sử dụng nó để lắp qua tất cả các thành phần của Menu... chúng có thể là các Menu khác hoặc chúng có thể là MenuItem. Vì cả Menu và MenuItem đều triển khai print(), chúng ta chỉ cần gọi print() và phần còn lại tùy thuộc vào chúng.

thử nghiệm lái xe menu tổng hợp

Đang chuẩn bị cho chuyến lái thử...

Đã đến lúc chúng ta thử nghiệm đoạn mã này, nhưng chúng ta cần cập nhật mã Waitress trước khi thực hiện - dù sao thì cô ấy cũng là máy khách chính của đoạn mã này:

```
lớp công khai Waitress {
    MenuComponent tắt cả Menu;

    cô hầu bàn công cộng(MenuComponent allMenus) {
        this.allMenus = allMenus;
    }

    công khai void printMenu() {
        allMenus.print();
    }
}
```

Đúng vậy! Mã Waitress thực sự đơn giản như thế này.

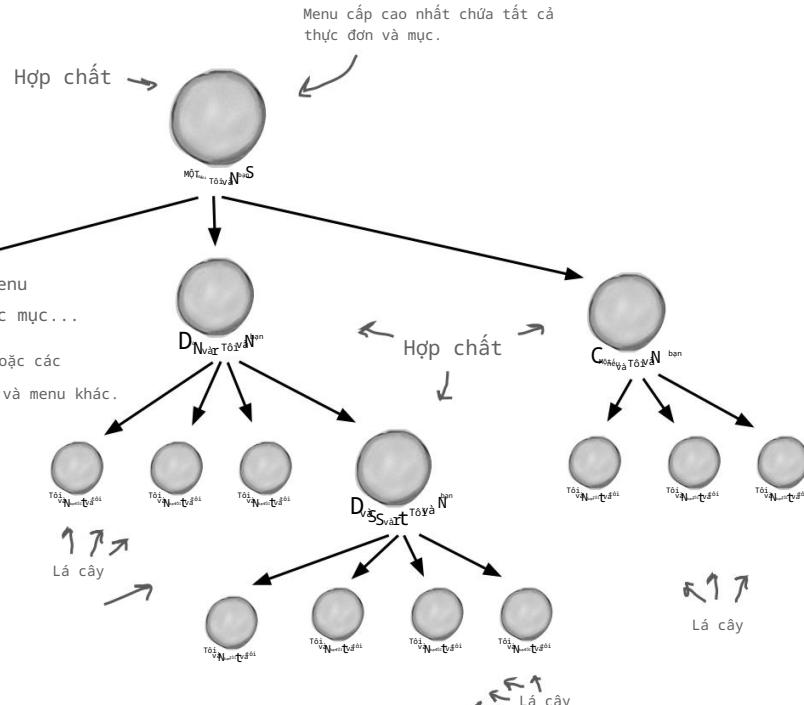
Bây giờ chúng ta chỉ đưa cho cô ấy thành phần menu cấp cao nhất, thành phần chứa tất cả các menu khác. Chúng tôi gọi đó là allMenus.

Tất cả những gì cô ấy phải làm để in toàn bộ hệ thống menu - tắt cả các menu và tất cả các mục menu - là gọi print() trên menu cấp cao nhất.

Chúng ta sẽ có một cô hầu bàn hạnh phúc.

Được rồi, một điều cuối cùng trước khi chúng ta viết bản thử nghiệm. Chúng ta hãy cùng xem menu composite sẽ trông như thế nào khi chạy:

Mỗi Menu và
MenuItem đều triển khai
giao diện MenuComponent.



trình lặp và các mẫu tổng hợp

Bây giờ là lúc lái thử...

Được rồi, bây giờ chúng ta chỉ cần một lần lái thử. Không giống như phiên bản trước, chúng ta sẽ xử lý tất cả việc tạo menu trong lần lái thử. Chúng ta có thể yêu cầu mỗi đầu bếp đưa cho chúng ta thực đơn mới của họ, nhưng trước tiên hãy thử nghiệm tất cả. Đây là mã:

```

lớp công khai MenuTestDrive {
    public static void main(String args[]) {
        MenuComponent pancakeHouseMenu =
            Menu mới ("MENU CỦA NHÀ HÀNG BÁNH KẸO", "Bữa sáng");
        MenuComponent dinerMenu =
            Menu mới ("MENU BỮA TỐI", "Bữa tối");
        MenuComponent cafeMenu = new Menu("THỰC
            ĐƠN CAFE", "Bữa tối");
        MenuComponent dessertMenu = new Menu("THỰC
            ĐƠN TRÁNG MIỆNG", "Tất nhiên là món tráng miệng rồi!");

        MenuComponent allMenus = new Menu("TẤT CẢ CÁC MENU", "Tất cả các menu được kết hợp");

        allMenus.add(pancakeHouseMenu);
        allMenus.add(dinerMenu);
        allMenus.add(cafeMenu);

        // thêm các mục menu ở đây

        dinerMenu.add(mới MenuItem(
            "Mì ống",
            "Mì Ý với sốt Marinara và một lát bánh mì chua",
            đúng,
            3,89));
        dinerMenu.add(dessertMenu);

        dessertMenu.add(mớiMenuItem(
            "Bánh táo",
            "Bánh táo với lớp vỏ mỏng, phủ kem vani",
            ĐÚNG VẬY,
            1,59));

        // thêm nhiều mục menu ở đây

        Nhân viên phục vụ bàn = Nhân viên phục vụ mới (allMenus);
        phục vụ bàn.printMenu();
    }
}

```

bạn đang ở đây 4 365

Tải xuống tại WoweBook.Com

trách nhiệm tổng hợp

Đang chuẩn bị cho chuyến lái thử...

LƯU Ý: nội dung này dựa trên nguồn đầy đủ.

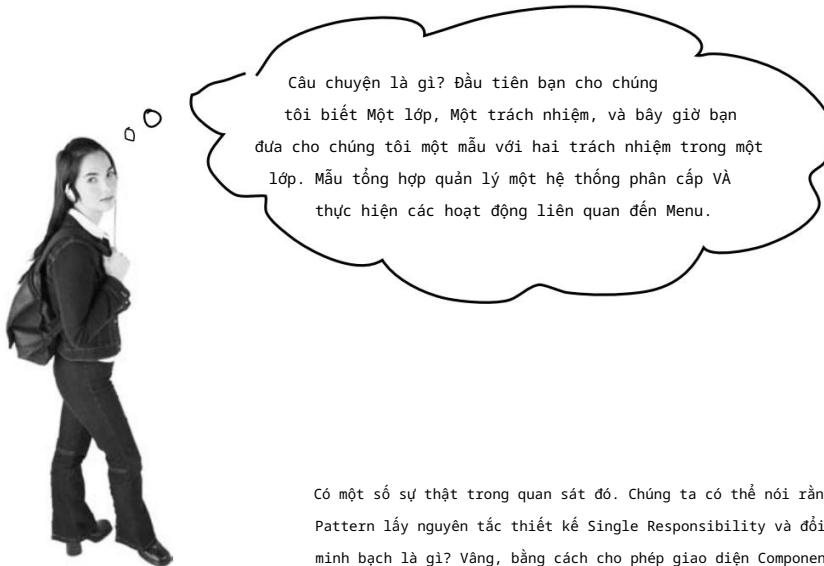
```

Cửa sổ chính sửa tệp Trợ giúp GreenEggs&Spam
% java MenuTestDrive

TẤT CẢ CÁC MENU, Tất cả các menu được kết hợp
-----
THỰC ĐƠN BÁNH XÈO, Bữa sáng
Bữa sáng bánh kép của K&B(v), 2,99 -- Bánh
    kép với trứng rán và bánh mì nướng Bữa sáng bánh kép thông
thường, 2,99 -- Bánh kép với trứng rán, xúc
    xích Bánh kép việt quất(v), 3,49
-- Bánh kép làm từ quả việt quất tươi và xi-rô việt quất Bánh quế(v), 3.59 -- Bánh
quế, với quả việt
    quất hoặc dâu tây tùy chọn
-----
THỰC ĐƠN ĂN TỐI, Bữa trưa
BLT chay (v), 2,99 -- (Giả) Thịt
    xông khói với xà lách và cà chua trên bánh mì nguyên cám BLT, 2,99
-- Thịt xông khói với xà lách và cà chua trên bánh mì nguyên cám
Súp trong ngày, 3.29
-- Một bát súp trong ngày, kèm theo một đĩa salad khoai tây Hotdog, 3.05 -- Một chiếc
hot dog, với dưa
    cải bắp, tương ớt, hành tây, phủ phô mai Rau hấp và gạo lứt(v), 3.99 -- Rau hấp với gạo
lứt Mì ống(v), 3.89 -- Mì Ý sốt Marinara và một lát
    bánh mì chua
-----
THỰC ĐƠN TRÁNG MIỆNG, tất nhiên là tráng miệng rồi!
Bánh táo (v), 1,59 --
    Bánh táo với lớp vỏ mỏng, phủ kem vani Bánh phô mai (v), 1,99 -- Bánh phô mai New
York kem, với lớp vỏ sô cô
    la graham Sorbet (v), 1,89 -- Một muỗng mâm xôi và một muỗng chanh
-----
THỰC ĐƠN CAFE, Bữa tối
Veggie Burger và Air Fries(v), 3,99 -- Veggie
    burger trên bánh mì nguyên cám, rau diếp, cà chua và khoai tây chiên
Súp trong ngày, 3,69 -- Một
    cốc súp trong ngày, kèm theo một đĩa salad Burrito(v), 4,29 -- Một
chiếc burrito lớn, kèm
    theo đậu pinto nguyên hạt, sốt salsa, guacamole
%
```

← Đây là tất cả các menu của chúng tôi... chúng tôi đã in tất
cả những thứ này chỉ bằng cách gọi print() trên menu cấp cao nhất

← Thực đơn tráng miệng mới
đã được in
Khi chúng ta là
in tất cả các
Thành thực đơn
phản của quán ăn



Câu chuyện là gì? Đầu tiên bạn cho chúng
tôi biết Một lớp, Một trách nhiệm, và bây giờ bạn
đưa cho chúng tôi một mẫu với hai trách nhiệm trong một
lớp. Mẫu tổng hợp quản lý một hệ thống phân cấp VÀ
thực hiện các hoạt động liên quan đến Menu.

Có một số sự thật trong quan sát đó. Chúng ta có thể nói rằng Composite
Pattern lấy nguyên tắc thiết kế Single Responsibility và đổi nó lấy tính minh bạch. Tính
minh bạch là gì? Vâng, bằng cách cho phép giao diện Component chứa các hoạt động
quản lý con và

Các hoạt động của lá, máy khách có thể xử lý cả thành phần hợp thành và nút lá một
cách thống nhất; do đó, việc một phần tử là thành phần hợp thành hay nút lá sẽ trở nên
minh bạch đối với máy khách.

Bây giờ, vì chúng ta có cả hai loại hoạt động trong lớp Component, chúng ta mất
đi một chút an toàn vì một máy khách có thể cố gắng làm điều gì đó không phù hợp hoặc
vô nghĩa trên một phần tử (như cố gắng thêm menu vào một mục menu). Đây là một quyết
định thiết kế; chúng ta có thể đưa thiết kế theo hướng ngược lại và tách các trách
nhiệm thành các giao diện.

Điều này sẽ làm cho thiết kế của chúng ta an toàn, theo nghĩa là bất kỳ lệnh gọi không
phù hợp nào tới các phần tử sẽ bị phát hiện tại thời điểm biên dịch hoặc thời gian chạy,
nhưng chúng ta sẽ mất đi tính minh bạch và mã của chúng ta sẽ phải sử dụng các điều kiện
và toán tử instanceof .

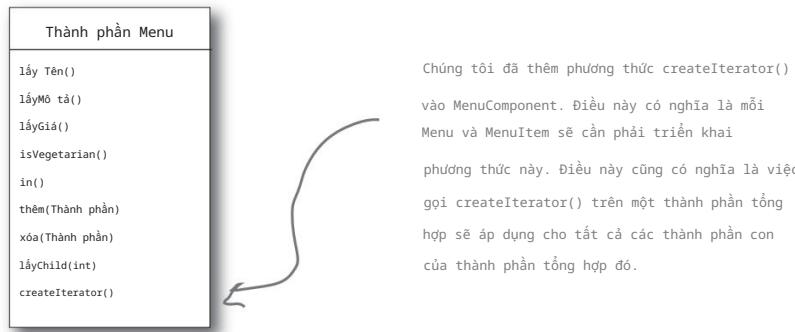
Vì vậy, để quay lại câu hỏi của bạn, đây là một trường hợp điển hình của sự đánh đổi.
Chúng ta được hướng dẫn bởi các nguyên tắc thiết kế, nhưng chúng ta luôn cần quan sát
tác động của chúng lên thiết kế của mình. Đôi khi chúng ta cố tình làm mọi thứ theo
cách có vẻ vi phạm nguyên tắc. Tuy nhiên, trong một số trường hợp, đây là vấn đề về
quan điểm; ví dụ, có vẻ không đúng khi có các hoạt động quản lý con trong các nút
lá (như add(), remove() và getChild()), nhưng một lần nữa, bạn luôn có thể thay đổi quan
điểm của mình và xem một lá như một nút không có con nào.

flash ashback đến iterator

Quay lại Iterator

Chúng tôi đã hứa với bạn ở một vài trang trước rằng chúng tôi sẽ chỉ cho bạn cách sử dụng Iterator với Composite. Bạn biết rằng chúng tôi đã sử dụng Iterator trong triển khai bộ của phương thức `print()`, nhưng chúng tôi cũng có thể cho phép Waitress lặp lại toàn bộ composite nếu cô ấy cần, ví dụ, nếu cô ấy muốn xem toàn bộ menu và lấy ra các món chay.

Để triển khai một trình lặp Composite, hãy thêm phương thức `createIterator()` vào mọi thành phần. Chúng ta sẽ bắt đầu với lớp trừu tượng `MenuComponent`:



Bây giờ chúng ta cần triển khai phương thức này trong các lớp `Menu` và `MenuItem`:

```
lớp công khai Menu mở rộng MenuComponent {
    Trình lặp iterator = null;
    // mã khác ở đây không thay đổi

    công khai Iterator createIterator() {
        nếu (trình lặp == null) {
            trình lặp = trình lặp CompositeIterator mới (menuComponents.iterator());
        }
        trả về trình lặp;
    }
}
```

←
Chúng ta chỉ cần một
trình lặp cho mỗi Menu.

Ở đây chúng ta sử dụng một trình lặp mới có tên là `CompositeIterator`. Trình lặp này biết cách lặp qua bất kỳ hợp phần nào.

←
Chúng ta truyền cho nó trình
lặp của hợp chất hiện tại.

```
lớp công khai MenuItem mở rộng MenuComponent {
```

```
// mã khác ở đây không thay đổi

công khai Iterator createIterator() {
    trả về NullIterator mới();
}
```

Bây giờ đến `MenuItem`...

←
Ồ! NullIterator này là gì thế?
Bạn sẽ thấy trong hai trang.

}

trình lặp và các mẫu tổng hợp

Trình lặp tổng hợp

CompositeIterator là một trình lặp NGHIÊM TÚC. Nó có nhiệm vụ lặp qua các MenuItem trong thành phần và đảm bảo tất cả các Menu con (và các Menu con con, v.v.) đều được bao gồm.

Đây là mã. Hãy cẩn thận, đây không phải là nhiều mã, nhưng nó có thể hơi khó hiểu. Chỉ cần lặp lại với chính mình khi bạn thực hiện nó "đệ quy là bạn của tôi, đệ quy là bạn của tôi."

```
nhập java.util.*;
```

```

lớp công khai CompositeIterator triển khai Iterator {
    Stack stack = new Stack();

    công khai CompositeIterator(Iterator iterator) {
        stack.push(trình lặp);
    }

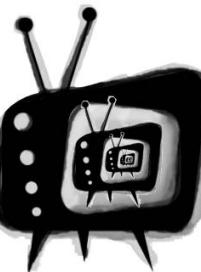
    Đối tượng công khai tiếp theo() {
        nếu (hasNext()) {
            Trình lặp iterator = (Trình lặp) stack.peek();
            Thành phần MenuComponent = (MenuComponent) iterator.next();
            nếu (thành phần thê hiện của Menu) {
                stack.push(thành phần.createIterator());
            }
            trả về thành phần;
        } khác {
            trả về giá trị null;
        }
    }

    boolean công khai hasNext() {
        nếu (stack.empty()) {
            trả về false;
        } khác {
            Trình lặp iterator = (Trình lặp) stack.peek();
            nếu (!iterator.hasNext()) {
                stack.pop();
                trả về hasNext();
            } khác {
                trả về giá trị đúng;
            }
        }
    }

    công khai void remove() {
        ném UnsupportedOperationException mới();
    }
}
```

Giống như tất cả các trình

lặp, chúng tôi đang triển khai giao diện
java.util.Iterator.



Hãy cẩn thận:

Đệ quy

Khu vực phía trước

Trình lặp của phần tử hợp thành cấp

cao nhất mà chúng ta sẽ lặp lại được truyền vào.
Chúng ta đưa trình lặp đó vào cấu trúc dữ
liệu ngắn xếp.

Được rồi, khi máy khách muốn lấy
phần tử tiếp theo, trước tiên chúng
ta phải đảm bảo rằng có phần tử tiếp
theo bằng cách gọi hasNext()...

Nếu có phần tử tiếp theo, chúng ta sẽ
lấy trình lặp hiện tại ra khỏi ngắn xếp và
lấy phần tử tiếp theo của nó.

Nếu phần tử đó là một menu, chúng ta có một
thành phần khác cần được đưa vào vòng lặp,
vì vậy chúng ta ném nó vào ngắn xếp. Trong
cả hai trường hợp, chúng ta trả về thành
phần.

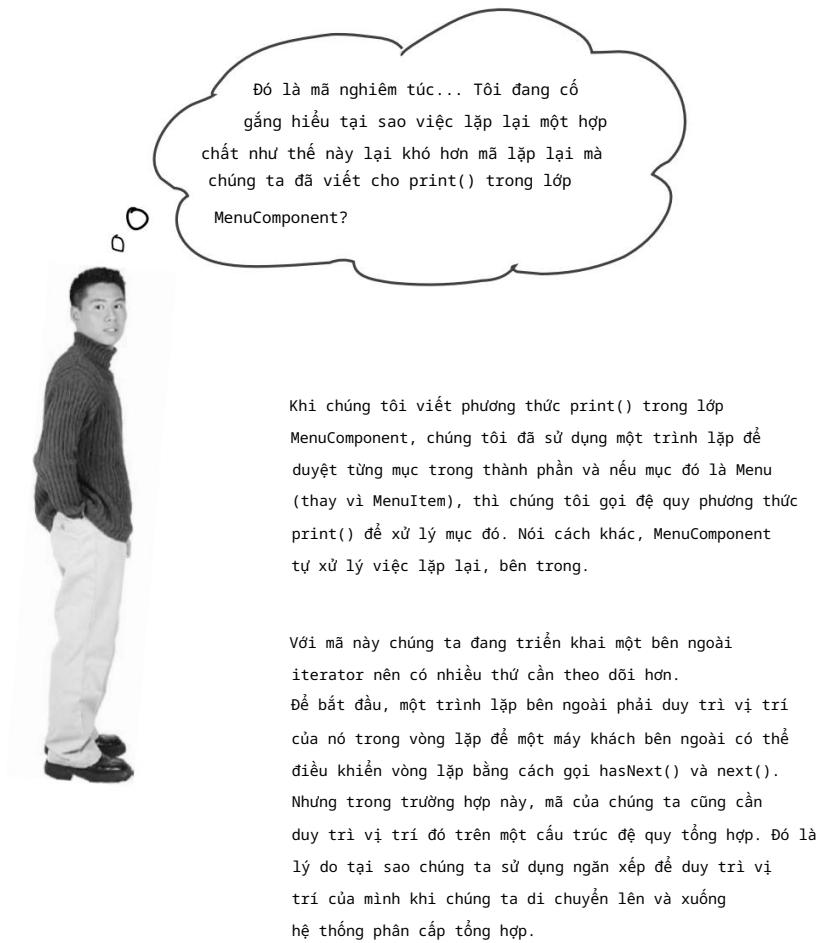
Để xem có phần tử tiếp theo hay không, chúng
ta kiểm tra xem ngắn xếp có trống không;
nếu có thì không có.

Nếu không, chúng ta lấy iterator ra khỏi
đầu ngắn xếp và xem nó có phần tử tiếp theo
không. Nếu không, chúng ta lấy iterator
ra khỏi ngắn xếp và gọi hasNext() đệ quy.

Nếu không thì có một phần tử tiếp theo và
chúng ta trả về giá trị true.

Chúng tôi không hỗ trợ xóa
mà chỉ hỗ trợ duyệt.

bên trong và bên ngoài



não Apower

Vẽ sơ đồ Menu và MenuItem. Sau đó, giả vờ bạn là CompositeIterator và nhiệm vụ của bạn là xử lý các lệnh gọi đến hasNext() và next(). Theo dõi cách CompositeIterator duyệt qua cấu trúc khi đoạn mã này được thực thi:

```
public void testCompositeIterator(Thành phần MenuComponent) { CompositeIterator iterator = new  
    CompositeIterator(Thành phần.iterator);  
  
    while(iterator.hasNext()) { Thành phần  
        MenuComponent = iterator.next();  
    }  
}
```

trình lặp null

Trình lặp Null

Được rồi, vậy Null Iterator này là gì? Hãy nghĩ theo cách này: MenuItem không có gì để lặp lại, đúng không? Vậy chúng ta xử lý việc triển khai phương thức createIterator() của nó như thế nào? Vâng, chúng ta có hai lựa chọn:

LƯU Ý: Một ví dụ khác về "Mẫu thiết kế" Đối tượng Null.

Lựa chọn một:

Trả về giá trị null

Chúng ta có thể trả về null từ createIterator(), nhưng sau đó chúng ta sẽ cần mã có điều kiện trong máy khách để xem null có được trả về hay không.

Lựa chọn thứ hai:

Trả về một trình lặp luôn trả về false khi hasNext()
được gọi

Có vẻ như đây là một kế hoạch tốt hơn. Chúng ta vẫn có thể trả về một trình lặp, nhưng máy khách không phải lo lắng về việc null có được trả về hay không. Trên thực tế, chúng ta đang tạo một trình lặp là "không có hoạt động".

Lựa chọn thứ hai chắc chắn có vẻ tốt hơn. Hãy gọi nó là NullIterator và triển khai nó.

```
nhập java.util.Iterator;
lớp công khai NullIterator thực hiện Iterator {
    Đối tượng công khai tiếp theo() {
        trả về giá trị null;
    }
    boolean công khai hasNext() {
        trả về false;
    }
    công khai void remove() {
        ném UnsupportedOperationException mới();
    }
}
```

Đây là Iterator lười biếng nhất mà bạn từng thấy, nó luôn chậm chạp ở mọi bước.

 Khi next() được gọi, chúng ta trả về null.

 Điều quan trọng nhất là khi hasNext() được gọi, chúng ta luôn trả về false.

 Vì NullIterator sẽ không nghĩ đến việc hỗ trợ xóa.

Cho tôi thực đơn chay

Bây giờ chúng ta có cách lắp lại từng mục trong Menu. Hãy lấy cách đó và cung cấp cho Waitress một phương thức có thể cho chúng ta biết chính xác những mục nào là đồ chay.

```

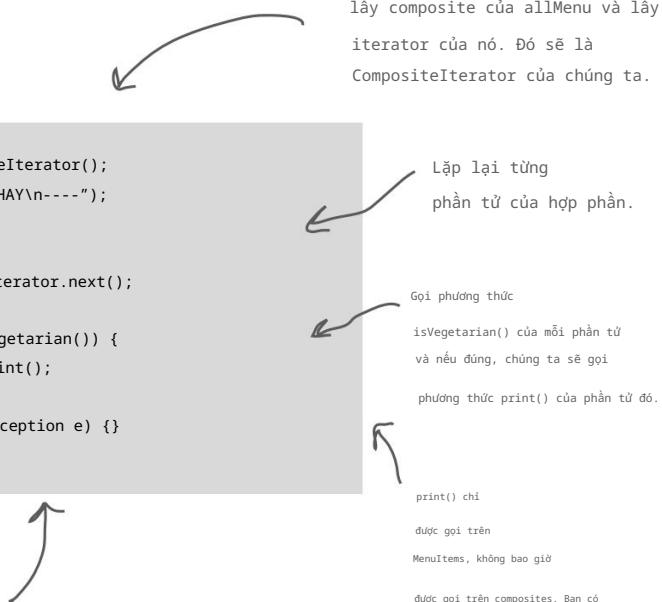
lớp công khai Waitress {
    MenuComponent tất cả Menu;

    có hầu bàn công cộng(MenuComponent allMenus) {
        this.allMenus = allMenus;
    }

    công khai void printMenu() {
        allMenus.print();
    }

    công khai void printVegetarianMenu() {
        Trình lắp iterator = allMenus.createIterator();
        System.out.println("\nTHỰC ĐƠN ĂN CHAY\n---");
        trong khi (iterator.hasNext()) {
            MenuComponent menuComponent =
                (MenuComponent) iterator.next();
            thử {
                nếu (menuComponent.isVegetarian()) {
                    menuComponent.print();
                }
            } bắt (UnsupportedOperationException e) {}
        }
    }
}

```



Chúng tôi đã triển khai isVegetarian() trên Menu để luôn ném ra ngoại lệ. Nếu điều đó xảy ra, chúng tôi sẽ bắt ngoại lệ, nhưng tiếp tục lắp lại.

phép thuật của iterator và composite

Sự kỳ diệu của Iterator và Composite khi kết hợp với nhau...

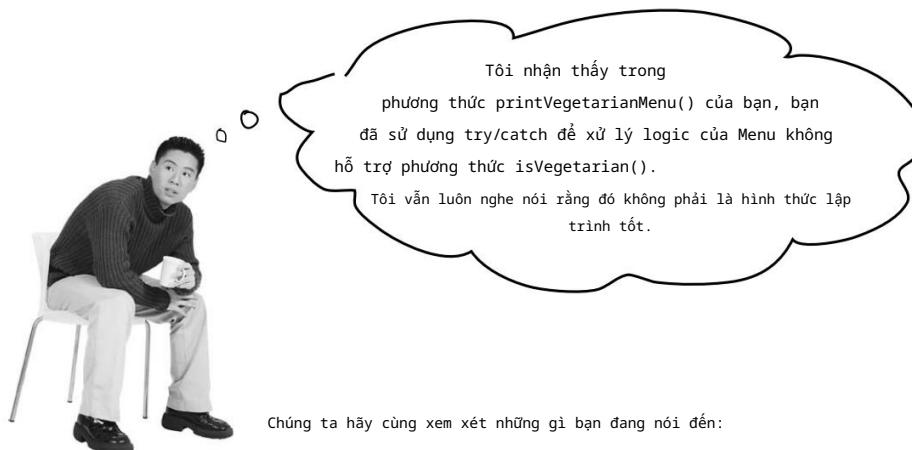
Whooo! Thật là một nỗ lực phát triển để đưa mã của chúng tôi đến được thời điểm này. Bây giờ chúng tôi đã có một cấu trúc menu chung có thể tồn tại trong đó chế Diner đang phát triển trong một thời gian. Bây giờ là lúc ngồi lại và gọi một số món ăn chay:

```
Trợ giúp về cửa sổ chỉnh sửa tệp HaveHungryIteratorToday?
% java MenuTestDrive
THỰC ĐƠN ĂN CHAY
-----
Bữa sáng bánh kếp của K&B(v), 2,99 -- Bánh
    kếp với trứng rán và bánh mì nướng Bánh kếp việt quất(v),
    3,49
        -- Bánh kếp làm từ quả việt quất tươi và siro việt quất Bánh quế (v), 3,59 -- Bánh
        quế, với quả việt
        quất hoặc đậu tây tùy chọn Bánh mì kẹp thịt chay (v), 2,99 -- Thịt xông khói (giả)
        với xà lách và cà chua trên lúa
        mì nguyên cám Rau hấp và gạo lứt (v), 3,99 -- Rau hấp trên gạo lứt Mì ống
        (v), 3,89 -- Mì Ý sốt Marinara và một lát bánh mì chua
        Bánh táo (v), 1,59 -- Bánh táo với lớp vỏ mỏng, phủ
        kem vani Bánh phô
        mai (v), 1,99 -- Bánh phô mai New York kem, với lớp vỏ sô cô la graham Sorbet (v), 1,89 --
        Một muỗng mâm xôi và một
        muỗng chanh Bánh táo (v), 1,59 -- Bánh táo với lớp vỏ mỏng, phủ kem vani Bánh phô mai
        (v), 1,99 -- Bánh phô mai
        New York kem, với lớp vỏ sô cô la graham Sorbet (v), 1,89 -- Một muỗng mâm xôi và
        một muỗng chanh
        Veggie Burger và Air Fries(v), 3,99 -- Veggie burger trên
        một chiếc bánh mì nguyên
        cám, rau diếp, cà chua và khoai tây chiên Burrito(v), 4,29 -- Một chiếc burrito lớn,
        với đậu pinto nguyên hạt,
        sốt salsa, guacamole

%
```



Thực đơn chay bao gồm các
món chay trong mọi thực đơn.



Chúng ta hãy cùng xem xét những gì bạn đang nói đến:

```
thử {
    nếu (menuComponent.isVegetarian()) {
        menuComponent.print();
    }
} bắt (UnsupportedOperationException) {}
```

Chúng tôi gọi isVegetarian() trên tất cả MenuComponents, nhưng Menus lại đưa ra ngoại lệ vì chúng không hỗ trợ hoạt động này.

Nếu thành phần menu không hỗ trợ thao tác, chúng ta chỉ cần bỏ qua ngoại lệ đó.

Nhin chung tôi đồng ý; try/catch dùng để xử lý lỗi, không phải logic chương trình. Các tùy chọn khác của chúng tôi là gì? Chúng tôi có thể kiểm tra loại thời gian chạy của thành phần menu bằng instanceof để đảm bảo đó là MenuItem trước khi gọi isVegetarian(). Nhưng trong quá trình này, chúng tôi sẽ mất đi tính minh bạch vì chúng ta sẽ không xử lý Menu và MenuItem một cách thống nhất.

Chúng ta cũng có thể thay đổi isVegetarian() trong Menus để nó trả về false. Điều này cung cấp một giải pháp đơn giản và chúng ta giữ được tính minh bạch của mình.

Trong giải pháp của chúng tôi, chúng tôi hướng đến sự rõ ràng: chúng tôi thực sự muốn truyền đạt rằng đây là một hoạt động không được hỗ trợ trên Menu (điều này khác với việc nói isVegetarian() là sai). Nó cũng cho phép ai đó đến và thực sự triển khai phương thức isVegetarian() hợp lý cho Menu và làm cho nó hoạt động với mã hiện có.

Đó là câu chuyện của chúng tôi và chúng tôi sẽ bám sát vào nó.

phỏng vấn với composite



Các mẫu được phơi bày

Cuộc phỏng vấn tuần này:

Mẫu tổng hợp, về các vấn đề triển khai

HeadFirst: Chúng tôi ở đây tối nay để nói chuyện với Composite Pattern. Tại sao bạn không kể cho chúng tôi một chút về bản thân mình, Composite?

Hợp thành: Chắc chắn rồi... Tôi là mẫu được sử dụng khi bạn có các tập hợp đối tượng có mối quan hệ toàn thể-bộ phận và bạn muốn có thể xử lý các đối tượng đó một cách thống nhất.

HeadFirst: Được rồi, chúng ta hãy đi sâu vào vấn đề này... ý bạn là gì khi nói đến mối quan hệ toàn thể-bộ phận?

Composite: Hãy tưởng tượng một giao diện người dùng đồ họa; ở đó bạn thường sẽ tìm thấy một thành phần cấp cao nhất như Frame hoặc Panel, chứa các thành phần khác, như menu, khung văn bản, thanh cuộn và nút. Vì vậy, GUI của bạn bao gồm một số phần, nhưng khi bạn hiển thị nó, bạn thường nghĩ về nó như một tổng thể. Bạn yêu cầu thành phần cấp cao nhất hiển thị và tin tưởng rằng thành phần đó sẽ hiển thị tất cả các phần của nó. Chúng tôi gọi các thành phần chứa các thành phần khác là đối tượng composite và các thành phần không chứa các thành phần khác là đối tượng leaf.

HeadFirst: Đó có phải là ý bạn muốn nói khi xử lý các đối tượng một cách đồng nhất không? Có những phương pháp chung mà bạn có thể gọi trên các hợp chất và lá không?

Composite: Đúng. Tôi có thể yêu cầu một đối tượng composite hiển thị hoặc một đối tượng leaf hiển thị và chúng sẽ làm đúng. Đối tượng composite sẽ hiển thị bằng cách yêu cầu tất cả các thành phần của nó hiển thị.

HeadFirst: Điều đó ngụ ý rằng mọi đối tượng đều có cùng giao diện. Nếu bạn có các đối tượng trong hợp chất của mình thực hiện những việc khác nhau thì sao?

Composite: Vâng, để composite hoạt động một cách minh bạch với máy khách, bạn phải triển khai cùng một giao diện cho tất cả các đối tượng trong composite, nếu không, máy khách sẽ phải lo lắng về việc mỗi đối tượng đang triển khai giao diện nào, điều này sẽ làm mất đi mục đích.

Rõ ràng là đôi khi bạn sẽ có những đối tượng mà một số lệnh gọi phương thức không có ý nghĩa.

HeadFirst: Vậy bạn xử lý việc đó thế nào?

Composite: Vâng, có một vài cách để xử lý; đôi khi bạn có thể không làm gì cả, hoặc trả về null hoặc false - bất cứ điều gì có ý nghĩa trong ứng dụng của bạn. Những lần khác, bạn sẽ muốn chủ động hơn và né tránh ngoại lệ. Tất nhiên, sau đó, khách hàng phải sẵn sàng làm một chút công việc và đảm bảo rằng lệnh gọi phương thức không làm điều gì đó bất ngờ.

HeadFirst: Nhưng nếu máy khách không biết họ đang xử lý loại đối tượng nào thì làm sao họ biết được phải gọi lệnh nào nếu không kiểm tra loại đối tượng?

Composite: Nếu bạn có một chút sáng tạo, bạn có thể cấu trúc các phương thức của mình sao cho các triển khai mặc định thực hiện một điều gì đó có ý nghĩa. Ví dụ, nếu máy khách gọi getChild(), trên composite thì điều này có ý nghĩa. Và nó cũng có ý nghĩa trên một lá, nếu bạn nghĩ về lá như một đối tượng không có con.

HeadFirst: À... thông minh đấy. Nhưng tôi nghe nói một số khách hàng rất lo lắng về vấn đề này, rằng họ yêu cầu các giao diện riêng biệt cho các đối tượng khác nhau để họ không được phép thực hiện các lệnh gọi phương thức vô nghĩa. Đó vẫn là Composite Pattern chứ?

Composite: Có. Đây là phiên bản an toàn hơn nhiều của Composite Pattern, nhưng nó yêu cầu máy khách phải kiểm tra kiểu của mọi đối tượng trước khi thực hiện lệnh gọi để đối tượng có thể được ép kiểu chính xác.

HeadFirst: Hãy cho chúng tôi biết thêm đôi chút về cách cấu trúc các đối tượng lá và hợp thành này.

Hợp thành: Thông thường là cấu trúc cây, một dạng phân cấp nào đó. Gốc là hợp thành cấp cao nhất và tất cả các con của nó đều là hợp thành hoặc nút lá.

HeadFirst: Trẻ em có bao giờ hướng mắt về phía cha mẹ mình không?

Composite: Đúng, một thành phần có thể có một con trỏ đến một thành phần cha để giúp việc duyệt cấu trúc dễ dàng hơn. Và, nếu bạn có một tham chiếu đến một thành phần con và bạn cần xóa nó, bạn sẽ cần phải yêu cầu thành phần cha xóa thành phần con. Có tham chiếu thành phần cha cũng giúp việc đó dễ dàng hơn.

HeadFirst: Thực sự có rất nhiều điều cần cân nhắc trong quá trình triển khai của bạn. Có vấn đề nào khác mà chúng ta nên nghĩ đến khi triển khai Composite Pattern không?

Composite: Thực ra có... một là sắp xếp các phần tử con. Nếu bạn có một phần tử hợp thành cần giữ các phần tử con theo một thứ tự cụ thể thì sao? Khi đó bạn sẽ cần một lược đồ quản lý tinh vi hơn để thêm và xóa các phần tử con, và bạn sẽ phải cẩn thận về cách bạn duyệt qua hệ thống phân cấp.

HeadFirst: Một điểm hay mà tôi chưa từng nghĩ tới.

Composite: Và bạn có nghĩ đến việc lưu trữ đệm không?

HeadFirst: Lưu trữ đệm?

Composite: Đúng vậy, lưu trữ đệm. Đôi khi, nếu cấu trúc composite phức tạp hoặc tốn kém để duyệt, thì việc triển khai lưu trữ đệm các nút composite sẽ hữu ích.

Ví dụ, nếu bạn liên tục duyệt một hợp phần và tất cả các phần tử con của nó để tính toán một kết quả nào đó, bạn có thể triển khai bộ nhớ đệm lưu trữ kết quả tạm thời để tiết kiệm các lần duyệt.

HeadFirst: Vâng, Composite Patterns còn nhiều điều hơn tôi từng nghĩ. Trước khi kết thúc, một câu hỏi nữa: Bạn cho rằng điểm mạnh nhất của mình là gì?

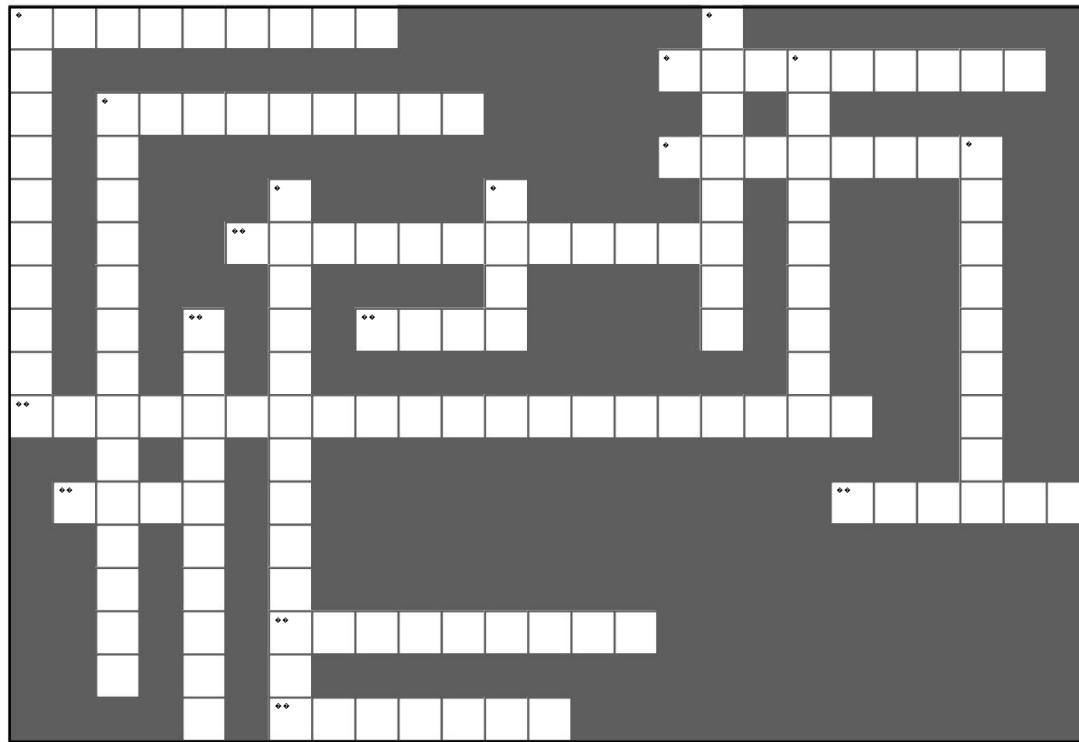
Composite: Tôi nghĩ tôi chắc chắn phải nói là đơn giản hóa cuộc sống cho khách hàng của tôi. Khách hàng của tôi không phải lo lắng về việc họ đang xử lý đối tượng composite hay đối tượng leaf, vì vậy họ không phải viết các câu lệnh if ở mọi nơi để đảm bảo họ đang gọi đúng phương thức trên đúng đối tượng. Thông thường, họ có thể thực hiện một lệnh gọi phương thức và thực hiện một hoạt động trên toàn bộ kết cấu.

HeadFirst: Nghe có vẻ như đây là một lợi ích quan trọng. Không còn nghi ngờ nữa, bạn là một mẫu hữu ích để thu thập và quản lý các đối tượng. Và, với điều đó, chúng ta đã hết thời gian... Cảm ơn rất nhiều vì đã tham gia cùng chúng tôi và hãy sớm quay lại với một Patterns Exposed khác.

trò chơi ô chữ



Lại đến thời điểm đó rồi...





Ghép mỗi mẫu với mô tả của nó:

Mẫu

Sự miêu tả

Chiến lược

Khách hàng xử lý các tập hợp đối
tương và các đối tượng riêng lẻ một
cách thống nhất

Bộ chuyển đổi

Cung cấp một cách để duyệt
qua một tập hợp các đối tượng
mà không làm lộ việc
triển khai tập hợp đó

Trình lắp lại

Đơn giản hóa giao diện của
một nhóm lớp

Mặt tiền

Thay đổi giao diện của một
hoặc nhiều lớp

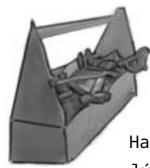
Hợp chất

Cho phép một nhóm đối tượng được
thông báo khi một số trạng thái
thay đổi

Người quan sát

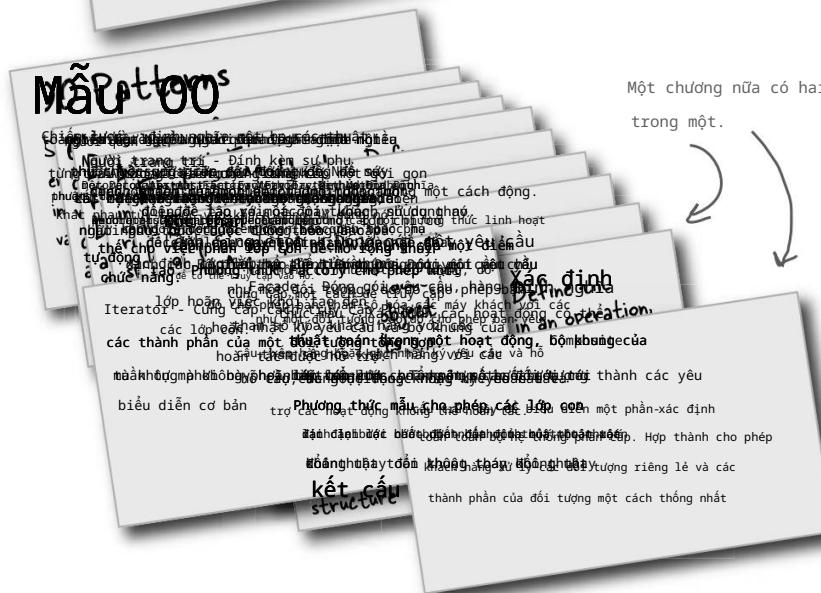
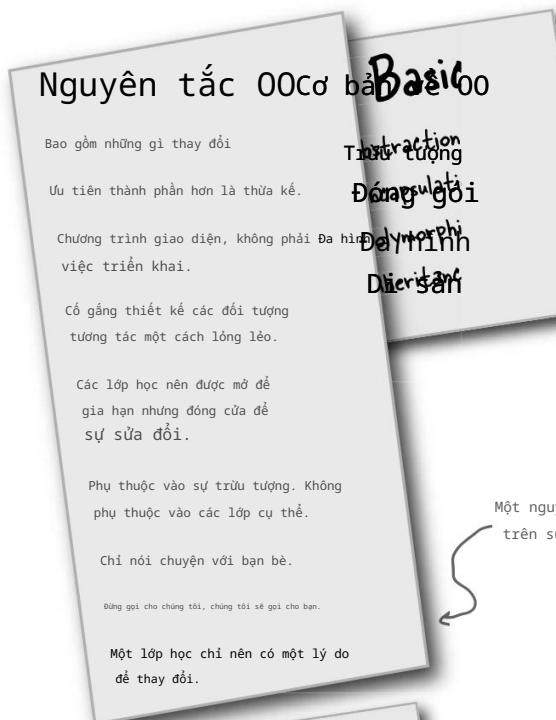
Đóng gói các hành vi có thể hoán
đổi cho nhau và sử dụng sự ủy quyền
để quyết định sử dụng hành vi nào

hộp công cụ thiết kế của bạn



Công cụ cho hộp công cụ thiết kế của bạn

Hai mẫu mới cho hộp công cụ của bạn - hai cách tuyệt vời để xử lý bộ sưu tập đối tượng.



ĐIỂM ĐẦU TIÊN

Iterator cho phép truy cập vào các phần tử của tổng hợp mà không cần tiết lộ cấu trúc bên trong của nó.

Iterator đảm nhiệm công việc lặp lại một tập hợp và đóng gói nó trong một đối tượng khác.

β Khi sử dụng Iterator, chúng ta
giải phóng tổng thể trách
nhiệm hỗ trợ các hoạt động
đọc và viết của nó.

Iterator cung cấp giao diện chung để duyệt qua các mục của một tổng hợp, cho phép bạn sử dụng đa hình khi viết mã sử dụng các mục của tổng hợp.

β Chúng ta nên cố gắng phân công
mỗi lớp chỉ có một trách nhiệm.

β Mẫu tổng hợp
cung cấp một cấu trúc để chứa
các vật thể riêng lẻ và
vật thể tổng hợp.

B Mẫu tổng hợp cho phép khách hàng xử lý vật liệu tổng hợp và các vật thể riêng lẻ một cách đồng nhất.

B Thành phần là bất kỳ đối tượng nào trong cấu trúc Hợp thành. Các thành phần có thể là các hợp chất khác hoặc các nút lá.

- β Có nhiều thiết kế
đánh đổi khi triển khai
Composite. Bạn cần cân bằng
tính minh bạch và an toàn
với nhu cầu của mình.

trình lặp và các mẫu tổng hợp



Giải pháp bài tập



Chuốt bút chì của bạn

Dựa trên việc triển khai printMenu() của chúng ta, điều nào sau đây được áp dụng?

- A. Chúng tôi đang mã hóa cho các triển khai cụ thể của PancakeHouseMenu và DinerMenu, không phải cho một giao diện.
- B. Có hầu bàn không thực hiện Java Waitress API và do đó không tuân theo một tiêu chuẩn nào.
- C. Nếu chúng ta quyết định chuyển từ sử dụng DinerMenu là một loại menu khác triển khai danh sách các mục menu bằng Hashtable, chúng ta sẽ phải sửa đổi rất nhiều mã trong Waitress.
- D. Người phục vụ cần biết cách mỗi menu biểu thị tập hợp các mục menu nội bộ của nó được triển khai, điều này vi phạm tính đóng gói.
- E. Chúng ta có mã trùng lặp: phương thức printMenu() cần hai triển khai vòng lặp riêng biệt để lặp qua hai loại menu khác nhau. Và nếu chúng ta thêm menu thứ ba, chúng ta có thể phải thêm một vòng lặp nữa.
- F. Việc thực hiện không dựa trên MXML (Menu XML) và do đó không có khả năng tương tác như mong đợi.



Chuốt bút chì của bạn

Trước khi lật trang, hãy nhanh chóng ghi ra ba điều chúng ta phải làm với đoạn mã này để phù hợp với khuôn khổ của chúng ta:

1. triển khai giao diện Menu
2. loại bỏ getItems()
3. thêm createIterator() và trả về một Iterator có thể bước qua các giá trị Hashtable

giải bài tập



Giải pháp Code Magnets

Trình lặp DinerMenu “Alternating” không bị xáo trộn

```
nhập java.util.Iterator;
nhập java.util.Calendar;
```

lớp công khai AlternatingDinerMenuItemator

thực hiện Iterator

```
MenuItem[] mục; int
vị trí;
```

```
public AlternatingDinerMenuItemator(MenuItem[] items)
```

}

```
this.items = items;
Lịch rightNow = Calendar.getInstance(); vị
trí = rightNow.get(Calendar.DAY_OF_WEEK) % 2;
```

}

boolean công khai hasNext() {

```
nếu (vị trí >= items.length || items[vị trí] == null) {
    trả về false; } else
{ trả về true;
}
```

}

Đối tượng công khai tiếp theo() {

```
MenuItem menuItem = items[vị trí]; vị trí = vị trí + 2; trả về
menuItem;
```

}

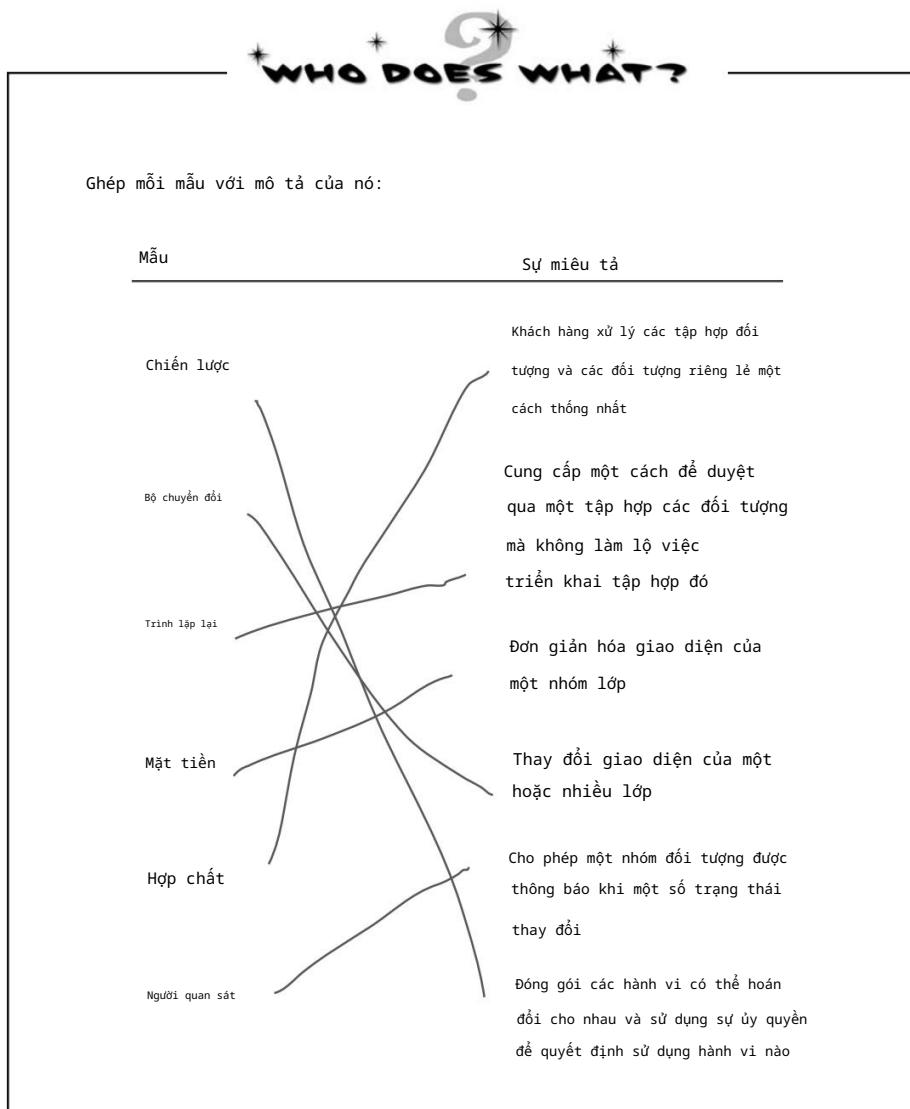
công khai void remove() {

mới(UnSupportedException) mới(
 “Trình lặp menu diner thay thế không hỗ trợ remove()”);

}

}

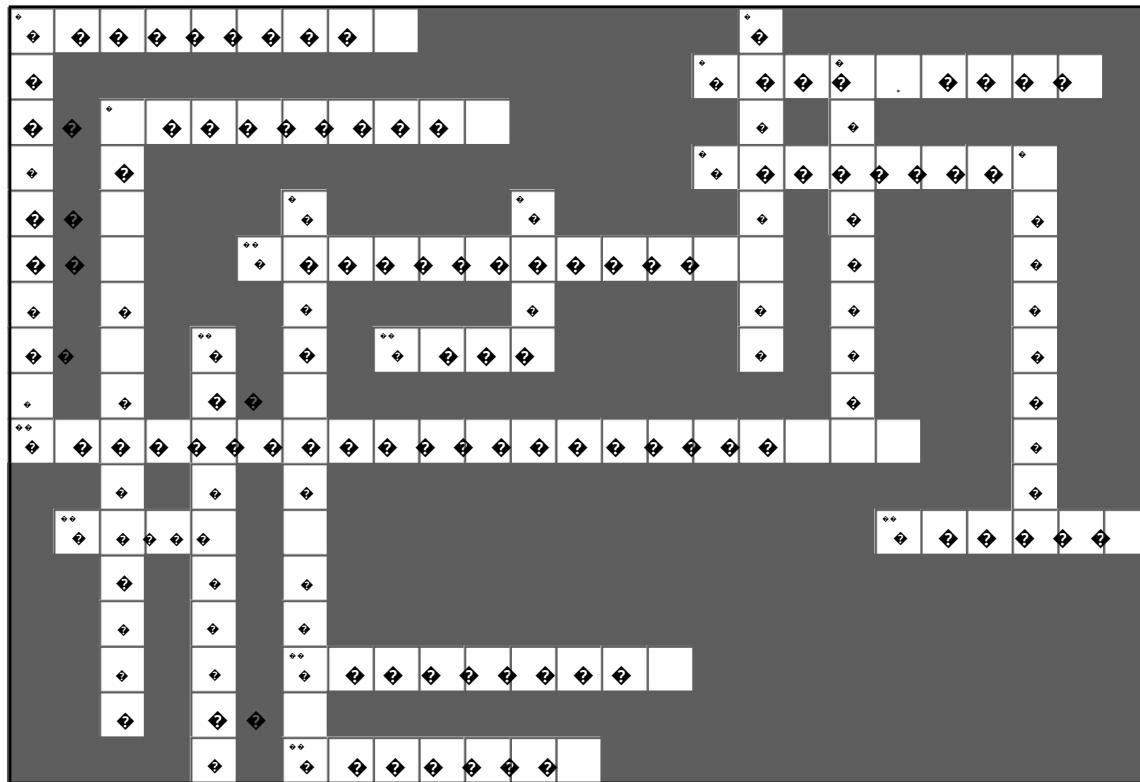
Lưu ý rằng việc triển khai Iterator này không hỗ trợ remove()



giải ô chữ



Giải pháp bài tập



.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

10 Mẫu Nhà Nước

h
g Các Tình trạng của Đồ đặc g



Tôi nghĩ mọi thứ trong Objectville sẽ
rất dễ dàng, nhưng giờ mỗi lần tôi quay lại
thì lại có một yêu cầu thay đổi khác được
gửi đến.
Tôi sắp đến giới hạn rồi! Ô, có lẽ tôi nên đến
nhóm mẫu tối thứ tư của Betty ngay từ đầu.
Tôi đang trong tình trạng như vậy!

Một sự thật ít người biết: Mẫu Chiến lược và Mẫu Nhà nước là cặp song sinh tách ra khi mới sinh. Như bạn đã biết, Mẫu Chiến lược đã tiếp tục tạo ra một doanh nghiệp thành công xung quanh các thuật toán có thể hoán đổi cho nhau. Tuy nhiên, Nhà nước đã có thể con đường cao quý hơn là giúp các đối tượng kiểm soát hành vi của chúng bằng cách thay đổi nội tại của chúng trạng thái. Người ta thường nghe thấy anh ta nói với khách hàng đối tượng của mình rằng, "Chi cần lập lại theo tôi: Tôi ôn dù rồi, tôi đủ thông minh rồi, và đồ khốn nạn..."

gặp gumball vĩ đại

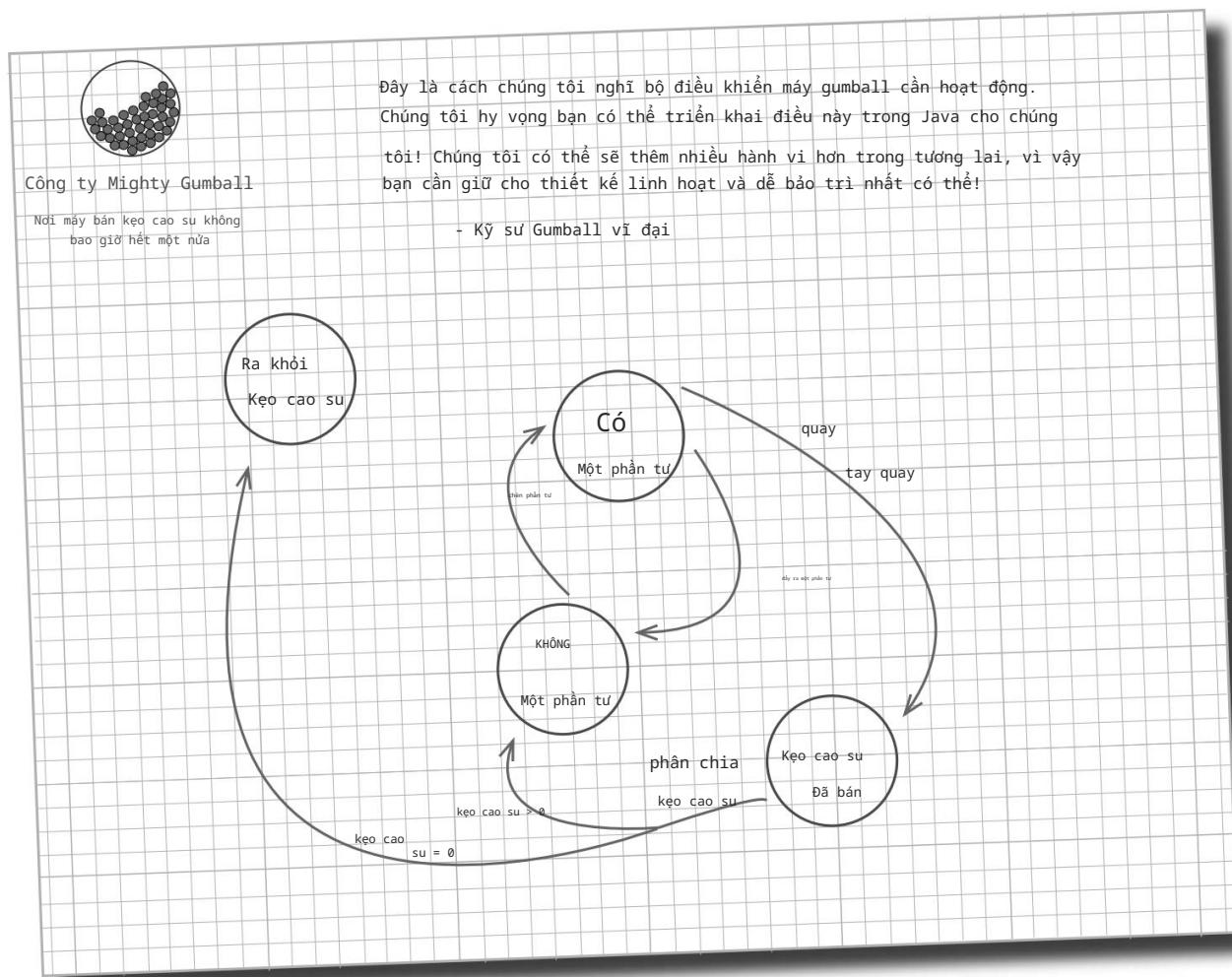
Máy bán kẹo và bể hàm

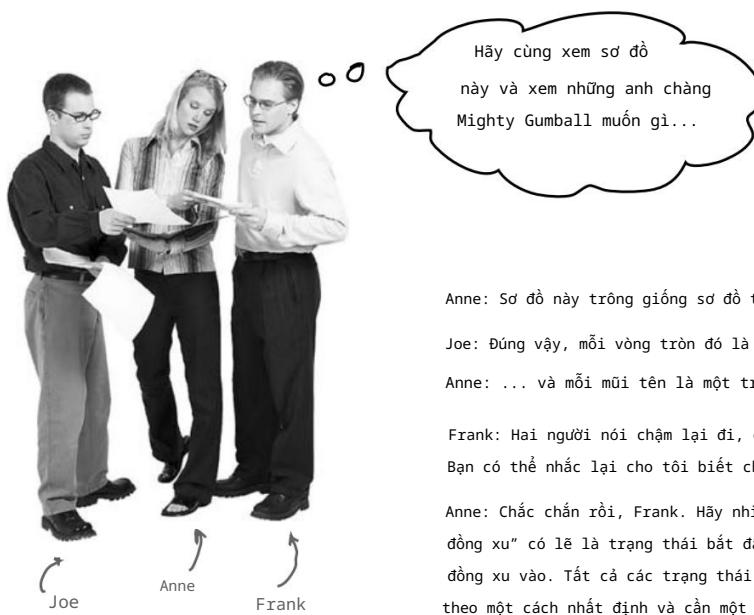
Máy nướng bánh mì Java là của thập niên 90. Ngày nay, mọi người đang xây dựng Java thành các thiết bị thực, như máy bán kẹo cao su. Đúng vậy, máy bán kẹo cao su đã trở thành công nghệ cao; các nhà sản xuất lớn đã phát hiện ra rằng bằng cách đưa CPU vào máy của họ, họ có thể tăng doanh số, theo dõi hàng tồn kho qua mạng và đo lường sự hài lòng của khách hàng chính xác hơn.

Nhưng những nhà sản xuất này là chuyên gia về máy bán kẹo cao su, không phải là nhà phát triển phần mềm và họ đã yêu cầu sự giúp đỡ của bạn:



Ít nhất thì đó là câu chuyện của họ
- chúng tôi nghĩ rằng họ chỉ cảm thấy chán với công nghệ của khoảng những năm 1800 và cần tìm cách để kiếm sống. thú vị hơn.





Cuộc trò chuyện trong ô

Anne: Sơ đồ này trông giống sơ đồ trạng thái.

Joe: Đúng vậy, mỗi vòng tròn đó là một trạng thái...

Anne: ... và mỗi mũi tên là một trạng thái chuyển đổi.

Frank: Hai người nói chậm lại đi, đã quá lâu rồi tôi không nghiên cứu sơ đồ trạng thái. Bạn có thể nhắc lại cho tôi biết chúng là gì không?

Anne: Chắc chắn rồi, Frank. Hãy nhìn vào các vòng tròn; đó là các trạng thái. "Không có đồng xu" có lẽ là trạng thái bắt đầu của máy kẹo cao su vì nó chỉ nằm đó chờ bạn bỏ đồng xu vào. Tất cả các trạng thái chỉ là các cấu hình khác nhau của máy hoạt động theo một cách nhất định và cần một số hành động đưa họ đến một trạng thái khác.

Joe: Đúng rồi. Xem này, để đi đến một tiểu bang khác, bạn cần phải làm gì đó như bỏ một đồng 25 xu vào máy. Bạn thấy mũi tên từ "Không có đồng 25 xu" đến "Có đồng 25 xu không?"

Frank: Vâng...

Joe: Điều đó chỉ có nghĩa là nếu máy bán kẹo cao su ở trạng thái "Không có đồng xu" và bạn bỏ một đồng xu vào, nó sẽ chuyển sang trạng thái "Có đồng xu". Đó là quá trình chuyển đổi trạng thái.

Frank: À, tôi hiểu rồi! Và nếu tôi đang ở trạng thái "Có đồng xu", tôi có thể xoay tay quay và chuyển sang trạng thái "Bán kẹo cao su", hoặc đẩy đồng xu ra và chuyển lại trạng thái "Không có đồng xu".

Anne: Bạn hiểu rồi!

Frank: Vậy thì trông không tệ lắm. Rõ ràng là chúng ta có bốn trạng thái, và tôi nghĩ chúng ta cũng có bốn hành động: "chèn đồng xu", "tháo đồng xu", "quay tay quay" và "phân phối". Nhưng... khi chúng ta phân phối, chúng ta kiểm tra xem có không hoặc nhiều viên kẹo cao su ở trạng thái "Đã bán kẹo cao su", rồi chuyển sang trạng thái "Hết kẹo cao su" hoặc trạng thái "Không có đồng xu". Vì vậy, chúng ta thực sự có năm lần chuyển đổi từ trạng thái này sang trạng thái khác.

Anne: Bài kiểm tra không có hoặc nhiều viên kẹo cao su cũng ngụ ý rằng chúng ta cũng phải theo dõi số lượng kẹo cao su. Bất cứ khi nào máy đưa cho bạn một viên kẹo cao su, đó có thể là viên cuối cùng, và nếu đúng như vậy, chúng ta cần chuyển sang trạng thái "Hết kẹo cao su".

Joe: Ngoài ra, đừng quên rằng bạn có thể làm những điều vô nghĩa, như cố gắng đẩy đồng xu 25 xu ra khi máy bán kẹo cao su đang ở trạng thái "Không có đồng xu 25 xu" hoặc nhét hai đồng xu 25 xu vào.

Frank: Ô, tôi không nghĩ tới điều đó; chúng ta cũng phải lo liệu những việc đó thôi.

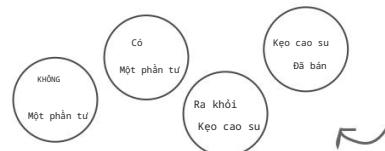
Joe: Đối với mọi hành động có thể, chúng ta chỉ cần kiểm tra xem chúng ta đang ở trạng thái nào và hành động phù hợp. Chúng ta có thể làm điều này! Hãy bắt đầu ánh xạ sơ đồ trạng thái vào mã...

dánh giá các máy trạng thái

Máy trạng thái 101

Làm thế nào chúng ta có thể chuyển từ sơ đồ trạng thái đó sang mã thực tế? Sau đây là phần giới thiệu nhanh về cách triển khai máy trạng thái:

- 1 Đầu tiên, hãy thu thập các trạng thái của bạn:



Sau đây là các tiêu bang - tổng cộng có bốn tiêu bang.

- 2 Tiếp theo, tạo một biến thể hiện để lưu trữ trạng thái hiện tại và xác định giá trị cho từng trạng thái:

Chúng ta hãy gọi là "Out of Gumballs"
"Đã bán hết" là viết tắt.

```

int tinh cuoi cung SOLD_OUT = 0;
int tinh cuoi cung NO_QUARTER = 1;
int tinh cuoi cung HAS_QUARTER = 2;
int tinh cuoi cung SOLD = 3;

int state = ĐÃ BÁN HẾT;
  
```

Sau đây là mỗi trạng thái được biểu diễn dưới dạng một số nguyên duy nhất...

...và đây là một biến thể hiện trạng thái hiện tại. Chúng ta sẽ tiếp tục và đặt nó thành "Đã bán hết" vì máy sẽ không có hàng khi lần đầu tiên được lấy ra khỏi hộp và bật lên.

- 3 Nay giờ chúng ta tập hợp tất cả các hành động có thể xảy ra trong hệ thống:

chèn phần tư

quay tay quay

đẩy ra một phần tư

Những hành động này

là giao diện của máy bán kẹo cao su - những việc bạn có thể làm với nó.

phân chia

Phân phối là một hành động nội bộ mà máy tự thực hiện.

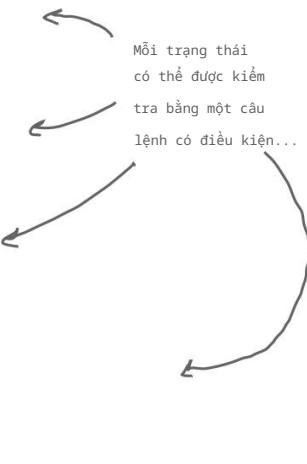
Nhìn vào sơ đồ, việc thực hiện bất kỳ hành động nào trong số này đều gây ra sự chuyển đổi trạng thái.

4

Bây giờ chúng ta tạo một lớp hoạt động như máy trạng thái. Đối với mỗi hành động, chúng ta tạo một phương thức sử dụng các câu lệnh có điều kiện để xác định hành vi nào là phù hợp trong mỗi trạng thái. Ví dụ, đối với hành động chèn một phần tư, chúng ta có thể viết một phương thức như thế này:

```
công khai void insertQuarter() {  
  
    nếu (trạng thái == HAS_QUARTER) {  
  
        System.out.println("Bạn không thể chèn thêm đồng 25 xu nữa");  
  
    } else if (state == ĐÃ BÁN HẾT) {  
  
        System.out.println("Bạn không thể nhét đồng 25 xu vào vì máy đã bán hết");  
  
    } else if (trạng thái == ĐÃ BÁN) {  
  
        System.out.println("Vui lòng đợi, chúng tôi đang tặng bạn một viên kẹo cao su");  
  
    } else if (trạng thái == NO_QUARTER) {  
  
        trạng thái = HAS_QUARTER;  
        System.out.println("Bạn đã chèn một đồng 25 xu");  
  
    }  
}
```

...nhưng cũng có thể chuyển sang các
trạng thái khác, như được mô tả trong sơ đồ.



...và thể hiện hành vi thích hợp
cho từng trạng thái có thể xảy ra...

Ở đây chúng ta sẽ nói
về một kỹ thuật phổ biến: mô
hình hóa trạng thái trong một đối
tượng bằng cách tạo một biến thể hiện để lưu
trữ các giá trị trạng thái và viết mã có điều
kiện trong các phương thức của chúng ta
để xử lý các trạng thái khác nhau.



Với bài đánh giá nhanh đó, chúng ta hãy cùng triển khai Máy Gumball nhé!

thực hiện máy kẹo cao su

Viết mã

Đã đến lúc triển khai Gumball Machine. Chúng ta biết rằng chúng ta sẽ có một biến thể hiện giữ trạng thái hiện tại. Từ đó, chúng ta chỉ cần xử lý tất cả các hành động, hành vi và chuyển đổi trạng thái có thể xảy ra. Đối với các hành động, chúng ta cần triển khai việc chèn một đồng 25 xu, lấy ra một đồng 25 xu, xoay tay quay và phân phối một viên kẹo cao su; chúng ta cũng có điều kiện viên kẹo cao su rỗng để triển khai.

```
lớp công khai GumballMachine {
    int tĩnh cuối cùng SOLD_OUT = 0;
    int tĩnh cuối cùng NO_QUARTER = 1;
    int tĩnh cuối cùng HAS_QUARTER = 2;
    int tĩnh cuối cùng SOLD = 3;

    int state = ĐÃ BÁN HẾT;
    int đếm = 0;

    công khai GumballMachine(int count) {
        this.count = đếm;
        nếu (số lượng > 0) {
            trạng thái = NO_QUARTER;
        }
    }
}
```

Dưới đây là bốn tiêu bang; chúng khớp với các tiêu bang trong sơ đồ tiểu bang của Mighty Gumball.

Đây là biến thể hiện sẽ theo dõi trạng thái hiện tại của chúng ta. Chúng ta bắt đầu ở trạng thái SOLD_OUT.

Chúng ta có một biến thể hiện thứ hai để theo dõi số lượng kẹo cao su trong máy.

Trình xây dựng sẽ kiểm kê ban đầu các viên kẹo cao su. Nếu kiểm kê không phải là số không, máy sẽ vào trạng thái NO_QUARTER, nghĩa là máy đang chờ ai đó nhét một đồng 25 xu vào, nếu không máy sẽ ở trạng thái SOLD_OUT.

Bây giờ chúng ta bắt đầu triển khai các hành động như phương thức....

Khi chèn một phần tử, nếu...

một phần tử đã được chèn vào chúng tôi nói với khách hàng;

nếu không, chúng tôi chấp nhận quý và chuyển sang trạng thái HAS_QUARTER.

```
công khai void insertQuarter() {
    nếu (trạng thái == HAS_QUARTER) {
        System.out.println("Bạn không thể chèn thêm đồng 25 xu nữa");
    } else if (trạng thái == NO_QUARTER) {
        trạng thái = HAS_QUARTER;
        System.out.println("Bạn đã chèn một đồng 25 xu");
    } else if (state == ĐÃ BÁN HẾT) {
        System.out.println("Bạn không thể nhét đồng 25 xu vào vì máy đã bán hết");
    } else if (trạng thái == ĐÃ BÁN) {
        System.out.println("Vui lòng đợi, chúng tôi đang tặng bạn một viên kẹo cao su");
    }
}
```

Nếu khách hàng vừa mua một viên kẹo cao su, họ cần phải đợi cho đến khi giao dịch hoàn tất trước khi cho thêm đồng xu 25 xu nữa vào.

và nếu máy đã bán hết, chúng tôi sẽ từ chối quý đó.

mô hình trạng thái

```

công khai void ejectQuarter() {
    nếu (trạng thái == HAS_QUARTER) {
        System.out.println("Quý trả về");
        trạng thái = NO_QUARTER;
    } else if (trạng thái == NO_QUARTER) {
        System.out.println("Bạn chưa chèn đồng 25 xu");
    } else if (trạng thái == ĐÃ BÁN) {
        System.out.println("Xin lỗi, bạn đã quay tay quay rồi");
    } else if (state == ĐÃ BÁN HẾT) {
        System.out.println("Bạn không thể đẩy ra vì bạn chưa cho đồng 25 xu vào");
    }
}

công khai void turnCrank() {
    nếu (trạng thái == ĐÃ BÁN) {
        System.out.println("Quay lại hai lần cũng không giúp bạn nhận được thêm viên kẹo cao su!");
    } else if (trạng thái == NO_QUARTER) {
        System.out.println("Bạn đã quay nhưng không có đồng xu nào cả");
    } else if (state == ĐÃ BÁN HẾT) {
        System.out.println("Bạn đã quay lại, nhưng không có kẹo cao su nào cả");
    } nếu không thì nếu (trạng thái == HAS_QUARTER) {
        System.out.println("Bạn đã quay...");
        trạng thái = ĐÃ BÁN;
        phân chia();
    }
}

công khai void dispense() {
    nếu (trạng thái == ĐÃ BÁN) {
        System.out.println("Một viên kẹo cao su lăn ra khỏi khe cắm");
        đếm = đếm - 1;
        nếu (đếm == 0) {
            System.out.println("Ồ, hết kẹo cao su rồi!");
            trạng thái = ĐÃ BÁN HẾT;
        } khác {
            trạng thái = NO_QUARTER;
        }
    } else if (trạng thái == NO_QUARTER) {
        System.out.println("Bạn cần phải thanh toán trước");
    } else if (state == ĐÃ BÁN HẾT) {
        System.out.println("Không phát kẹo cao su");
    } nếu không thì nếu (trạng thái == HAS_QUARTER) {
        System.out.println("Không phát kẹo cao su");
    }
}

// các phương thức khác ở đây như toString() và refill()
}

```

Bây giờ, nếu khách hàng cố gắng lấy đi đồng xu...

Nếu có một phần tư,
chúng ta trả lại nó và
quay lại trạng thái NO_QUARTER.

Nếu không, nếu không có thì
chúng tôi không thể trả lại được.

Bạn không thể đẩy máy ra nếu máy đã bán
hết, máy không chấp nhận tiền xu!

Nếu khách hàng chỉ
quay tay quay, chúng tôi
không thể hoàn lại tiền;
anh ta đã có kẹo cao su rồi!

Khách hàng cố gắng quay tay quay...

Có người đang cố gắng gian lận máy.

Chúng ta cần một
quý đầu tiên.

Được gọi để phát kẹo cao su.

Chúng tôi không thể
giao kẹo cao su; ở đó
không có gì.

Thành công! Họ nhận được một viên kẹo cao
su. Đổi trạng thái thành ĐÃ BÁN và
gọi phương thức dispense() của máy.

Chúng tôi đang ở
trạng thái BÁN; hãy tặng
họ một viên kẹo cao su!

Đây là nơi chúng ta xử lý tình trạng
"hết kẹo cao su": Nếu đây là tình trạng
cuối cùng, chúng ta đặt trạng thái của
máy thành SOLD_OUT; nếu không, chúng
ta sẽ không có đồng 25 xu.

Không điều nào trong số này
nên xảy ra, nhưng nếu có,
chúng ta sẽ cho họ một lỗi chứ
không phải một viên kẹo cao su.

thủ máy bán kẹo cao su

Kiểm tra nội bộ

Cảm giác như một thiết kế chắc chắn đẹp mắt sử dụng phương pháp được cân nhắc kỹ lưỡng phải không? Chúng ta hãy tiến hành một chút thử nghiệm nội bộ trước khi giao cho Mighty Gumball để đưa vào máy gumball thực tế của họ. Đây là dây nịt thử nghiệm của chúng tôi:

```

lớp công khai GumballMachineTestDrive {
    public static void main(String[] args) {
        GumballMachine gumballMachine = GumballMachine mới(5);

        System.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();

        System.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.ejectQuarter();
        gumballMachine.turnCrank();

        System.out.println(gumballMachine);

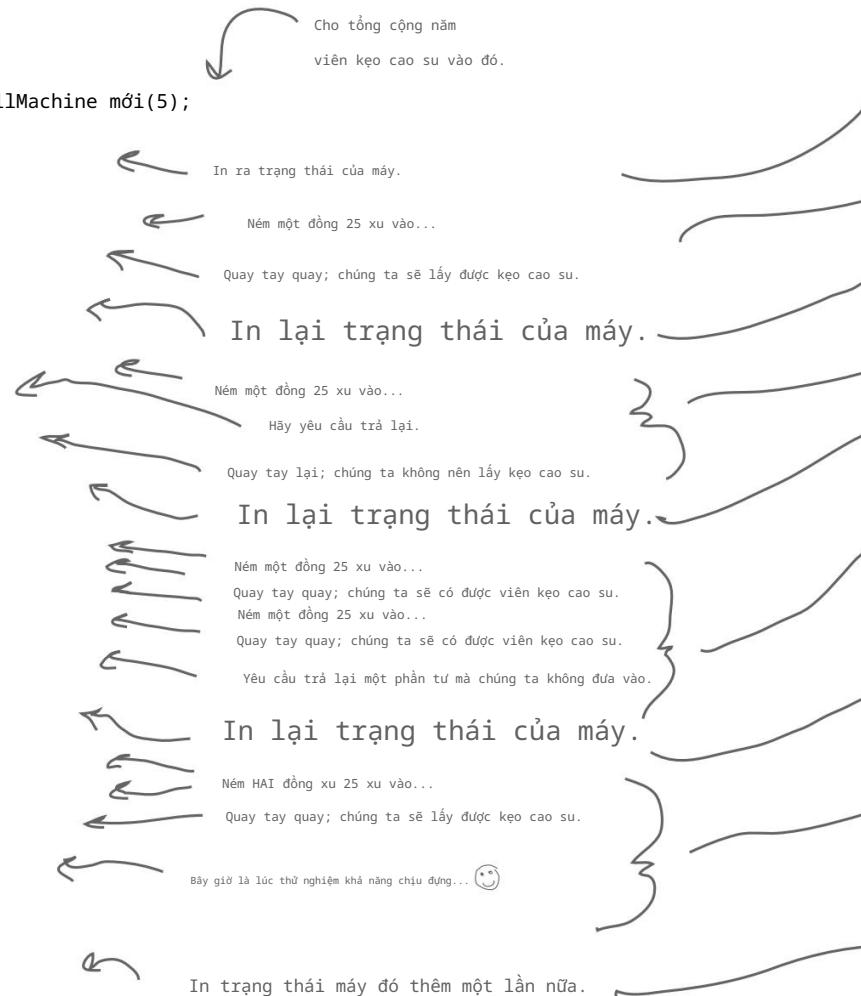
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.ejectQuarter();

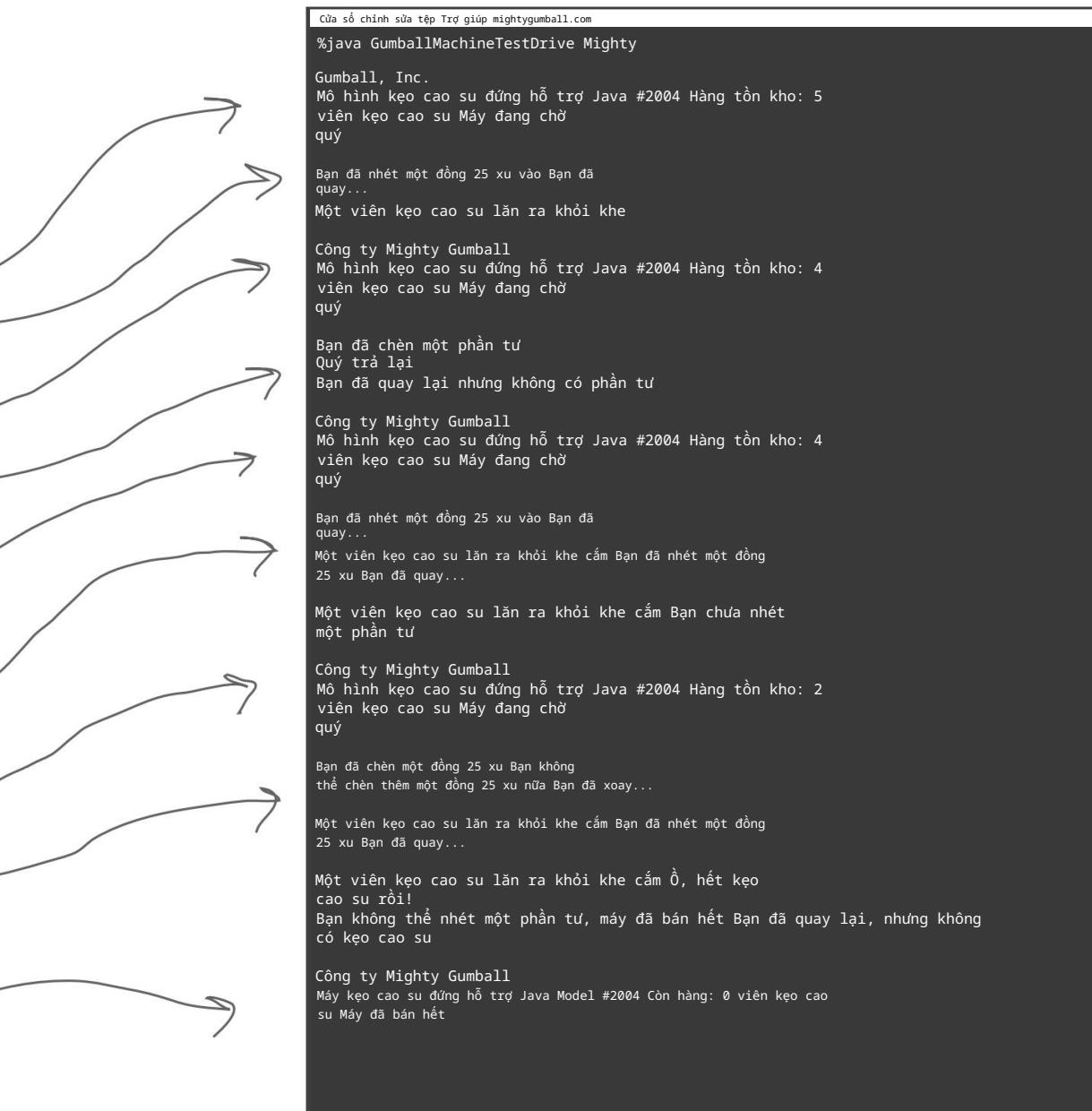
        System.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();

        System.out.println(gumballMachine);
    }
}

```



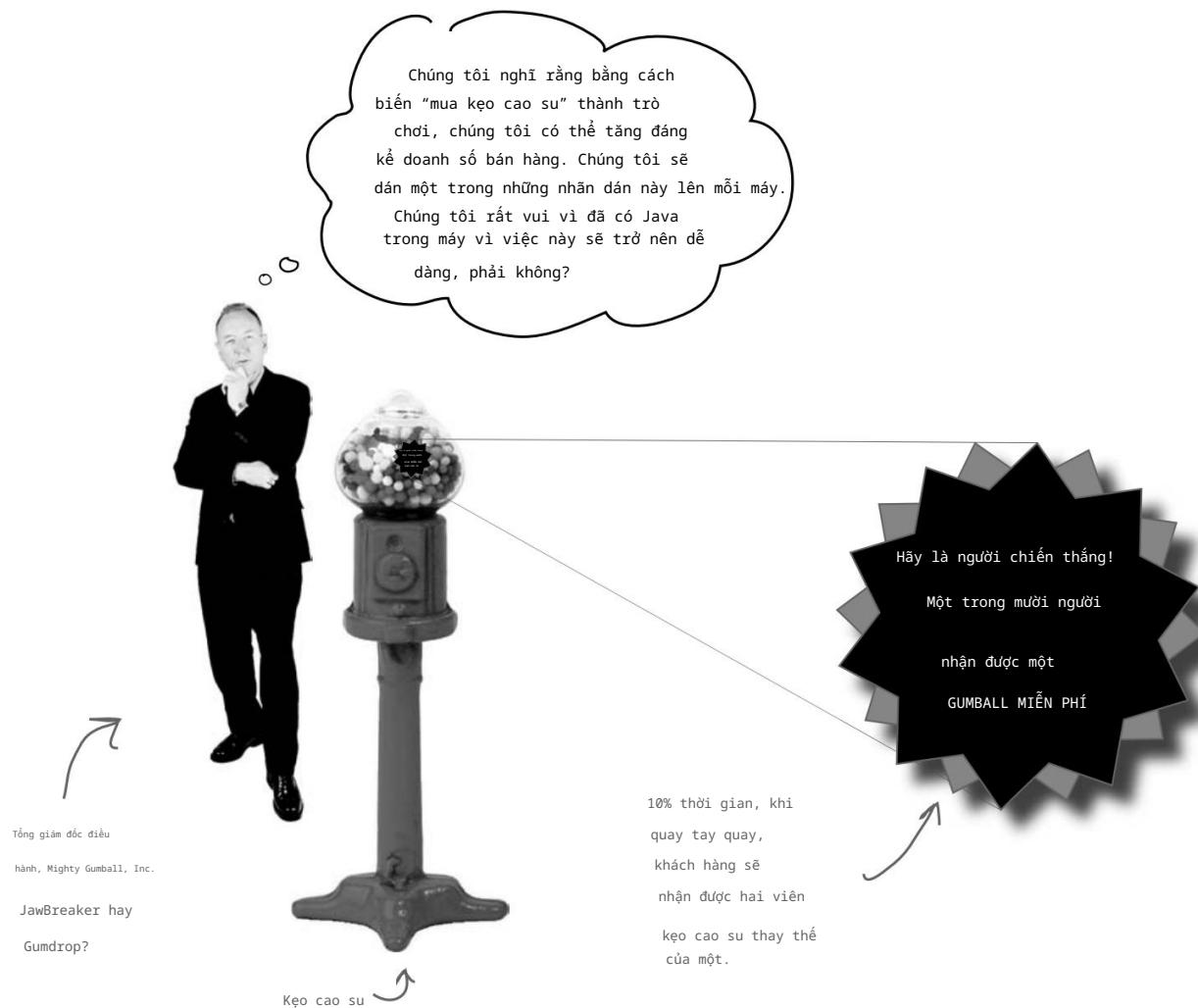


trò chơi mua kẹo cao su

Bạn biết điều đó sắp xảy ra... một yêu cầu thay đổi!

Mighty Gumball, Inc. đã tải mã của bạn vào máy mới nhất của họ và các chuyên gia đảm bảo chất lượng của họ đang kiểm tra nó. Cho đến nay, mọi thứ đều tuyệt vời theo quan điểm của họ.

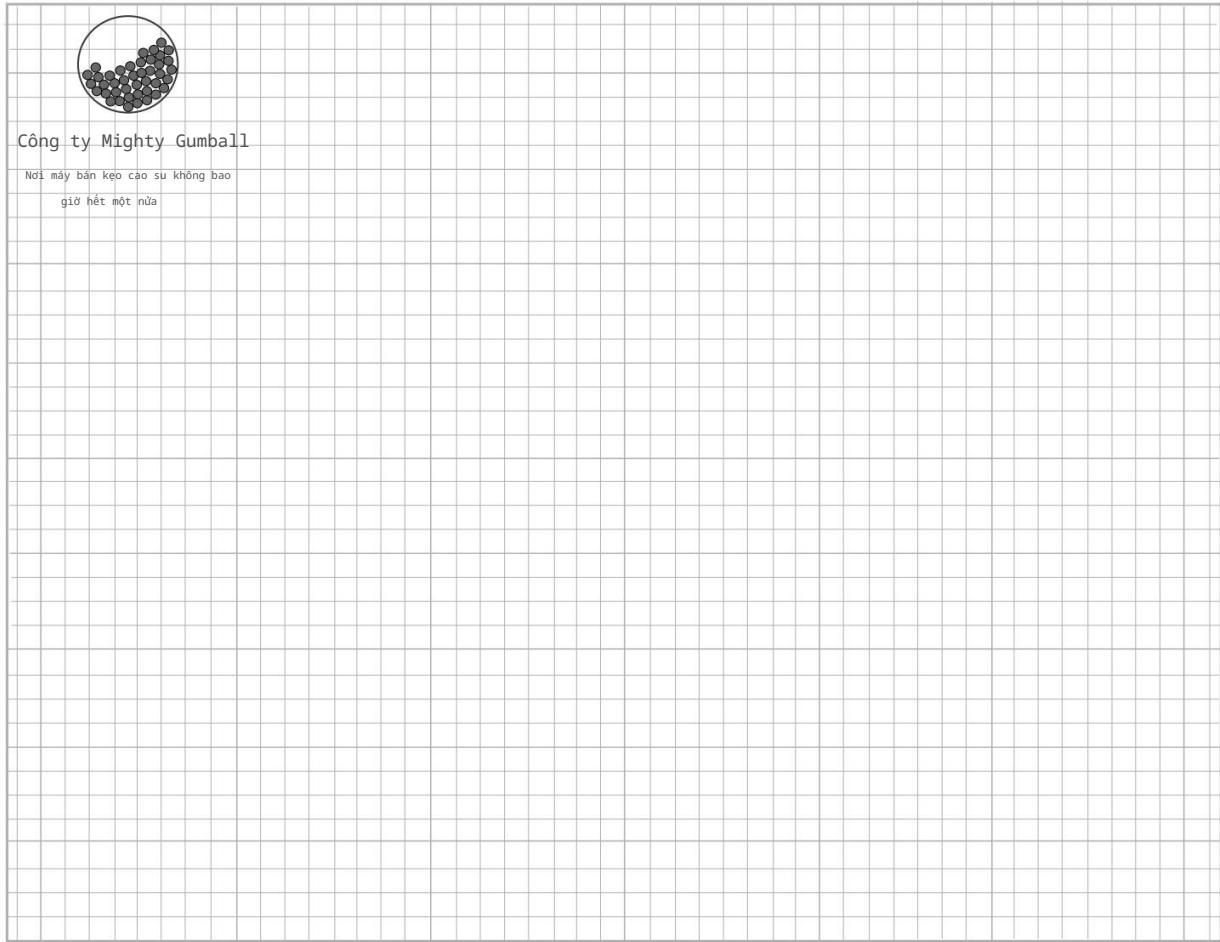
Trên thực tế, mọi việc diễn ra rất suôn sẻ đến mức họ muôn đưa nó lên một tầm cao mới...





Thiết kế câu đố

Vẽ sơ đồ trạng thái cho Máy Gumball xử lý cuộc thi 1 trong 10. Trong cuộc thi này, 10% thời gian trạng thái Đã bán dẫn đến việc hai quả bóng được thả ra, không phải một. Kiểm tra câu trả lời của bạn với câu trả lời của chúng tôi (ở cuối chương) để đảm bảo chúng ta đồng ý trước khi bạn đi xa hơn...



Sử dụng đồ dùng học tập của Mighty Gumball để vẽ sơ đồ trạng thái của bạn.

bạn đang ở đây 4 395

mọi thứ trở nên lộn xộn

Tình trạng hỗn loạn của mọi thứ...

Chỉ vì bạn đã viết máy gumball của mình bằng một phương pháp được cân nhắc kỹ lưỡng không có nghĩa là nó sẽ dễ dàng mở rộng. Trên thực tế, khi bạn quay lại và xem mã của mình và nghĩ về những gì bạn sẽ phải làm để sửa đổi nó, thì...

```
int tinh cuoi cung SOLD_OUT = 0;
int tinh cuoi cung NO_QUARTER = 1;
int tinh cuoi cung HAS_QUARTER = 2;
int tinh cuoi cung SOLD = 3;
```

```
công khai void insertQuarter() {
    // chèn mă quý ở đây
}

công khai void ejectQuarter() {
    // đẩy mă quý ra đây
}

công khai void turnCrank() {
    // quay mă tay quay ở đây
}

công khai void dispense() {
    // phân phối mă ở đây
}
```

Đầu tiên, bạn phải thêm một tiêu bang WINNER mới đây. Cũng không tệ lắm...

... nhưng sau đó, bạn sẽ phải thêm một điều kiện mới vào từng phương thức để xử lý trạng thái WINNER; đó là rất nhiều mã cần sửa đổi.

turnCrank() sẽ trả nền đặc biệt lộn xộn, vì bạn phải thêm mã để kiểm tra xem bạn có WINNER hay không rồi chuyển sang trạng thái WINNER hoặc trạng thái SOLD.

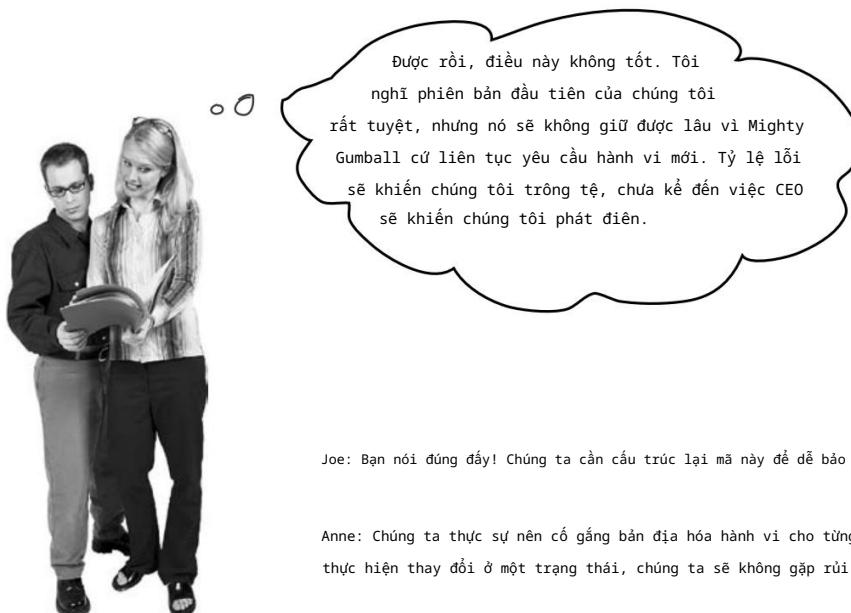


Chuốt bút chì của bạn

Câu nào sau đây mô tả trạng thái triển khai của chúng tôi?
(Chọn tất cả những câu trả lời đúng.)

A. Mã này chắc chắn không tuân thủ Nguyên tắc Mở Đóng.
B. Đoạn mã này sẽ khiến một lập trình viên FORTRAN phải tự hào.
C. Thiết kế này thậm chí không phải là đối tượng có định hướng.
Câu nào sau đây mô tả trạng thái triển khai của chúng tôi?
(Chọn tất cả những câu trả lời đúng.)

C. Các chuyển đổi trạng thái không rõ ràng; chúng bị chôn vùi ở giữa một loạt các câu lệnh có điều kiện.
D. Chúng tôi chưa đóng gói bất cứ thứ gì thay đổi ở đây.
E. Việc bổ sung thêm có thể gây ra lỗi trong mã đang hoạt động.



Joe: Bạn nói đúng đấy! Chúng ta cần cấu trúc lại mã này để dễ bảo trì và sửa đổi.

Anne: Chúng ta thực sự nên cố gắng bán địa hóa hành vi cho từng trạng thái để nếu chúng ta thực hiện thay đổi ở một trạng thái, chúng ta sẽ không gặp rủi ro làm hỏng mã khác.

Joe: Đúng vậy; nói cách khác, hãy tuân theo nguyên tắc "gói gọn những gì thay đổi".

Anne: Chính xác.

Joe: Nếu chúng ta đặt hành vi của mỗi trạng thái vào một lớp riêng, thì mỗi trạng thái sẽ chỉ thực hiện các hành động của riêng mình.

Anne: Đúng vậy. Và có lẽ Gumball Machine có thể chỉ cần chuyển giao cho đối tượng trạng thái biểu diễn trạng thái hiện tại.

Joe: À, bạn nói đúng: ứng hộ việc sáng tác... nhiều nguyên tắc hơn trong công việc.

Anne: Dễ thương. Ô, tôi không chắc chắn 100% là việc này sẽ diễn ra thế nào, nhưng tôi nghĩ chúng ta đang đi đúng hướng.

Joe: Tôi tự hỏi liệu điều này có giúp việc thêm trạng thái mới dễ dàng hơn không?

Anne: Tôi nghĩ vậy... Chúng ta vẫn phải thay đổi mã, nhưng phạm vi thay đổi sẽ hạn chế hơn nhiều vì việc thêm trạng thái mới có nghĩa là chúng ta chỉ phải thêm một lớp mới và có thể thay đổi một vài hiệu ứng chuyển tiếp ở đây và ở đó.

Joe: Tôi thích điều đó. Hãy bắt đầu thảo luận về thiết kế mới này nhé!

một thiết kế nhà nước mới

Thiết kế mới

Có vẻ như chúng ta có một kế hoạch mới: thay vì duy trì mã hiện tại, chúng ta sẽ làm lại nó để đóng gói các đối tượng trạng thái vào các lớp riêng của chúng rồi chuyển giao cho trạng thái hiện tại khi một hành động xảy ra.

Chúng tôi đang tuân theo các nguyên tắc thiết kế của mình ở đây, vì vậy chúng tôi sẽ có một thiết kế dễ bảo trì hơn trong tương lai. Đây là cách chúng tôi sẽ thực hiện:

- 1 Đầu tiên, chúng ta sẽ định nghĩa một giao diện State chứa phương thức cho mọi hành động trong Gumball Machine.
- 2 Sau đó, chúng ta sẽ triển khai một lớp State cho mọi trạng thái của máy. Các lớp này sẽ chịu trách nhiệm về hành vi của máy khi nó ở trạng thái tương ứng.
- 3 Cuối cùng, chúng ta sẽ loại bỏ toàn bộ mã có điều kiện và thay vào đó chuyển giao cho lớp trạng thái để lớp này thực hiện công việc thay chúng ta.

Chúng ta không chỉ tuân theo các nguyên tắc thiết kế, như bạn sẽ thấy, mà chúng ta thực sự đang triển khai State Pattern. Nhưng chúng ta sẽ tìm hiểu tất cả các nội dung chính thức của State Pattern sau khi chúng ta làm lại mã của mình...

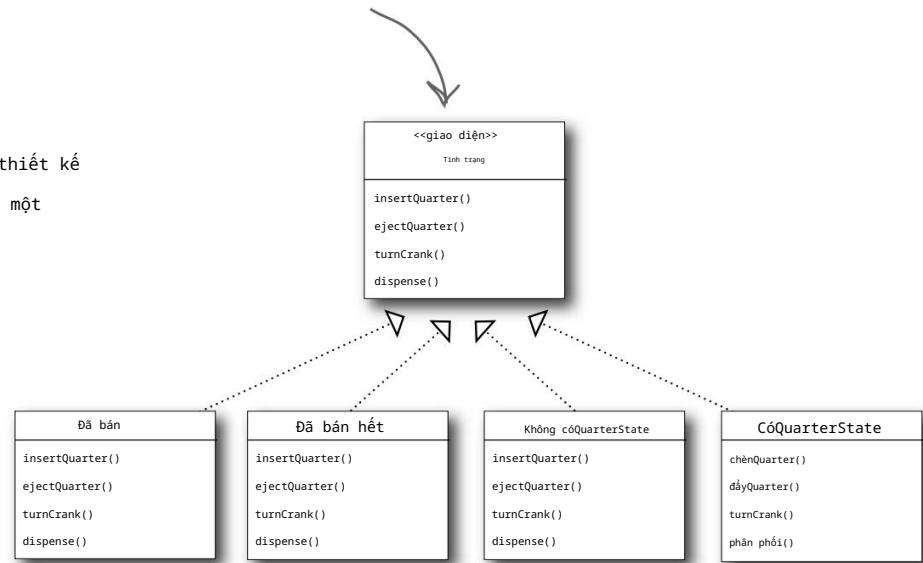


Xác định các giao diện và lớp trạng thái

Đầu tiên, hãy tạo một giao diện cho State, được tất cả các state của chúng ta triển khai:

Đây là giao diện cho tất cả các trạng thái. Các phương thức ánh xạ trực tiếp đến các hành động có thể xảy ra với Gumball Machine (đây là các phương thức giống như trong mã trước).

Sau đó lấy từng trạng thái trong thiết kế của chúng ta và đóng gói nó trong một lớp triển khai giao diện State.



Để tìm ra trạng thái chúng ta cần, chúng ta hãy xem mã trước đó...

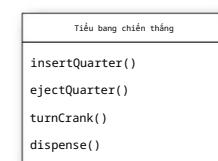
```

lớp công khai GumballMachine {
    int tinh cuối cùng SOLD_OUT = 0; int tinh cuối cùng NO_QUARTER = 1; int tinh cuối cùng HAS_QUARTER = 2; int tinh cuối cùng SOLD = 3;

    int state = ĐÃ BÁN HẾT; int count = 0;
}
  
```

... và chúng tôi ánh xạ mỗi trạng thái trực tiếp vào một lớp.

Đừng quên, chúng ta cũng cần một trạng thái "người chiến thắng" mới để triển khai giao diện trạng thái. Chúng ta sẽ quay lại vấn đề này sau khi triển khai lại phiên bản đầu tiên của Gumball Machine.



tất cả các tiêu bang là gì ?

Chuốt bút chì của bạn

Để triển khai các trạng thái của chúng ta, trước tiên chúng ta cần chỉ định hành vi của các lớp khi mỗi hành động được gọi. Chú thích sơ đồ bên dưới với hành vi của mỗi hành động trong mỗi lớp; chúng tôi đã điền một số cho bạn.

Đi đến HasQuarterState

Nói với khách hàng, "Bạn chưa bỏ đồng 25 xu vào."

Không cóQuarterState
chènQuarter()
dẩyQuarter()
quayCrank()
phân chia()

Đi đến SoldState

CóQuarterState
insertQuarter()
ejectQuarter()
turnCrank()
dispense()

Hãy nói với khách hàng, "Xin hãy đợi, chúng tôi đang tặng bạn một viên kẹo cao su."

Phân phối một viên kẹo cao su. Kiểm tra số lượng kẹo cao su; nếu > 0 , hãy chuyển đến NoQuarterState, nếu không, hãy chuyển đến SoldOutState

Đã bán
chènQuarter()
ejectQuarter()
quayCrank()
phân phối()

Nói với khách hàng: "Không có kẹo cao su nào cả".

Đã bán hết
chènQuarter()
ejectQuarter()
quayCrank()
phân phối()

Hãy điền vào mẫu này ngay cả khi chúng tôi sẽ triển khai sau.

Tiêu bang chiến thắng
insertQuarter()
ejectQuarter()
turnCrank()
dispense()

Triển khai các lớp State của chúng tôi

Đã đến lúc triển khai trạng thái: chúng ta biết những hành vi nào mình muốn; chúng ta chỉ cần đưa nó vào mã. Chúng ta sẽ theo sát mã máy trạng thái mà chúng ta đã viết, nhưng lần này mọi thứ được chia thành các lớp khác nhau.

Chúng ta hãy bắt đầu với NoQuarterState:

```
Đầu tiên chúng ta cần triển khai giao diện State.
↓
lớp công khai NoQuarterState thực hiện State {
    Máy GumballMáy Gumball;

    công khai NoQuarterState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

    công khai void insertQuarter() {
        System.out.println("Bạn đã chèn một đồng 25 xu");
        gumballMachine.setState(gumballMachine.getHasQuarterState());
    }

    công khai void ejectQuarter() {
        System.out.println("Bạn chưa chèn đồng 25 xu");
    }

    công khai void turnCrank() {
        System.out.println("Bạn đã quay lại, nhưng không có đồng xu nào cả");
    }

    công khai void dispense() {
        System.out.println("Bạn cần phải thanh toán trước");
    }
}
```

Chúng ta được truyền tham chiếu đến Gumball Machine thông qua constructor. Chúng ta sẽ chỉ lưu trữ tham chiếu này trong một biến thể hiện.

Nếu ai đó nhét một đồng 25 xu vào, chúng tôi sẽ in ra thông báo cho biết đồng 25 xu đã được chấp nhận và sau đó thay đổi trạng thái của máy thành HasQuarterState.

Bạn sẽ thấy chúng hoạt động như thế nào chỉ trong giây lát...

Bạn sẽ không thể lấy lại tiền nếu bạn chưa bao giờ đưa tiền cho chúng tôi!

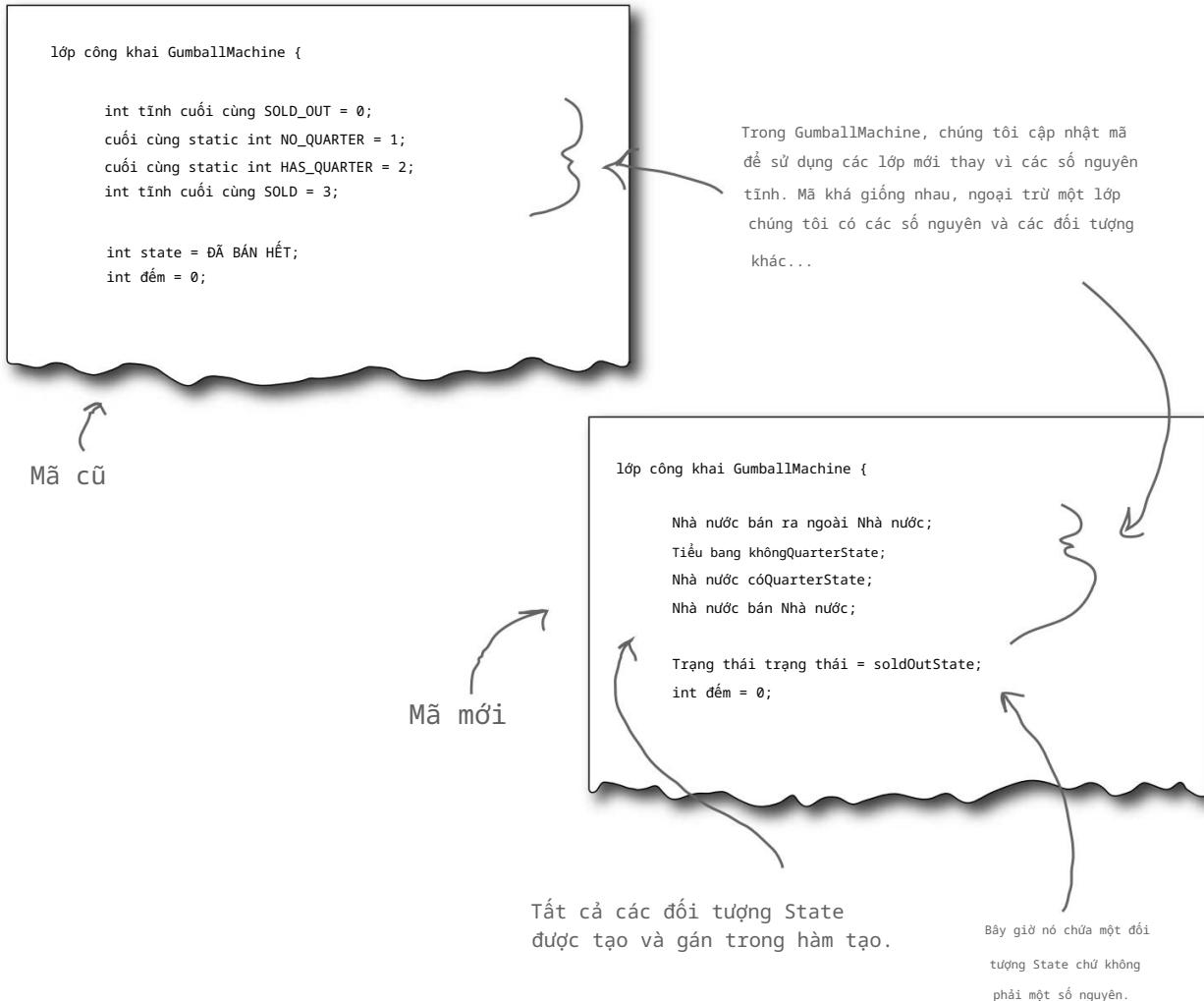
Và bạn sẽ không thể có được kẹo cao su nếu không trả tiền cho chúng tôi.
Chúng tôi không thể phát kẹo cao su nếu bạn không trả tiền.



các đối tượng trạng thái trong máy kẹo cao su

Làm lại máy Gumball

Trước khi hoàn thành các lớp State, chúng ta sẽ làm lại Gumball Machine - theo cách đó, bạn có thể thấy tất cả chúng khớp với nhau như thế nào. Chúng ta sẽ bắt đầu với các biến thể hiện liên quan đến state và chuyển mã từ sử dụng số nguyên sang sử dụng các đối tượng state:



Bây giờ, chúng ta hãy xem lớp GumballMachine hoàn chỉnh...

```
lớp công khai GumballMachine {
```

```
    Nhà nước bán ra ngoài Nhà nước;
    Tiêu bang khôngQuarterState;
    Nhà nước cóQuarterState;
    Nhà nước bán Nhà nước;
```

```
    Trạng thái trạng thái = soldOutState;
    int đếm = 0;
```

```
    công khai GumballMachine(int numberGumballs) {
        soldOutState = new SoldOutState(này);
        noQuarterState = new NoQuarterState(này);
        hasQuarterState = new HasQuarterState(này);
        soldState = new SoldState(này);
        this.count = sốKẹo cao su;
        nếu (số Gumballs > 0) {
            trạng thái = noQuarterState;
        }
    }
```

```
    công khai void insertQuarter() {
        state.insertQuarter();
    }
```

```
    công khai void ejectQuarter() {
        state.ejectQuarter();
    }
```

```
    công khai void turnCrank() {
        trạng thái.turnCrank();
        state.dispense();
    }
```

```
    void setState(Trạng thái trạng thái) {
        this.state = trạng thái;
    }
```

```
    void releaseBall() {
        System.out.println("Một viên kẹo cao su lăn ra khỏi khe...");
        nếu (đếm != 0) {
            đếm = đếm - 1;
        }
    }
```

```
// Thêm nhiều phương thức ở đây bao gồm các phương thức lấy dữ liệu cho từng trạng thái...
```

```
}
```

Đây lại là tất cả các tiêu bang một lần nữa...

...và biến thể hiện State.

Biến thể hiện count giữ
số lượng kẹo cao su - ban
đầu máy sẽ trống.

Hàm tạo của chúng tôi lấy số lượng kẹo
cao su ban đầu và lưu trữ nó trong một biến
thể hiện.

Nó cũng tạo ra các thể hiện trạng thái,
mỗi thể hiện một trạng thái.

Nếu có nhiều hơn 0 viên kẹo
cao su, chúng ta sẽ đặt trạng thái
thành NoQuarterState.

Bây giờ đến phần hành động. Những hành
động này RẤT ĐỄ thực hiện ngay bây giờ.

Chúng ta chỉ cần ủy quyền cho trạng thái hiện tại.

Lưu ý rằng chúng ta không cần
phương thức hành động cho
dispense() trong GumballMachine vì nó
chỉ là hành động nội bộ; người dùng
không thể yêu cầu máy phân phối trực
tiếp. Nhưng chúng ta gọi dispense() trên
đối tượng State từ phương thức turnCrank().

Phương pháp này cho phép các đối tượng
khác (như đối tượng State của chúng
ta) chuyển máy sang trạng thái khác.

Máy hỗ trợ phương thức trợ giúp
releaseBall() để giải phóng bóng và giảm
biến thể hiện count.

Điều này bao gồm các phương thức như getNoQuarterState() để lấy từng
đối tượng trạng thái và getCount() để lấy số lượng kẹo cao su.

nhiều tiền hơn cho máy bán kẹo cao su

Triển khai thêm nhiều trạng thái

Bây giờ bạn đã bắt đầu hiểu được cách Gumball Machine và các trạng thái kết hợp với nhau, hãy cùng triển khai các lớp HasQuarterState và SoldState...

```

lớp công khai HasQuarterState thực hiện State {
    Máy GumballMáy Gumball;

    công khai HasQuarterState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

    công khai void insertQuarter() {
        System.out.println("Bạn không thể chèn thêm đồng 25 xu nữa");
    }

    công khai void ejectQuarter() {
        System.out.println("Quý trả về");
        gumballMachine.setState(gumballMachine.getNoQuarterState());
    }

    công khai void turnCrank() {
        System.out.println("Bạn đã quay...");
        gumballMachine.setState(gumballMachine.getSoldState());
    }

    công khai void dispense() {
        System.out.println("Không phát kẹo cao su");
    }
}

```

Một hành động không phù hợp với tiền bang này.

Khi trạng thái được khởi tạo, chúng ta truyền cho nó một tham chiếu đến GumballMachine. Tham chiếu này được sử dụng để chuyển máy sang một trạng thái khác.

Một hành động không phù hợp với tiền bang này.

Trả lại đồng 25 xu của khách hàng và chuyển lại về NoQuarterState.

Khi quay tay quay, chúng ta chuyển máy sang trạng thái SoldState bằng cách gọi phương thức setState() và truyền cho nó đối tượng SoldState. Đối tượng SoldState được lấy bằng phương thức getSoldState() (có một trong những phương thức get này cho mỗi trạng thái).

mô hình trạng thái

Bây giờ, chúng ta hãy kiểm tra lớp SoldState...

```

lớp công khai SoldState thực hiện State {
    //constructor và các biến thể hiện ở đây

    công khai void insertQuarter() {
        System.out.println("Vui lòng đợi, chúng tôi đang tặng bạn một viên kẹo cao su");
    }

    công khai void ejectQuarter() {
        System.out.println("Xin lỗi, bạn đã quay tay quay rồi");
    }

    công khai void turnCrank() {
        System.out.println("Quay lại hai lần cũng không giúp bạn nhận được thêm viên kẹo cao su!");
    }

    công khai void dispense() {
        gumballMachine.releaseBall();
        nếu (gumballMachine.getCount() > 0) {
            gumballMachine.setState(gumballMachine.getNoQuarterState());
        } khác {
            System.out.println("Ồ, hết kẹo cao su rồi!");
            gumballMachine.setState(gumballMachine.getSoldOutState());
        }
    }
}

```

Và đây chính là nơi công việc thực sự bắt đầu...

Chúng ta đang ở trạng thái SoldState, nghĩa là khách hàng đã trả tiền. Vì vậy, trước tiên chúng ta cần yêu cầu máy nhả ra một viên kẹo cao su.

Sau đó, chúng ta sẽ hỏi máy về số lượng kẹo cao su và chuyển sang NoQuarterState hoặc SoldOutState.

Dưới đây là tất cả các hành động không phù hợp cho trạng thái này

não Apower

Hãy xem lại cách triển khai GumballMachine. Nếu tay quay được quay và không thành công (ví dụ khách hàng không nhét đồng 25 xu vào trước), chúng tôi vẫn gọi dispense, mặc dù điều đó không cần thiết.

Bạn có thể khắc phục điều này bằng cách nào?

đến lượt bạn thực hiện một trạng thái

Chuốt bút chì của bạn

Chúng ta vẫn còn một lớp chưa triển khai: SoldOutState.

Tại sao bạn không triển khai nó? Để làm được điều này, hãy suy nghĩ cẩn thận về cách Gumball Machine nên hoạt động trong từng tình huống. Kiểm tra câu trả lời của bạn trước khi tiếp tục...

```
lớp công khai SoldOutState thực hiện State
    Máy GumballMáy Gumball;

    public SoldOutState(GumballMachine gumballMachine) {

}

    công khai void insertQuarter() {

}

    công khai void ejectQuarter() {

}

    công khai void turnCrank() {

}

    công khai void dispense() {

}

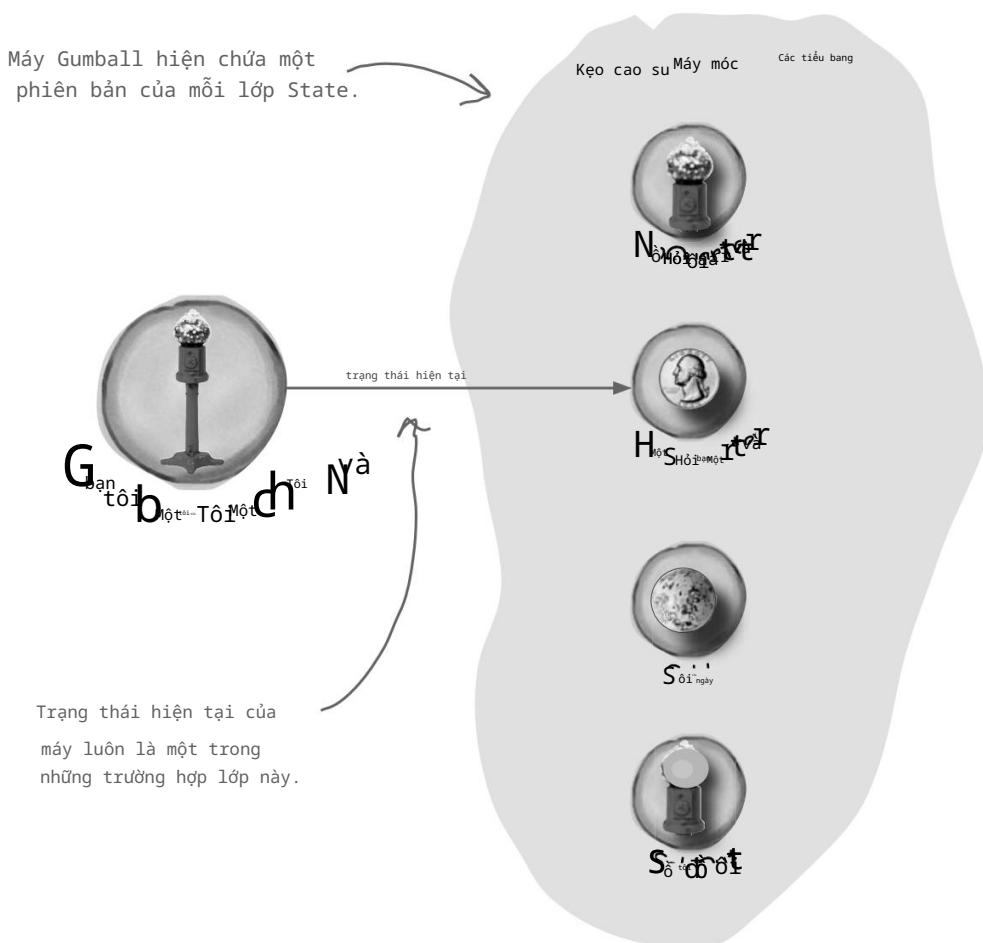
}
```

Hãy cùng xem những gì chúng tôi đã làm cho đến nay...

Để bắt đầu, bây giờ bạn có một bản triển khai Gumball Machine có cấu trúc khá khác so với phiên bản đầu tiên của bạn, nhưng về mặt chức năng thì hoàn toàn giống nhau. Bằng cách thay đổi cấu trúc triển khai, bạn đã:

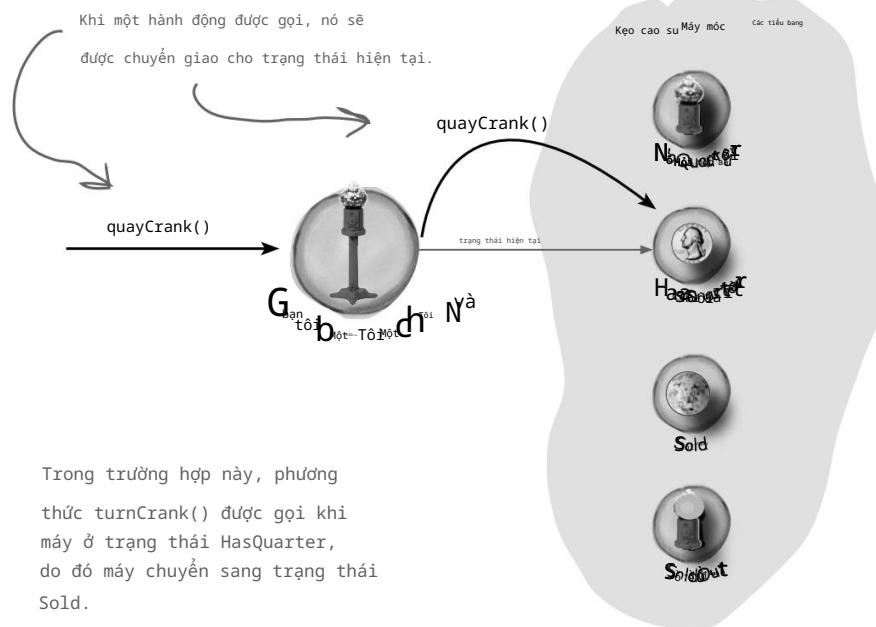
- ß Bản địa hóa hành vi của mỗi trạng thái vào lớp riêng của nó.
- ß Đã xóa tất cả các câu lệnh if gây khó khăn cho việc duy trì.
- ß Đã đóng từng tiểu bang để sửa đổi, nhưng vẫn để Gumball Machine mở để mở rộng thêm các lớp trạng thái mới (và chúng ta sẽ thực hiện việc này sau).
- ß Đã tạo cơ sở mã và cấu trúc lớp gần giống với sơ đồ Mighty Gumball hơn và dễ đọc và hiểu hơn.

Bây giờ chúng ta hãy xem xét kỹ hơn khía cạnh chức năng của những gì chúng tôi đã làm:

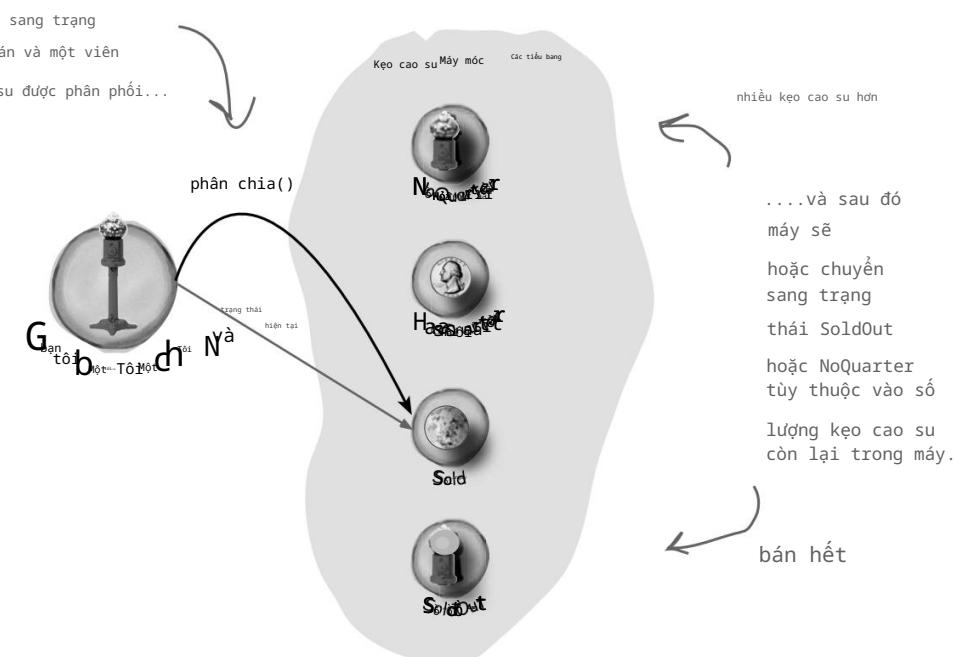


bạn đang ở đây 4 407

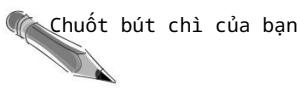
chuyển đổi trạng thái



CHUYỂN ĐỔI SANG TRẠNG THÁI ĐÃ BÁN



mô hình trạng thái



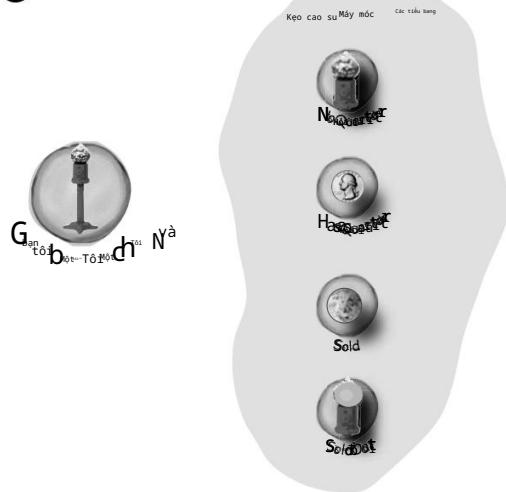
Hậu trường:

Tour tự hướng dẫn

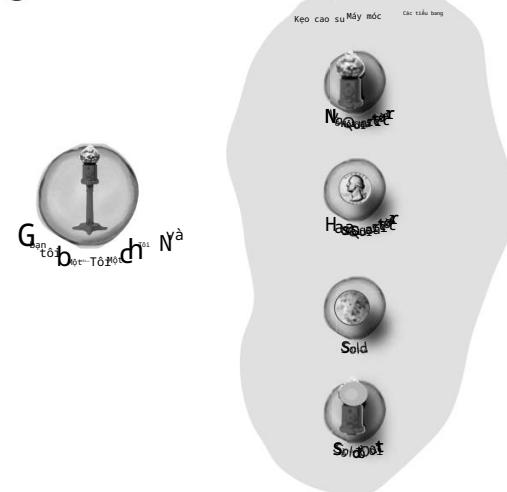


Theo dõi các bước của Gumball Machine bắt đầu từ trạng thái NoQuarter. Ngoài ra, hãy chú thích sơ đồ với các hành động và đầu ra của máy. Đối với bài tập này, bạn có thể cho rằng có rất nhiều gumball trong máy.

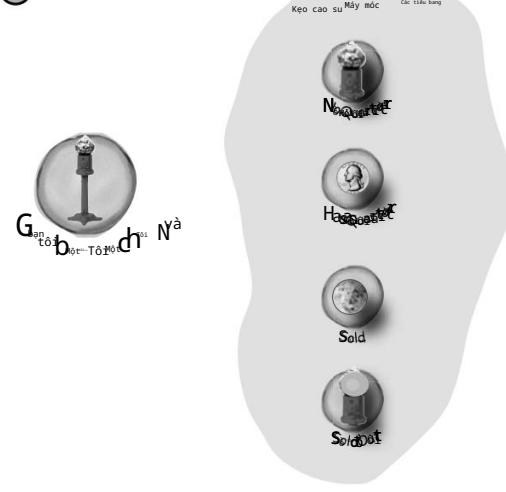
①



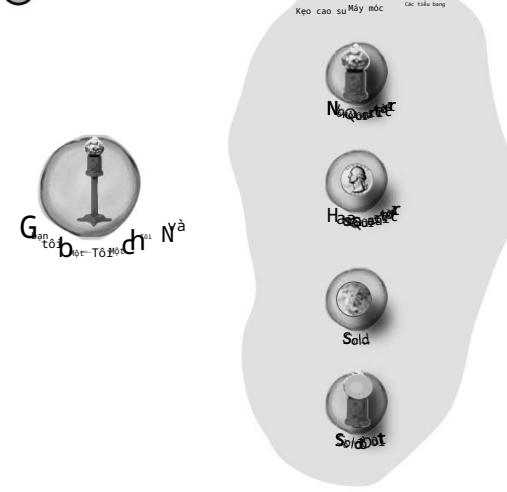
②



③



④



bạn đang ở đây 4 409

mẫu trạng thái được xác định

Mẫu trạng thái được xác định

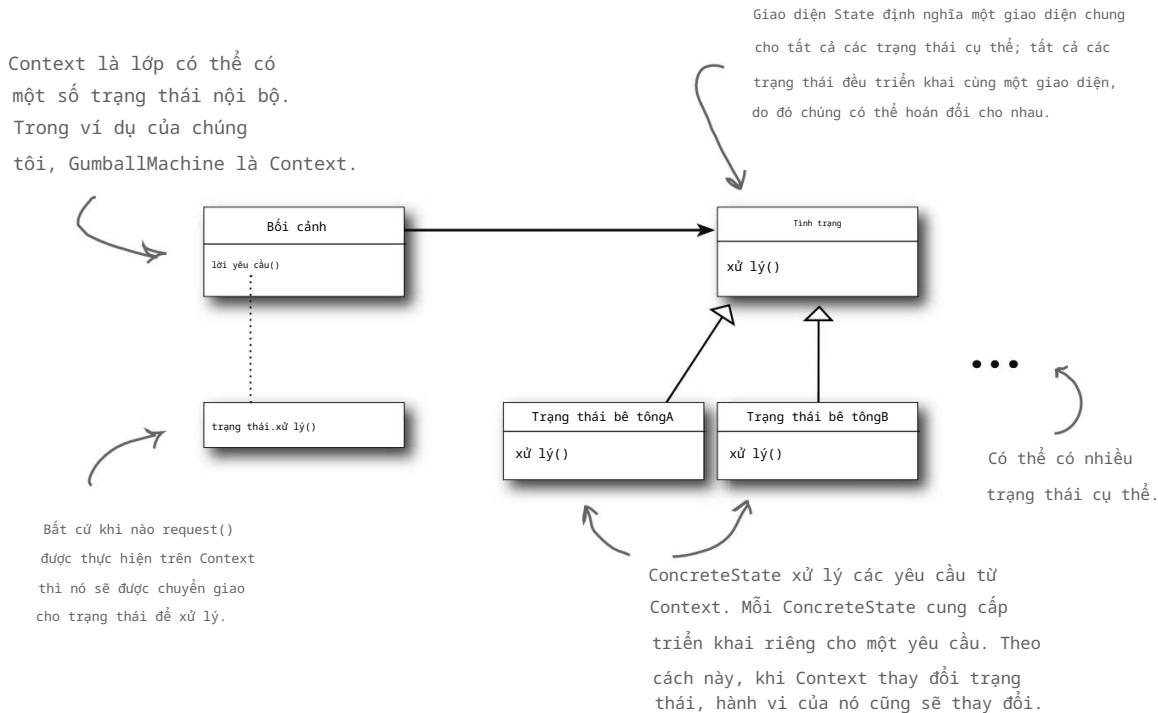
Vâng, đúng vậy, chúng tôi vừa triển khai State Pattern! Bây giờ, hãy cùng xem xét nó là gì:

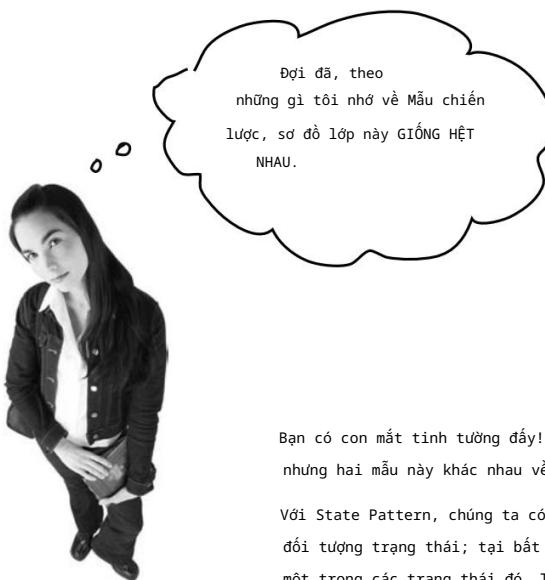
Mẫu trạng thái cho phép một đối tượng thay đổi hành vi khi trạng thái bên trong của nó thay đổi. Đối tượng sẽ xuất hiện để thay đổi lớp của nó.

Phần đầu tiên của mô tả này có nhiều ý nghĩa, đúng không? Bởi vì mẫu đóng gói trạng thái thành các lớp riêng biệt và chuyển giao cho đối tượng biểu diễn trạng thái hiện tại, chúng ta biết rằng hành vi thay đổi cùng với trạng thái nội bộ. Gumball Machine cung cấp một ví dụ hay: khi máy gumball ở NoQuarterState và bạn chèn một đồng 25 xu, bạn sẽ có hành vi khác (máy chấp nhận đồng 25 xu) so với khi bạn chèn một đồng 25 xu khi nó ở HasQuarterState (máy từ chối đồng 25 xu).

Còn phần thứ hai của định nghĩa thì sao? Một đối tượng “có vẻ như thay đổi lớp của nó” có nghĩa là gì? Hãy nghĩ về nó từ góc nhìn của một khách hàng: nếu một đối tượng bạn đang sử dụng có thể thay đổi hoàn toàn hành vi của nó, thì đối với bạn, có vẻ như đối tượng đó thực sự được khởi tạo từ một lớp khác. Tuy nhiên, trên thực tế, bạn biết rằng chúng ta đang sử dụng thành phần để tạo ra sự xuất hiện của một lớp thay đổi chỉ bằng cách tham chiếu đến các đối tượng trạng thái khác nhau.

Được rồi, bây giờ là lúc kiểm tra sơ đồ lớp Mẫu trạng thái:





Đợi đã, theo
những gì tôi nhớ về Mẫu chiến
lược, sơ đồ lớp này GIỐNG HỆT
NHAU.

Bạn có con mắt tinh tường đấy! Đúng vậy, sơ đồ lớp về cơ bản là giống nhau, nhưng hai mẫu này khác nhau về mục đích.

Với State Pattern, chúng ta có một tập hợp các hành vi được đóng gói trong các đối tượng trạng thái; tại bất kỳ thời điểm nào, ngữ cảnh sẽ chuyển giao cho một trong các trạng thái đó. Theo thời gian, trạng thái hiện tại thay đổi trên tập hợp các đối tượng trạng thái để phản ánh trạng thái nội bộ của ngữ cảnh, do đó, hành vi của ngữ cảnh cũng thay đổi theo thời gian. Khách hàng thường biết rất ít, nếu có, về các đối tượng trạng thái.

Với Strategy, khách hàng thường chỉ định đối tượng chiến lược mà ngữ cảnh được tạo thành. Bây giờ, trong khi mẫu cung cấp tính linh hoạt để thay đổi đối tượng chiến lược khi chạy, thường có một đối tượng chiến lược phù hợp nhất với đối tượng ngữ cảnh. Ví dụ, trong Chương 1, một số con vịt của chúng tôi được định cấu hình để bay với hành vi bay thông thường (như vịt trời), trong khi những con khác được định cấu hình với hành vi bay khiến chúng nằm trên mặt đất (như vịt cao su và vịt mồi).

Nhìn chung, hãy nghĩ về Mẫu chiến lược như một giải pháp thay thế linh hoạt cho việc phân lớp; nếu bạn sử dụng tính kế thừa để xác định hành vi của một lớp, thi bạn sẽ bị mắc kẹt với hành vi đó ngay cả khi bạn cần thay đổi nó.

Với Strategy, bạn có thể thay đổi hành vi bằng cách kết hợp với một đối tượng khác.

Hãy coi Mẫu trạng thái như một giải pháp thay thế cho việc đặt nhiều điều kiện vào ngữ cảnh của bạn; bằng cách đóng gói các hành vi trong các đối tượng trạng thái, bạn có thể dễ dàng thay đổi đối tượng trạng thái trong ngữ cảnh để thay đổi hành vi của nó.

hỏi đáp về mô hình nhà nước

không có Những câu hỏi ngắn

H: Trong GumballMachine, các tiêu bang quyết định trạng thái tiếp theo sẽ là gì. ConcreteStates có luôn quyết định trạng thái nào sẽ chuyển đến tiếp theo không?

A: Không, không phải lúc nào cũng vậy. Giải pháp thay thế là để Bối cảnh quyết định luồng chuyển đổi trạng thái.

Theo nguyên tắc chung, khi các chuyển đổi trạng thái được cố định, chúng sẽ phù hợp để đưa vào Context; tuy nhiên, khi các chuyển đổi động hơn, chúng thường được đặt trong chính các lớp trạng thái (ví dụ, trong GumballMachine, lựa chọn chuyển đổi sang NoQuarter hoặc SoldOut phụ thuộc vào số lượng gumball trong thời gian chạy).

Nhược điểm của việc có các chuyển đổi trạng thái trong các lớp trạng thái là chúng ta tạo ra sự phụ thuộc giữa các lớp trạng thái. Trong quá trình triển khai GumballMachine, chúng tôi đã cố gắng giảm thiểu điều này bằng cách sử dụng các phương thức getter trên Context, thay vì mã hóa cứng các lớp trạng thái cụ thể rõ ràng.

Lưu ý rằng khi đưa ra quyết định này, bạn đang đưa ra quyết định về việc lớp nào sẽ được đóng để sửa đổi - lớp Context hay lớp trạng thái - khi hệ thống phát triển.

Q: Khách hàng có bao giờ tương tác trực tiếp với tiêu bang?

A: Không. Các trạng thái được sử dụng bởi Bối cảnh để đại diện cho trạng thái và hành vi nội bộ của nó, vì vậy tất cả các yêu cầu đến các trạng thái đều đến từ Context. Client không trực tiếp thay đổi trạng thái của Context. Nhiệm vụ của Context là giám sát trạng thái của nó và bạn thường không muốn một client thay đổi trạng thái của Context mà không có sự hiểu biết của Context đó.

H: Nếu tôi có nhiều trường hợp của Context trong ứng dụng, có thể chia sẻ các đối tượng trạng thái giữa chúng không?

A: Vâng, chắc chắn rồi, và thực tế đây là một điều rất phổ biến kịch bản. Yêu cầu duy nhất là các đối tượng trạng thái của bạn không giữ trạng thái nội bộ của riêng chúng; nếu không, bạn sẽ cần một

trường hợp duy nhất cho mỗi ngữ cảnh.

Dễ chia sẻ trạng thái của bạn, bạn thường sẽ gán mỗi trạng thái cho một biến thể hiện tinh. Nếu trạng thái của bạn cần sử dụng các phương thức hoặc biến thể hiện trong Context của bạn, bạn cũng sẽ phải cung cấp cho nó một tham chiếu đến Context trong mỗi phương thức handler().

H: Có vẻ như luôn sử dụng Mẫu trạng thái tăng số lượng lớp trong thiết kế của chúng tôi. Hãy xem GumballMachine của chúng tôi có nhiều lớp hơn thiết kế ban đầu bao nhiêu!

A: Bạn nói đúng, bằng cách đóng gói hành vi trạng thái thành các lớp trạng thái riêng biệt, bạn sẽ luôn kết thúc với nhiều lớp hơn trong thiết kế của mình. Đó thường là cái giá bạn phải trả cho sự linh hoạt. Trừ khi mã của bạn là một số triển khai "một lần" mà bạn sẽ vứt bỏ (vâng, đúng vậy), hãy cân nhắc xây dựng nó với các lớp bổ sung và bạn có thể sẽ tự cảm ơn mình sau này. Lưu ý rằng thường thì điều quan trọng là số lượng lớp mà bạn đưa ra cho khách hàng của mình và có nhiều cách để ẩn các lớp bổ sung này khỏi khách hàng của bạn (ví dụ, bằng cách khai báo chúng là gói có thể nhìn thấy).

Ngoài ra, hãy cân nhắc phương án thay thế: nếu bạn có một ứng dụng có nhiều trạng thái và bạn quyết định không sử dụng các đối tượng riêng biệt, thay vào đó bạn sẽ kết thúc bằng các câu lệnh điều kiện nguyên khôi rất lớn. Điều này khiến mã của bạn khó bảo trì và hiểu. Bằng cách sử dụng các đối tượng, bạn làm cho các trạng thái trở nên rõ ràng và giảm bớt nỗ lực cần thiết để hiểu và bảo trì mã của mình.

Q: Biểu đồ lớp Mẫu trạng thái hiển thị State là một lớp trừu tượng. Nhưng bạn không sử dụng giao diện trong quá trình triển khai trạng thái của máy gumball sao?

A: Vâng. Vì chúng tôi không có chức năng chung để đưa vào một lớp trừu tượng, chúng tôi đã sử dụng một giao diện. Trong triển khai của riêng bạn, bạn có thể muốn xem xét một lớp trừu tượng. Làm như vậy có lợi ích là cho phép bạn thêm các phương thức vào lớp trừu tượng sau này mà không làm hỏng các triển khai trạng thái cụ thể.

Chúng ta vẫn cần phải hoàn thành trò chơi Gumball 1 trong 10

Hãy nhớ rằng, chúng ta vẫn chưa xong. Chúng ta còn một trò chơi để triển khai; nhưng giờ chúng ta đã triển khai State Pattern, mọi thứ sẽ dễ dàng. Đầu tiên, chúng ta cần thêm một trạng thái vào lớp GumballMachine:

```
lớp công khai GumballMachine {
```

```
    Nhà nước bán ra ngoài Nhà nước;
    Tiêu bang khôngQuarterState;
    Nhà nước cóQuarterState;
    Nhà nước bán Nhà nước;
    Người chiến thắng cấp tiêu bangTiêu bang;
```

Tất cả những gì bạn cần thêm vào đây là WinnerState mới và khởi tạo nó trong hàm tạo.

```
    Trạng thái trạng thái = soldOutState; int
    count = 0;
    // các phương pháp ở đây
}
```

Đừng quên bạn cũng phải thêm phương thức lấy dữ liệu cho WinnerState.

Bây giờ chúng ta hãy triển khai lớp WinnerState, nó rất giống với lớp SoldState:

```
lớp công khai WinnerState thực hiện State {
```

```
    // biến thể hiện và hàm tạo
    // thông báo lỗi insertQuarter
    // thông báo lỗi ejectQuarter
    // thông báo lỗi turnCrank

    public void dispense() { System.out.println("BẠN
        LÀ NGƯỜI CHIẾN THẮNG! Bạn nhận được hai viên kẹo cao su cho đồng xu của mình"); gumballMachine.releaseBall(); if (gumballMachine.getCount() ==
        0) { gumballMachine.setState(gumballMachine.getSoldOutState()); }
        else { gumballMachine.releaseBall(); if (gumballMachine.getCount() >
        0) {
```

Giống như SoldState.

Ở đây chúng tôi thả hai viên kẹo cao su và sau đó hoặc là đến NoQuarterState hoặc Đã bán hết.

```
            gumballMachine.setState(gumballMachine.getNoQuarterState()); } else { System.out.println("Rất
            tiếc, hết
            kẹo cao su rồi!"); gumballMachine.setState(gumballMachine.getSoldOutState());
        }
    }
}
```

Chỉ cần còn viên kẹo cao su thử

hai thì chúng tôi sẽ thả nó ra.

thực hiện trò chơi 1 trong 10

Kết thúc trò chơi

Chúng ta chỉ cần thực hiện thêm một thay đổi nữa: chúng ta cần triển khai trò chơi ngẫu nhiên và thêm một chuyển tiếp vào WinnerState. Chúng ta sẽ thêm cả hai vào HasQuarterState vì đó là nơi khách hàng quay tay quay:

```

lớp công khai HasQuarterState thực hiện State {
    Ngẫu nhiên randomWinner = new Random(System.currentTimeMillis());
    Máy GumballMáy Gumball;

    công khai HasQuarterState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

    công khai void insertQuarter() {
        System.out.println("Bạn không thể chèn thêm đồng 25 xu nữa");
    }

    công khai void ejectQuarter() {
        System.out.println("Quý trả về");
        gumballMachine.setState(gumballMachine.getNoQuarterState());
    }

    công khai void turnCrank() {
        System.out.println("Bạn đã quay...");
        int người chiến thắng = randomWinner.nextInt(10);
        nếu ((người chiến thắng == 0) && (gumballMachine.getCount() > 1)) {
            gumballMachine.setState(gumballMachine.getWinnerState());
        } khác {
            gumballMachine.setState(gumballMachine.getSoldState());
        }
    }

    công khai void dispense() {
        System.out.println("Không phát kẹo cao su");
    }
}

```

Đầu tiên chúng ta thêm một máy tạo số ngẫu nhiên để tạo ra 10% cơ hội chiến thắng...

...sau đó chúng ta xác định xem khách hàng này có thắng hay không.

Wow, thực hiện khá đơn giản! Chúng tôi chỉ cần thêm một trạng thái mới vào GumballMachine và sau đó thực hiện nó. Tất cả những gì chúng tôi phải làm từ đó là thực hiện trò chơi may rủi và chuyển sang trạng thái chính xác. Có vẻ như chiến lược mã mới của chúng tôi đang phát huy tác dụng...

Nếu họ thắng và còn đủ kẹo cao su để họ lấy hai viên, chúng ta sẽ đến WinnerState; nếu không, chúng ta sẽ đến SoldState (giống như chúng ta vẫn thường làm).

Bản demo dành cho CEO của Mighty Gumball, Inc.

CEO của Mighty Gumball đã ghé qua để demo mã trò chơi gumball mới của bạn. Hy vọng là các trạng thái đó đều ổn! Chúng tôi sẽ giữ bản demo ngắn gọn và hấp dẫn (khả năng tập trung ngắn của các CEO đã được ghi chép lại), nhưng hy vọng là đủ dài để chúng tôi có thể thắng ít nhất một lần.

```

Mã này thực ra không hề thay đổi; chúng
tôi chỉ rút ngắn nó một chút.

lớp công khai GumballMachineTestDrive {
    public static void main(String[] args) {
        GumballMachine gumballMachine = GumballMachine mới(5);

        System.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();

        System.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();

        System.out.println(gumballMachine);
    }
}

```

Một lần nữa, hãy bắt đầu với máy bán kẹo cao su có 5 viên kẹo cao su.

Chúng tôi muốn có một tiểu bang chiến thắng, vì vậy chúng tôi chỉ cần tiếp tục bơm vào các khu vực đó và quay tay quay. Thỉnh thoảng chúng tôi in ra trạng thái của máy bán kẹo cao su...

Toàn bộ nhóm kỹ sư đang đợi bên ngoài phòng hội nghị để xem liệu thiết kế dựa trên Mẫu trạng thái mới có hoạt động hay không!!



thử nghiệm máy bán kẹo cao su



Trời ơi, chúng ta may mắn quá phải không?
Trong bản trình bày với CEO, chúng tôi
đã thắng không chỉ một mà là hai lần!



```

Trợ giúp của sổ chính sửa tệp Khi nào thi isagumballajawbreaker?

%java GumballMachineKiểm tra ở đây
Công ty Mighty Gumball
Mô hình kẹo cao su đứng hỗ trợ Java #2004
Hàng tồn kho: 5 viên kẹo cao su
Máy đang chờ quý

Bạn đã chèn một phần tư
Bạn đã quay...
BẠN LÀ NGƯỜI CHIẾN THẮNG! Bạn nhận được hai viên kẹo cao su cho quý của bạn
Một viên kẹo cao su lăn ra khỏi khe cắm...
Một viên kẹo cao su lăn ra khỏi khe cắm...

Công ty Mighty Gumball
Mô hình kẹo cao su đứng hỗ trợ Java #2004
Hàng tồn kho: 3 viên kẹo cao su
Máy đang chờ quý

Bạn đã chèn một phần tư
Bạn đã quay...
Một viên kẹo cao su lăn ra khỏi khe cắm...
Bạn đã chèn một phần tư
Bạn đã quay...
BẠN LÀ NGƯỜI CHIẾN THẮNG! Bạn nhận được hai viên kẹo cao su cho quý của bạn
Một viên kẹo cao su lăn ra khỏi khe cắm...
Một viên kẹo cao su lăn ra khỏi khe cắm...
Ô, hết kẹo cao su rồi!

Công ty Mighty Gumball
Mô hình kẹo cao su đứng hỗ trợ Java #2004
Hàng tồn kho: 0 viên kẹo cao su
Máy đã bán hết
%
```

không có

Những câu hỏi ngớ ngẩn

H: Tại sao chúng ta cần WinnerState? Chúng ta không thể để SoldState phân phát hai viên kẹo cao su sao?

A: Đó là một câu hỏi tuyệt vời. SoldState và WinnerState gần như giống hệt nhau, ngoại trừ WinnerState phân phối hai gumballs thay vì một. Bạn chắc chắn có thể đưa mã để phân phối hai gumballs vào SoldState. Nhược điểm là, tất nhiên, bây giờ bạn có HAI trạng thái được biểu diễn trong một lớp State: trạng thái mà bạn là người chiến thắng và trạng thái mà bạn không phải là người chiến thắng. Vì vậy, bạn đang hy sinh tính rõ ràng trong lớp State của mình để giám sự trùng lặp của mã. Một điều khác cần cân nhắc là nguyên tắc mà bạn đã học ở chương trước: Một lớp, Một trách nhiệm. Bằng cách đưa trách nhiệm WinnerState vào SoldState, bạn vừa trao cho SoldState HAI trách nhiệm. Điều gì xảy ra khi chương trình khuyến mãi kết thúc? Hoặc tiền cược của cuộc thi thay đổi? Vì vậy, đó là một sự đánh đổi và phụ thuộc vào quyết định thiết kế.



Kiểm tra sự tinh táo...

Vâng, CEO của Mighty Gumball có lẽ cần kiểm tra lại sự tinh táo, nhưng đó không phải là điều chúng ta đang nói đến ở đây. Hãy cùng xem xét một số khía cạnh của GumballMachine mà chúng ta có thể muốn cung cấp trước khi phát hành phiên bản vàng:

β Chúng tôi có rất nhiều mã trùng lặp trong Sold and Winning states và chúng ta có thể muốn dọn dẹp chúng. Chúng ta sẽ làm như thế nào?
Chúng ta có thể biến State thành một lớp trừu tượng và xây dựng một số hành vi mặc định cho các phương thức; sau cùng, các thông báo lỗi như "Bạn đã chèn một phần tử" sẽ không được khách hàng nhìn thấy. Vì vậy, tất cả các hành vi "phản hồi lỗi" có thể là chung chung và được kế thừa từ lớp State trừu tượng.

Chết tiệt Jim,
tôi là máy bán
kẹo cao su chứ
không phải máy tính!

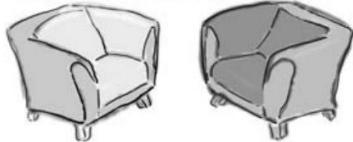
β Phương thức dispense() luôn được gọi, ngay cả khi tay quay được quay khi không có đồng xu nào. Trong khi máy hoạt động đúng và không phản phối trừ khi ở đúng trạng thái, chúng ta có thể dễ dàng sửa lỗi này bằng cách để turnCrank() trả về giá trị boolean hoặc bằng cách đưa ra các ngoại lệ. Bạn nghĩ giải pháp nào tốt hơn?

β Tất cả thông tin tình báo cho quá trình chuyển đổi trạng thái đều nằm trong Nhà nước lớp học. Điều này có thể gây ra những vấn đề gì? Chúng ta có muốn chuyển logic đó vào Gumball Machine không? Ưu và nhược điểm của việc đó là gì?

β Bạn có khởi tạo nhiều đối tượng GumballMachine không?
Nếu vậy, bạn có thể muốn di chuyển các thẻ hiện trạng thái vào các biến thẻ hiện tĩnh và chia sẻ chúng. Điều này sẽ yêu cầu những thay đổi gì đối với GumballMachine và các trạng thái?

trò chuyện bên lò sưởi: tình hình và chiến lược

Fireside Chats



Tôi nay: Cuộc hội ngộ về Chiến lược và Mô hình Nhà nước.

Chiến lược

Tình trạng

Này anh bạn. Anh có nghe là tôi đang ở Chương 1 không?

Vâng, tin tức này chắc chắn đang lan truyền.

Tôi vừa mới giúp đỡ những anh chàng của Template Method - họ cần tôi giúp họ hoàn thành chương của họ. Vậy, dù sao thì, anh trai quý tộc của tôi đang làm gì?

Giống như mọi khi - giúp các lớp thể hiện những hành vi khác nhau ở những trạng thái khác nhau.

Tôi không biết, bạn luôn có vẻ như bạn chỉ sao chép những gì tôi làm và bạn đang sử dụng những từ ngữ khác nhau để mô tả nó. Hãy nghĩ về điều đó: Tôi cho phép các đối tượng kết hợp các hành vi hoặc thuật toán khác nhau thông qua thành phần và sự ủy quyền.

Bạn chỉ đang sao chép tôi.

Tôi thừa nhận rằng những gì chúng ta làm chắc chắn có liên quan, nhưng mục đích của tôi hoàn toàn khác với bạn. Và cách tôi dạy khách hàng của mình sử dụng thành phần và ủy quyền cũng hoàn toàn khác.

Ô vây sao? Sao thế? Tôi không hiểu.

Vâng, nếu bạn dành nhiều thời gian hơn một chút để suy nghĩ về điều gì đó khác ngoài bản thân mình, bạn có thể. Dù sao, hãy nghĩ về cách bạn làm việc: bạn có một lớp mà bạn đang khởi tạo và bạn thường cung cấp cho nó một đối tượng chiến lược thực hiện một số hành vi.

Giống như, trong Chương 1, bạn đã đưa ra các hành vi quack, phải không? Vì thật có tiếng quack thật, vì cao su có tiếng quack kêu chích chích.

Vâng, đó thực sự là một công việc tuyệt vời ... và tôi chắc rằng bạn có thể thấy rằng điều đó mạnh mẽ hơn nhiều so với việc thưa hướng hành vi của mình, phải không?

Vâng, tất nhiên rồi. Bây giờ, hãy nghĩ về cách tôi làm việc; nó hoàn toàn khác biệt.

Xin lỗi, bạn sẽ phải giải thích điều đó.

Chiến lược

Tình trạng

Được rồi, khi các đối tượng Context của tôi được tạo, tôi có thể cho chúng biết trạng thái bắt đầu, nhưng sau đó chúng sẽ thay đổi trạng thái của chính mình theo thời gian.

Này, thôi nào, tôi cũng có thể thay đổi hành vi khi chạy; đó chính là mục đích của thành phần!

Chắc chắn là có thể, nhưng cách tôi làm việc được xây dựng xung quanh các trạng thái rời rạc; các đối tượng Context của tôi thay đổi trạng thái theo thời gian theo một số chuyển đổi trạng thái được xác định rõ ràng. Nói cách khác, việc thay đổi hành vi được tích hợp vào cơ sở của tôi - đó là cách tôi làm việc!

Vâng, tôi thừa nhận, tôi không khuyến khích các đối tượng của mình có một tập hợp các chuyển đổi được xác định rõ ràng giữa các trạng thái.

Trên thực tế, tôi thường thích kiểm soát chiến lược mà các đối tượng của mình đang sử dụng.

Này, chúng ta đã nói rằng chúng ta giống nhau về mặt cấu trúc, nhưng những gì chúng ta làm thì khá khác nhau về mục đích. Hãy đổi mặt với sự thật, thế giới có công dụng cho cả hai chúng ta.

Ừ, ừ, cứ sống với giấc mơ viễn vông của anh đi anh bạn. Bạn hành động như thế bạn là một người theo khuôn mẫu lớn giống tôi, nhưng hãy xem thử: Tôi đang ở Chương 1; họ kéo bạn xa xa ở Chương 10. Ý tôi là, có bao nhiêu người thực sự sẽ đọc đến tận đây?

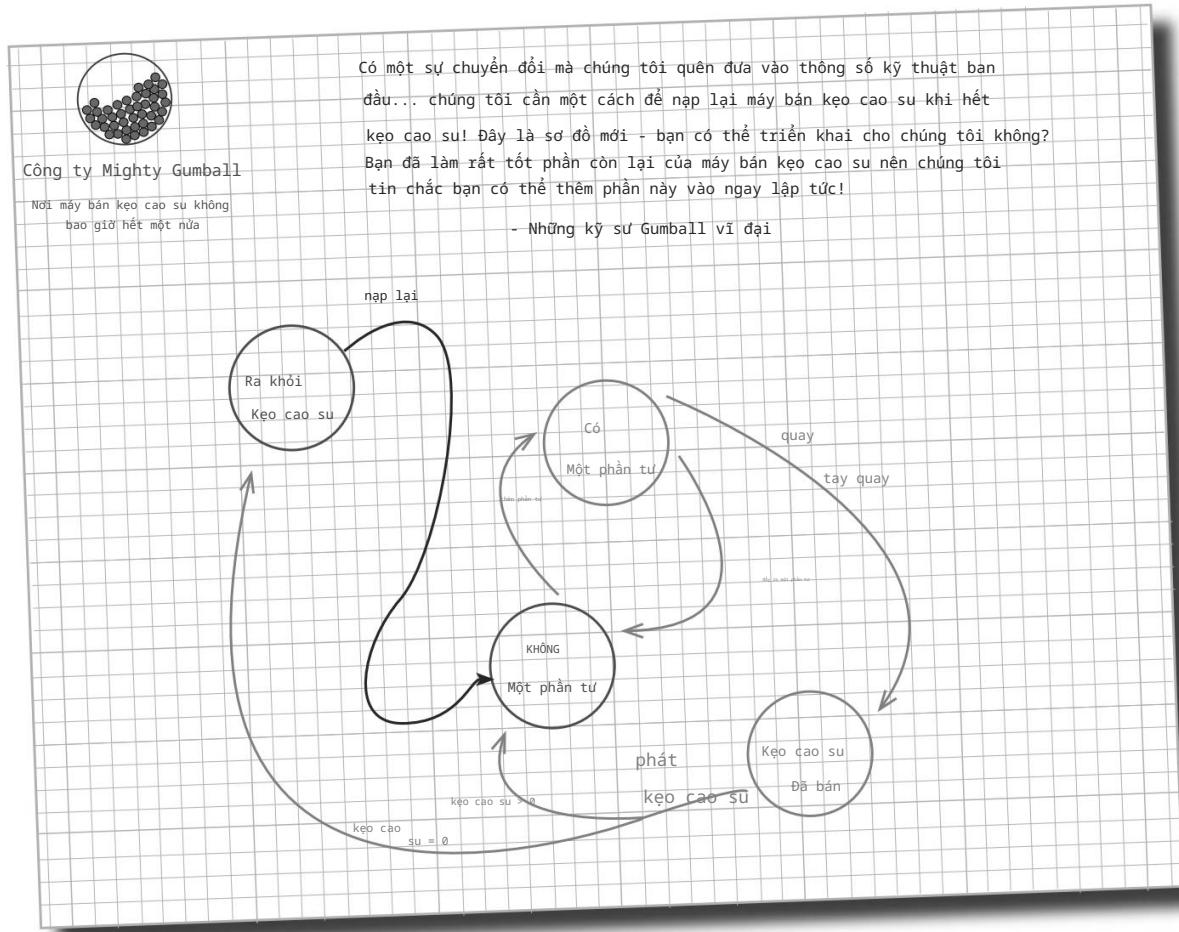
Bạn đùa à? Đây là một cuốn sách Head First và độc giả của Head First rất thích. Tất nhiên là họ sẽ đọc đến Chương 10!

Đó là anh trai tôi, người luôn mơ mộng.

bạn đang ở đây 4 419

nạp lại bài tập

Chúng tôi gần như quên mất!



 Chuốt bút chì của bạn

Chúng tôi cần bạn viết phương thức `refill()` cho máy Gumball. Phương thức này có một đối số số lượng gumball bạn đang thêm vào máy và sẽ cập nhật số lượng máy gumball và thiết lập lại trạng thái của máy.

Bạn đã làm một công việc tuyệt vời!
Tôi có thêm một số ý tưởng sẽ
thay đổi ngành công nghiệp kẹo cao
su và tôi cần bạn triển khai chúng.
Suyt! Tôi sẽ cho bạn biết những ý tưởng
này trong chương tiếp theo.



Ai làm gì?



Ghép mỗi mẫu với mô tả của nó:

Mẫu

Sự miêu tả

Tình trạng

Đóng gói các hành vi có thể hoán

đổi cho nhau và sử dụng sự phân quyền
để quyết định hành vi nào sẽ sử dụng

Chiến lược

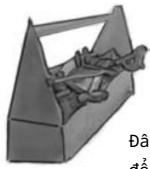
Các lớp con quyết định

cách thực hiện các bước trong
một thuật toán

Phương pháp mẫu

Đóng gói hành vi dựa trên
trạng thái và chuyển
giao hành vi cho trạng thái hiện tại

mô hình trạng thái



Công cụ cho hộp công cụ thiết kế của bạn

Đây là phần kết của một chương nữa; bạn đã có đủ các mẫu câu để vượt qua bất kỳ cuộc phỏng vấn xin việc nào!

Nguyên tắc 00

Bao gồm những gì thay đổi.

Ưu tiên thành phần hơn là thừa kế.

Chương trình giao diện, không việc triển khai.

Có gắng thiết kế các đối tượng tương tác một cách lồng lèo.

Các lớp học nên mở để mở rộng nhưng đóng để sửa đổi.

Phụ thuộc vào sự trừu tượng. Không phụ thuộc vào các lớp cụ thể.

Chỉ nói chuyện với bạn bè.

Đừng gọi cho chúng tôi, chúng tôi sẽ gọi cho bạn.

Một lớp học chỉ nên có một lý do để thay đổi.

Không có nguyên tắc mới nào

trong chương này, điều đó cho bạn thời gian để suy nghĩ về chúng.

Mẫu 00

Sau khi hoàn thành, bạn có thể bắt đầu với một số dòng lệnh để thêm thêm các lớp và các thành phần khác nhau. Điều này có thể giúp bạn dễ dàng hơn trong việc tạo ra các ứng dụng phức tạp.

Để bắt đầu, hãy thêm các lớp và thành phần sau:

- Lớp `Calculator` để thực hiện các phép tính cơ bản.
- Lớp `Storage` để lưu trữ dữ liệu.
- Lớp `Network` để kết nối với Internet.
- Lớp `UI` để tạo giao diện người dùng.
- Lớp `Database` để quản lý cơ sở dữ liệu.

Đến đây, bạn có thể bắt đầu kết hợp các lớp và thành phần này để tạo ra một ứng dụng đầy đủ chức năng.

ĐIỂM ĐẦU TIÊN



β Mẫu trạng thái cho phép mở

Đối tượng có nhiều hành vi khác nhau dựa trên trạng thái bên trong của nó.

β Không giống như máy trạng thái thủ tục, Mẫu trạng thái biểu diễn trạng thái như một lớp hoàn chỉnh.

β Context có được hành vi của nó bằng cách ủy quyền cho đối tượng trạng thái hiện tại mà nó được tạo thành.

β Bằng cách đóng gói từng trạng thái vào một lớp, chúng ta có thể bảnh địa hóa mọi thay đổi cần thực hiện.

β Các mẫu trạng thái và chiến lược có cùng sơ đồ lớp, nhưng chúng khác nhau về mục đích.

β Mẫu chiến lược thường cấu hình các lớp ngũ cung bằng hành vi hoặc thuật toán.

β Mẫu trạng thái cho phép một ngũ cung thay đổi hành vi khi trạng thái của ngũ cung thay đổi.

β Chuyển đổi trạng thái có thể được kiểm soát bởi các lớp Trạng thái hoặc các lớp Ngũ cung.

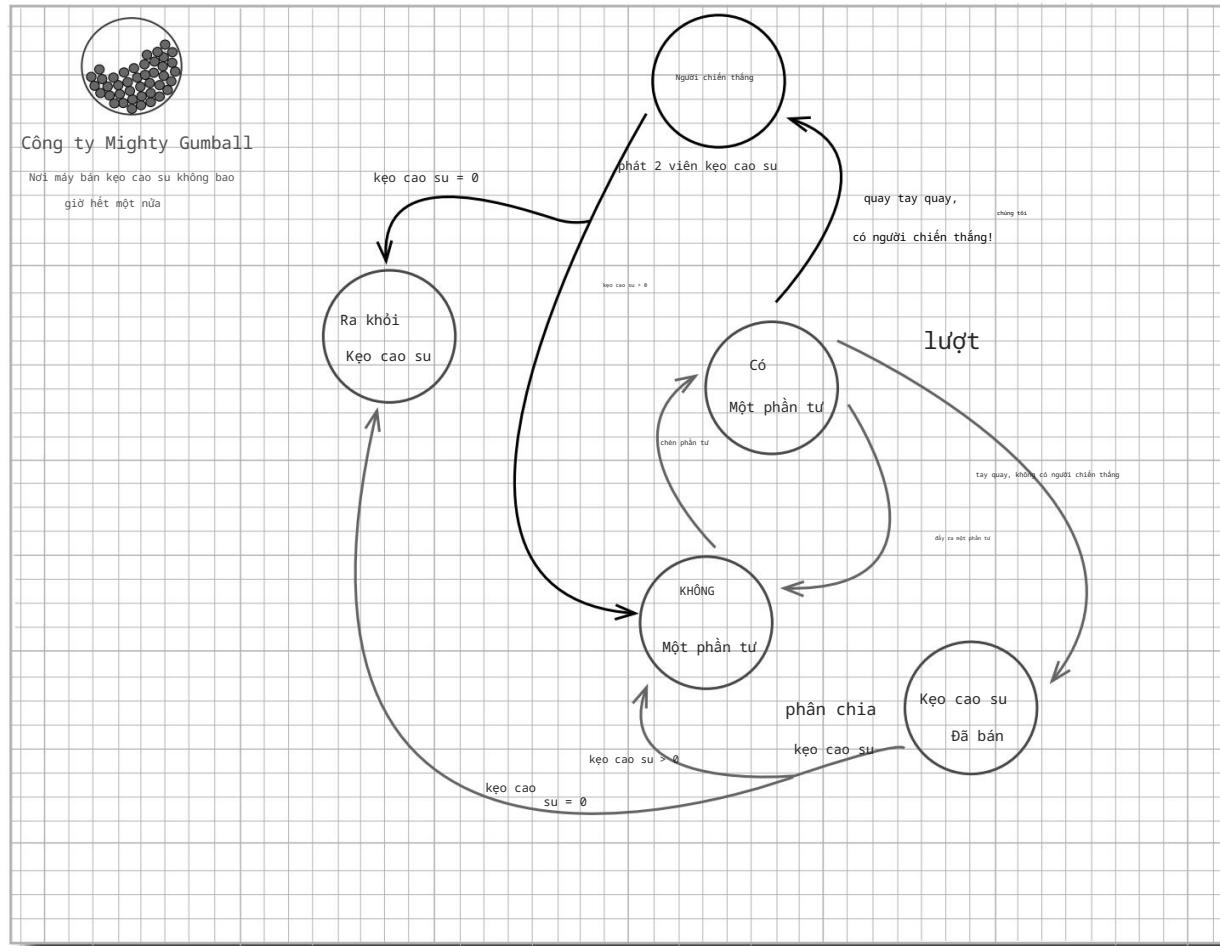
β Sử dụng Mẫu trạng thái thường sẽ tạo ra số lượng lớp lớn hơn trong thiết kế của bạn.

β Các lớp trạng thái có thể được chia sẻ giữa các thẻ hiện Context.

giải bài tập



Giải pháp bài tập





Giải pháp bài tập

Chuốt bút chì của bạn

Dựa trên lần triển khai đầu tiên, điều nào sau đây đúng?

(Chọn tất cả những câu trả lời đúng.)

- A. Mã này chắc chắn không tuân thủ Nguyên tắc Mở-Đóng!
- B. Đoạn mã này sẽ khiến một lập trình viên FORTRAN phải tự hào.
- C. Thiết kế này thậm chí không phải là đối tượng có định hướng.
- D. Các chuyển đổi trạng thái không rõ ràng; chúng bị chôn vùi giữa một loạt mã có điều kiện.
- E. Việc bổ sung thêm có thể gây ra lỗi trong mã đang hoạt động.

Chuốt bút chì của bạn

Chúng ta vẫn còn một lớp chưa triển khai: SoldOutState.

Tại sao bạn không triển khai nó? Để làm được điều này, hãy suy nghĩ cẩn thận về cách Gumball Machine nên hoạt động trong từng tình huống. Kiểm tra câu trả lời của bạn trước khi tiếp tục...

Ở trạng thái Hết hàng, chúng tôi thực sự không thể làm gì cho đến khi có người nạp đầy Gumball Machine.

```

lớp công khai SoldOutState thực hiện State {
    Máy GumballMáy Gumball;

    public SoldOutState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

    công khai void insertQuarter() {
        System.out.println("Bạn không thể nhét đồng 25 xu vào vì máy đã bán hết");
    }

    công khai void ejectQuarter() {
        System.out.println("Bạn không thể đẩy ra vì bạn chưa cho đồng 25 xu vào");
    }

    công khai void turnCrank() {
        System.out.println("Bạn đã quay lại, nhưng không có kẹo cao su nào cả");
    }

    công khai void dispense() {
        System.out.println("Không phát kẹo cao su");
    }
}

```

giải bài tập



Chuốt bút chì của bạn

Để triển khai các trạng thái, trước tiên chúng ta cần xác định hành vi sẽ như thế nào khi hành động tương ứng được gọi. Chú thích sơ đồ bên dưới với hành vi của từng hành động trong mỗi lớp; chúng tôi đã điền một số cho bạn.

Đi đến HasQuarterState

Nói với khách hàng rằng "bạn chưa bỏ một đồng 25 xu"



Không cóQuarterState
chènQuarter()
đẩyQuarter()
turnCrank()
phân phối()

Nói với khách hàng rằng "bạn đã quay lại nhưng không có đồng xu nào cả"



Nói với khách hàng "bạn cần phải trả tiền trước"



Nói với khách hàng rằng "bạn không thể nhét thêm phần tư nữa"



Trả lại một phần tư, di đến tiểu bang Không có phần tư



Đi đến SoldState



Nói với khách hàng, "không có kẹo cao su nào được phân phát"

CóQuarterState
insertQuarter()
ejectQuarter()
turnCrank()
dispense()

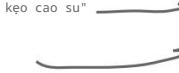
Nói với khách hàng "vui lòng đợi, chúng tôi đang tặng bạn một viên kẹo cao su"



Nói với khách hàng rằng "xin lỗi, bạn đã quay tay quay rồi"



Nói với khách hàng rằng "quay lại hai lần cũng không giúp bạn có thêm viên kẹo cao su"



Phân phối một viên kẹo cao su. Kiểm tra số lượng kẹo cao su; nếu > 0 , chuyển đến trạng thái NoQuarter, nếu không, chuyển đến trạng thái Sold Out

Đã bán
chènQuarter()
đẩyQuarter()
quayCrank()
phân chia()

Nói với khách hàng rằng "máy đã bán hết"



Nói với khách hàng rằng "bạn vẫn chưa bỏ một đồng 25 xu vào"



Nói với khách hàng rằng "Không có kẹo cao su"



Nói với khách hàng rằng "không có kẹo cao su nào được phân phát"



Nói với khách hàng "vui lòng đợi, chúng tôi đang tặng bạn một viên kẹo cao su"



Nói với khách hàng rằng "xin lỗi, bạn đã quay tay quay rồi"



Nói với khách hàng rằng "quay lại hai lần cũng không giúp bạn có thêm viên kẹo cao su"



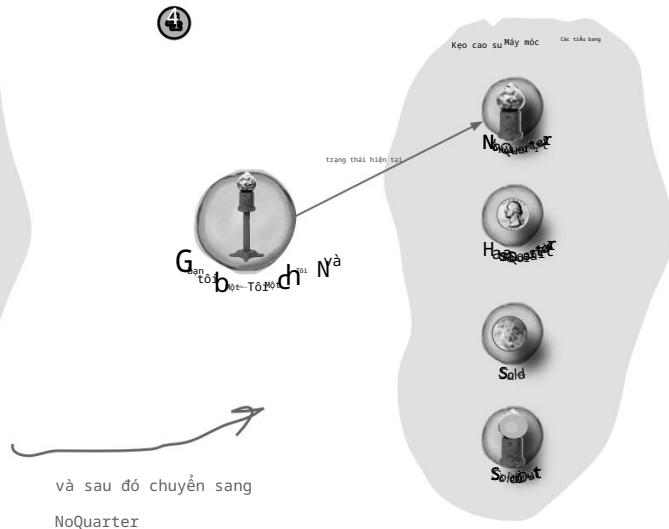
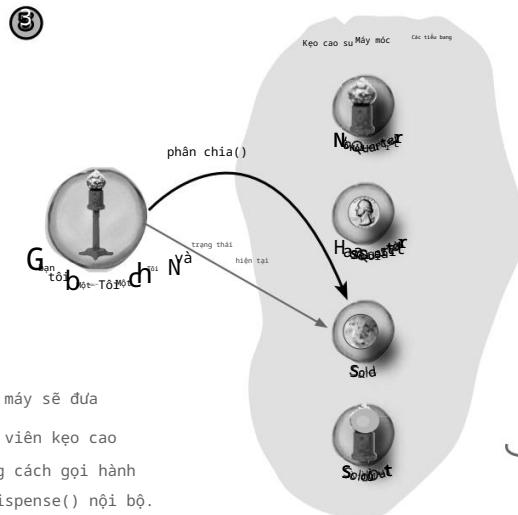
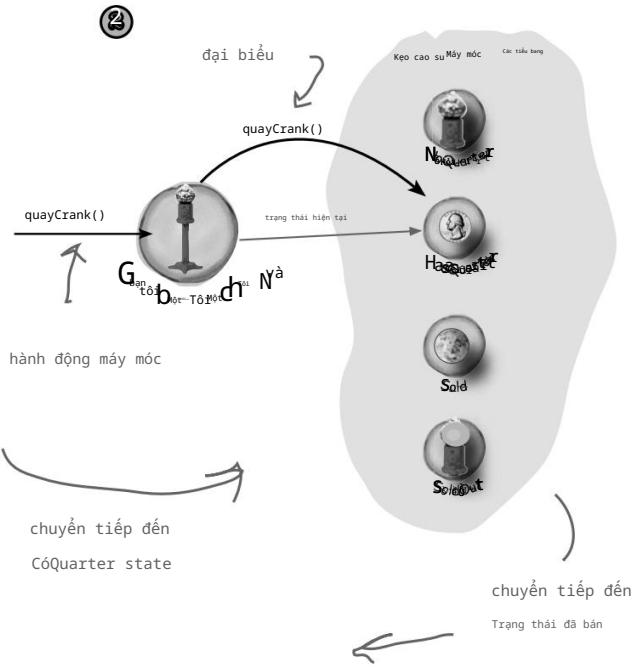
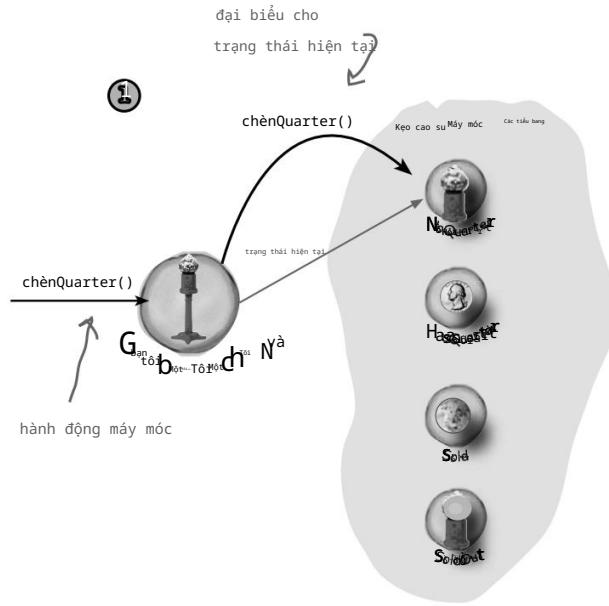
Phân phối hai viên kẹo cao su. Kiểm tra số lượng kẹo cao su; nếu > 0 ,

hãy chuyển đến NoQuarter state, nếu không, hãy chuyển đến SoldOutState

Tiểu bang chiến thắng
chènQuarter()
ejectQuarter()
quayCrank()
phân phối()

mô hình trạng thái

Hậu trường:
Tour tự hướng dẫn
Giải pháp



bạn đang ở đây 4 427

giải bài tập



Chuốt bút chì của bạn

Chúng tôi cần bạn viết phương thức refill() cho máy Gumball. Phương thức này có một đối số, số lượng gumball bạn đang thêm vào máy và sẽ cập nhật số lượng gumball của máy và thiết lập lại trạng thái của máy.

```
void nạp lại(int count) {
    this.count = đếm;
    trạng thái = noQuarterState;
}
```

11 Mẫu Proxy

Kiểm soát h g Sự vật Truy cập g

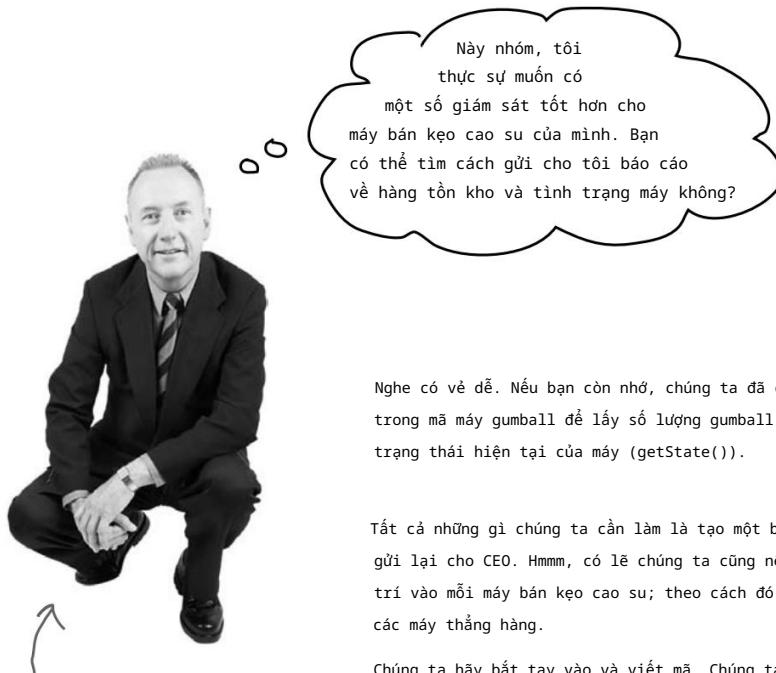


Bạn đã bao giờ đóng vai cảnh sát tốt, cảnh sát xấu chưa? Bạn là cảnh sát tốt và bạn cung cấp mọi dịch vụ của mình theo cách tử tế và thân thiện, nhưng bạn không muốn mọi người yêu cầu bạn cung cấp dịch vụ, vì vậy bạn để cảnh sát xấu kiểm soát quyền truy cập vào bạn. Đó là những gì proxy làm: kiểm soát và quản lý quyền truy cập. Như bạn sẽ thấy, có rất nhiều cách mà proxy thay thế cho các đối tượng mà chúng ủy quyền. Proxy được biết đến là có thể vận chuyển toàn bộ các cuộc gọi phương thức qua Internet cho các đối tượng được ủy quyền của chúng; chúng cũng được biết đến là kiên nhẫn thay thế một số những đồ vật lười biếng.

đây là một chương mới

429

mục tiêu là gì



Này nhóm, tôi
thực sự muốn có
một số giám sát tốt hơn cho
máy bán kẹo cao su của mình. Bạn
có thể tìm cách gửi cho tôi báo cáo
về hàng tồn kho và tình trạng máy không?

Nghé có vẻ dễ. Nếu bạn còn nhớ, chúng ta đã có các phương thức
trong mã máy gumball để lấy số lượng gumball (`getCount()`) và lấy
trạng thái hiện tại của máy (`getState()`).

Tất cả những gì chúng ta cần làm là tạo một báo cáo có thẻ in ra và
gửi lại cho CEO. Hmm, có lẽ chúng ta cũng nên thêm một trường vị
trí vào mỗi máy bán kẹo cao su; theo cách đó, CEO có thể giữ cho
các máy thẳng hàng.

Chúng ta hãy bắt tay vào và viết mã. Chúng ta sẽ gây ấn tượng
với CEO bằng tốc độ xử lý cực nhanh.

Bạn còn nhớ CEO của
Mighty Gumball, Inc. không?

Mã hóa màn hình

Chúng ta hãy bắt đầu bằng cách thêm hỗ trợ cho lớp GumballMachine để nó có thể xử lý vị trí:

```
lớp công khai GumballMachine {
    // các biến thể hiện khác
    Vị trí chuỗi:

    public GumballMachine(Vị trí, số nguyên đếm) {
        // mã xây dựng khác ở đây
        this.location = vị trí;
    }

    công khai String getLocation() {
        vị trí trả về;
    }

    // các phương pháp khác ở đây
}
```



Vị trí chỉ là một chuỗi.



Vị trí được truyền vào hàm tạo, và lưu trữ trong biến thể hiện.



Chúng ta cũng hãy thêm một phương thức getter để lấy vị trí khi cần.

Bây giờ chúng ta hãy tạo một lớp khác, GumballMonitor, lớp này sẽ lấy vị trí của máy, danh mục cao su và trạng thái hiện tại của máy rồi in chúng thành một báo cáo nhỏ đẹp mắt:

```
lớp công khai GumballMonitor {
    Máy GumballMachine;

    public GumballMonitor(máy GumballMachine) {
        this.machine = máy mót;
    }

    công khai void báo cáo() {
        System.out.println("Máy Gumball: " + machine.getLocation());
        System.out.println("Số lượng hàng tồn kho hiện tại: " + machine.getCount() + " gumballs");
        System.out.println("Trạng thái hiện tại: " + máy.getState());
    }
}
```



Trình giám sát lấy máy trong hàm khởi tạo của nó và gán nó cho biến thể hiện máy.



Phương pháp báo cáo của chúng tôi chỉ in báo cáo về vị trí, hàng tồn kho và trạng thái của máy.

giám sát gumball địa phương

Kiểm tra màn hình

Chúng tôi đã triển khai điều đó ngay lập tức. Tổng giám đốc điều hành sẽ rất vui mừng và kinh ngạc trước kỹ năng phát triển của chúng tôi.

Bây giờ chúng ta chỉ cần khởi tạo GumballMonitor và cấp cho nó một máy để giám sát:

```

lớp công khai GumballMachineTestDrive {
    public static void main(String[] args) {
        int đếm = 0;

        nếu (args.length < 2) {
            System.out.println("GumballMachine <tên> <hang tồn kho>");
            Hệ thống. thoát(1);
        }

        đếm = Integer.parseInt(args[1]);
        GumballMachine gumballMachine = new GumballMachine(args[0], count);

        Màn hình GumballMonitor = new GumballMonitor(gumballMachine);

        // phần còn lại của mã kiểm tra ở đây
    }
}

```

Nhập vị trí và số lượng kẹo cao su ban đầu vào dòng lệnh.

Đừng quên cung cấp cho hàm tạo vị trí và số lượng...

...và khởi tạo một màn hình và truyền cho nó một máy tính để cung cấp báo cáo.

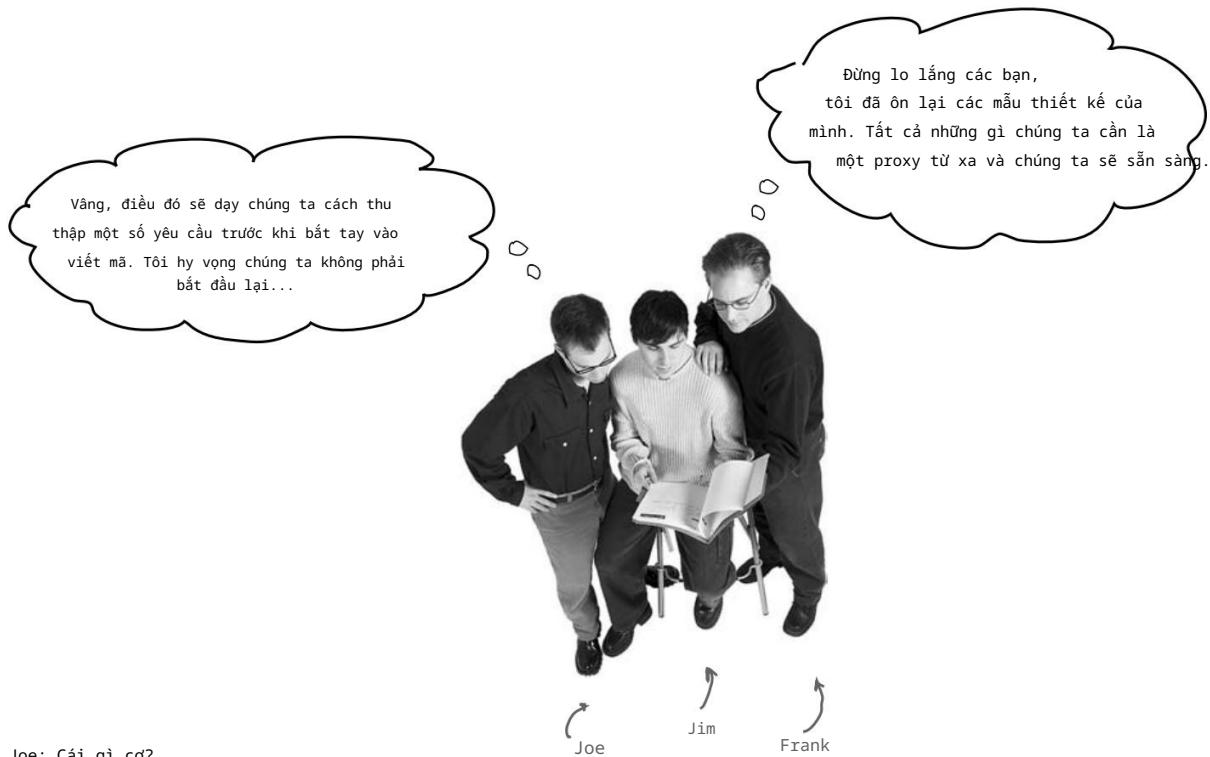
```
Cửa sổ chính sửa tệp Trợ giúp FlyingFish
%java GumballMachineTestDrive Seattle 112
Máy Gumball: Seattle
Số lượng hiện tại: 112 viên kẹo cao su
Tình trạng hiện tại: đang chờ quý
```



Đầu ra của màn hình trông tuyệt vời,
nhưng tôi đoán là tôi đã không nói rõ.
Tôi cần phải giám sát máy bán kẹo cao su TỪ XA!
Trên thực tế, chúng ta đã có mạng lưới để
giám sát rồi. Thời nào các bạn, các bạn
được cho là thẻ hệ Internet mà!

Và đây là kết quả!

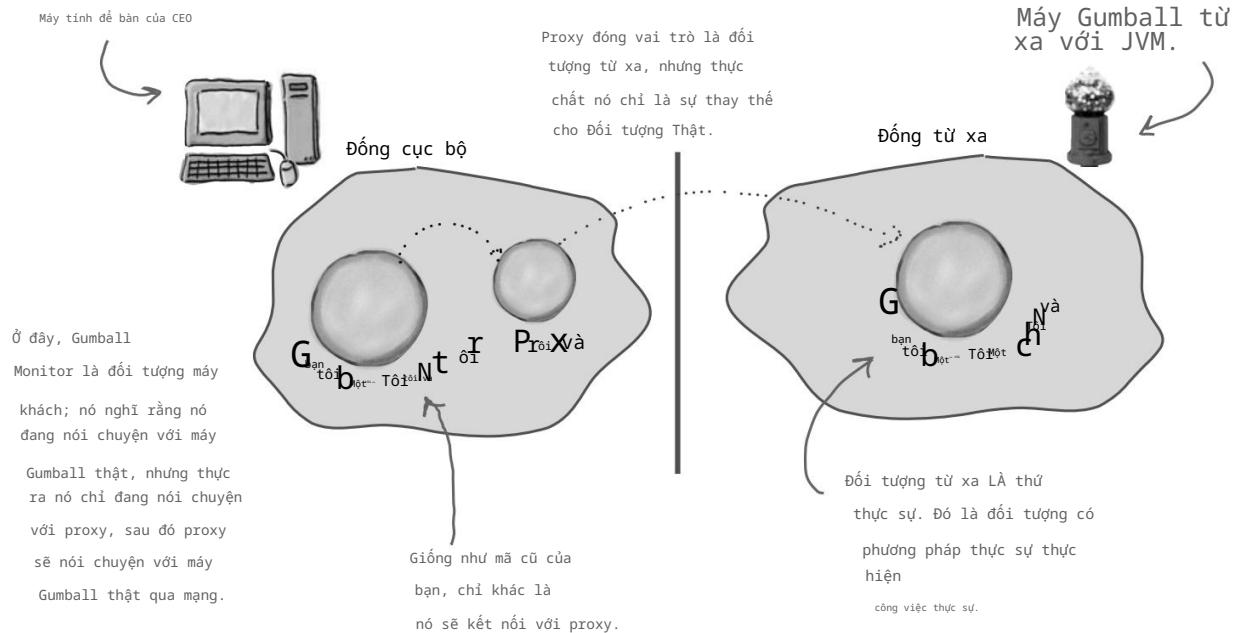
mẫu proxy



proxy từ xa

Vai trò của 'proxy từ xa'

Proxy từ xa hoạt động như một đại diện cục bộ cho một đối tượng từ xa. "Đối tượng từ xa" là gì? Đó là một đối tượng nằm trong heap của một Java Virtual Machine khác (hay nói chung hơn là một đối tượng từ xa đang chạy trong một không gian địa chỉ khác). "Đại diện cục bộ" là gì? Đó là một đối tượng mà bạn có thể gọi các phương thức cục bộ và chuyển tiếp chúng đến đối tượng từ xa.



Đối tượng máy khách của bạn hoạt động như thể đang thực hiện các cuộc gọi phương thức từ xa.

Nhưng chức năng thực sự của nó là gọi các phương thức trên đối tượng 'proxy' cục bộ trên heap để xử lý mọi chi tiết cấp thấp của giao tiếp mạng.

mẫu proxy



não Apower

Trước khi đi sâu hơn, hãy suy nghĩ về cách bạn thiết kế một hệ thống để cho phép gọi phương thức từ xa. Bạn sẽ làm thế nào để nhà phát triển có thể viết càng ít mã càng tốt? Bạn sẽ làm thế nào để việc gọi từ xa trở nên liền mạch?

não Apower 2

Việc thực hiện cuộc gọi từ xa có nên hoàn toàn minh bạch không? Đó có phải là một ý tưởng hay không? Vấn đề có thể gặp phải với cách tiếp cận đó là gì?

Đường vòng RMI

Thêm proxy từ xa vào Gumball Mã giám sát máy

Trên lý thuyết thì có vẻ ổn, nhưng làm sao chúng ta có thể tạo một proxy biết cách gọi phương thức trên một đối tượng nằm trong JVM khác?

Hmmm. Vâng, bạn không thể lấy tham chiếu đến cái gì đó trên một đồng khác, đúng không? Nói cách khác, bạn không thể nói:

Vịt d = <đối tượng trong đồng khác>

Bất kỳ biến d nào tham chiếu đến đều phải nằm trong cùng một không gian heap với mã chạy câu lệnh. Vậy chúng ta tiếp cận vấn đề này như thế nào? Vâng, đó là nơi mà Remote Method Invocation của Java xuất hiện... RMI cung cấp cho chúng ta một cách để tìm các đối tượng trong JVM từ xa và cho phép chúng ta gọi các phương thức của chúng.

Bạn có thể đã gặp RMI trong Head First Java; nếu chưa, chúng ta sẽ đi sâu hơn một chút và tìm hiểu về RMI trước khi thêm hỗ trợ proxy vào mã Gumball Machine.

Vậy, đây là những gì chúng ta sẽ làm:

- 1 Đầu tiên, chúng ta sẽ đi theo RMI Detour và khám phá RMI. Ngay cả khi bạn đã quen với RMI, bạn vẫn có thể muốn đi theo và khám phá phong cảnh.
- 2 Sau đó, chúng ta sẽ sử dụng GumballMachine và biến nó thành một dịch vụ từ xa cung cấp một tập hợp các lệnh gọi phương thức có thể được gọi từ xa.
- 3 Sau đó, chúng ta sẽ tạo một proxy có thể giao tiếp với GumballMachine từ xa, một lần nữa sử dụng RMI và xây dựng lại hệ thống giám sát để CEO có thể giám sát bất kỳ số lượng máy từ xa nào.



Một Đường Vòng RMI

Nếu bạn mới biết đến RMI,
hãy đọc qua vài trang tiếp
theo; nếu không, bạn có thể
chỉ muốn lướt qua nhanh phần này
như một phần đánh giá.

mẫu proxy

Phương pháp từ xa 101



Giả sử chúng ta muốn thiết kế một hệ thống cho phép chúng ta gọi một đối tượng cục bộ chuyển tiếp mỗi yêu cầu đến một đối tượng từ xa. Chúng ta sẽ thiết kế nó như thế nào? Chúng ta sẽ cần một vài đối tượng trợ giúp thực sự thực hiện việc giao tiếp cho chúng ta. Các trợ giúp giúp máy khách có thể hoạt động như thế nó đang gọi một phương thức trên một đối tượng cục bộ (thực tế là như vậy). Máy khách gọi một phương thức trên trợ giúp máy khách, như thế trợ giúp máy khách là dịch vụ thực tế. Sau đó, trợ giúp máy khách sẽ xử lý việc chuyển tiếp yêu cầu đó cho chúng ta.

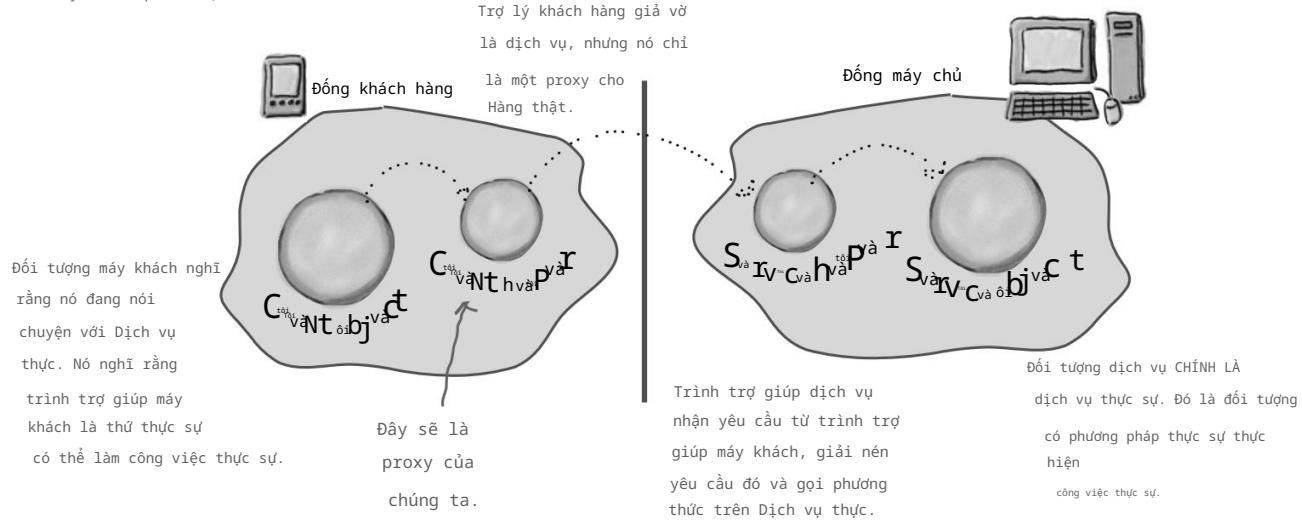
Nói cách khác, đối tượng máy khách nghĩ rằng nó đang gọi một phương thức trên dịch vụ từ xa, vì trình trợ giúp máy khách đang giả vờ là đối tượng dịch vụ. Giả vờ là thứ có phương thức mà máy khách muốn gọi.

Nhưng trình trợ giúp khách hàng không thực sự là dịch vụ từ xa. Mặc dù trình trợ giúp khách hàng hoạt động giống như vậy (vì nó có cùng phương thức mà dịch vụ đang quảng cáo), trình trợ giúp khách hàng không có bất kỳ logic phương thức thực tế nào mà khách hàng mong đợi. Thay vào đó, trình trợ giúp khách hàng liên hệ với máy chủ, chuyển thông tin về lệnh gọi phương thức (ví dụ: tên phương thức, đối số, v.v.) và chờ máy chủ trả về.

Ở phía máy chủ, trình trợ giúp dịch vụ nhận được yêu cầu từ trình trợ giúp máy khách (qua kết nối Socket), giải nén thông tin về cuộc gọi, sau đó gọi phương thức thực trên đối tượng dịch vụ thực. Vì vậy, đối với đối tượng dịch vụ, cuộc gọi là cục bộ. Nó đến từ trình trợ giúp dịch vụ, không phải từ máy khách từ xa.

Trình trợ giúp dịch vụ lấy giá trị trả về từ dịch vụ, đóng gói và chuyển lại (qua luồng đầu ra của Socket) cho trình trợ giúp máy khách. Trình trợ giúp máy khách giải nén thông tin và trả về giá trị cho đối tượng máy khách.

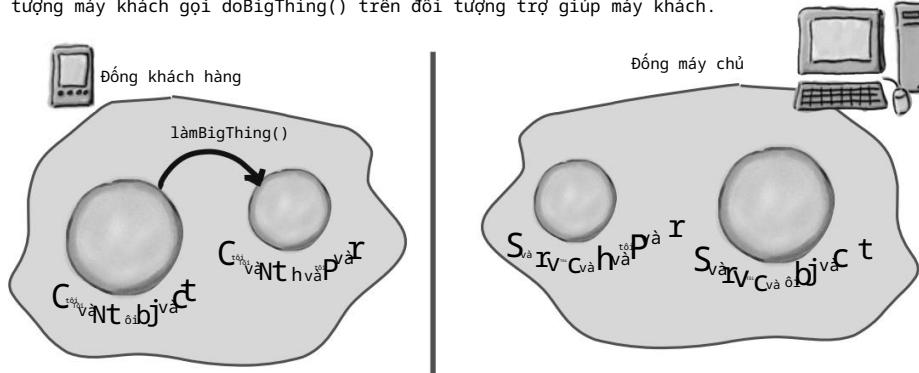
Điều này có vẻ quen thuộc...



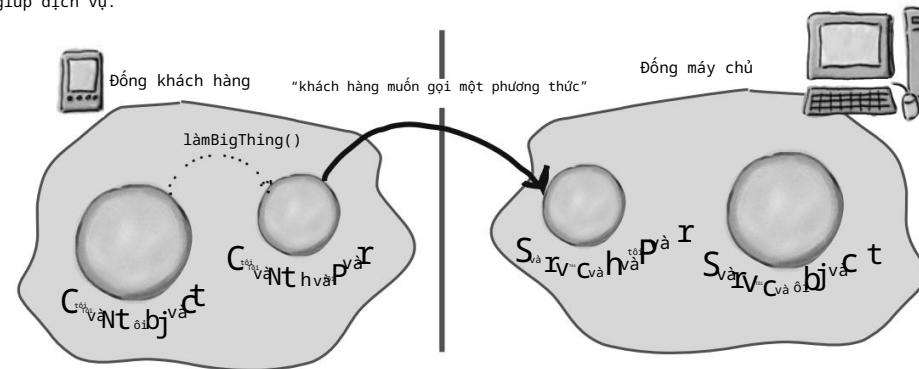
gọi phương thức từ xa

Cuộc gọi phương thức diễn ra như thế nào

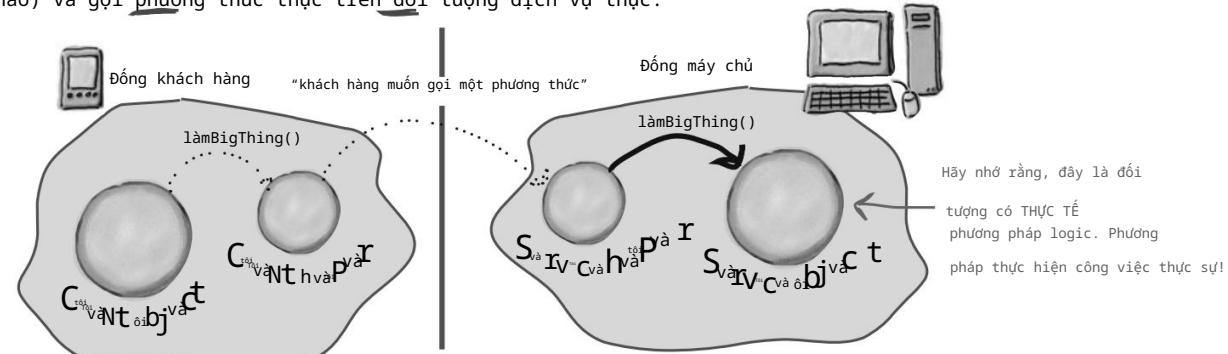
- 1 Đôi tượng máy khách gọi `doBigThing()` trên đối tượng trợ giúp máy khách.



- 2 Trình trợ giúp máy khách đóng gói thông tin về cuộc gọi (đối số, tên phương thức, v.v.) và chuyển thông tin đó qua mạng đến trình trợ giúp dịch vụ.



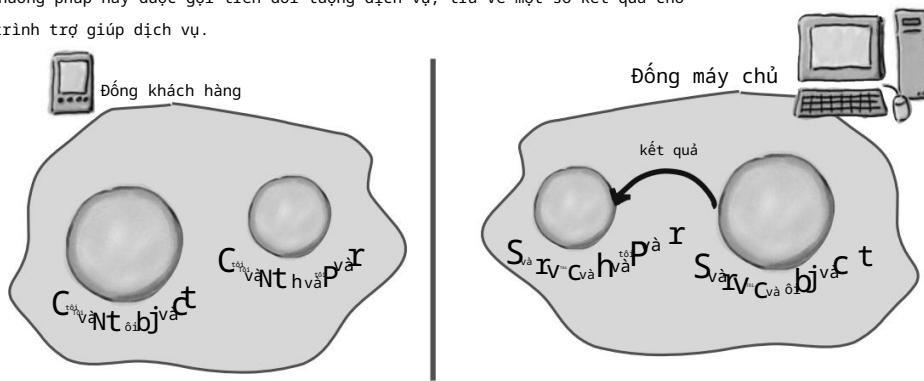
- 3 Trình trợ giúp dịch vụ giải nén thông tin từ trình trợ giúp máy khách, tìm ra phương thức nào cần gọi (và trên đối tượng nào) và gọi phương thức thực trên đối tượng dịch vụ thực.



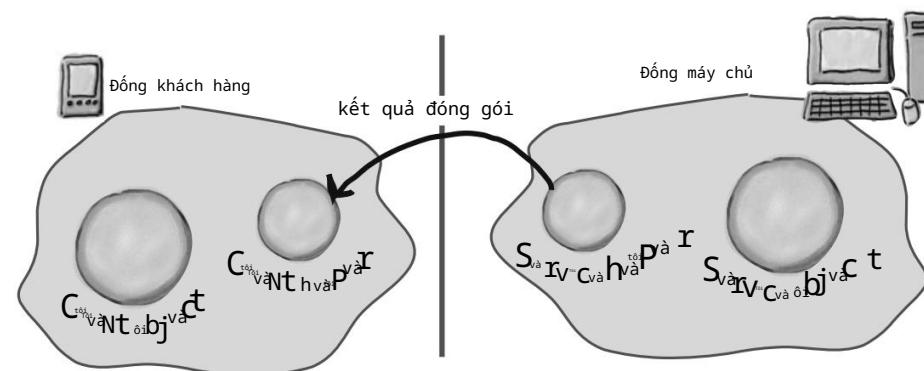
mẫu proxy



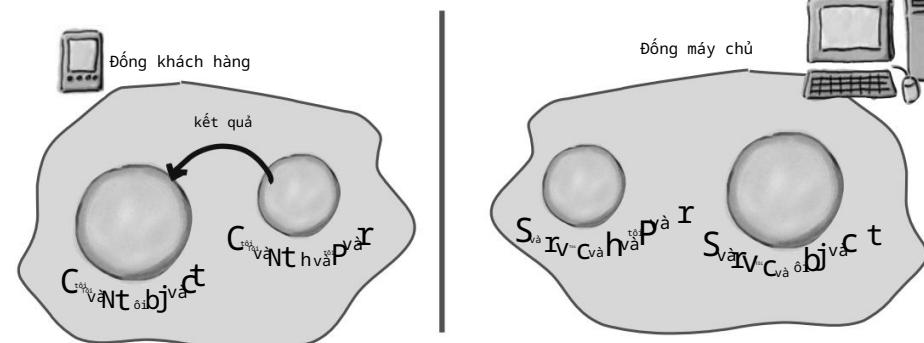
- ④ Phương pháp này được gọi trên đối tượng dịch vụ, trả về một số kết quả cho trình trợ giúp dịch vụ.



- ⑤ Trợ lý dịch vụ đóng gói thông tin trả về từ cuộc gọi và chuyển thông tin đó qua mạng đến trợ lý khách hàng.



- ⑥ Trình trợ giúp khách hàng giải nén các giá trị được trả về và trả về chúng cho đối tượng khách hàng. Đối với đối tượng khách hàng, tất cả đều minh bạch.



Bạn đang ở đây 4 439

RMI: bức tranh toàn cảnh

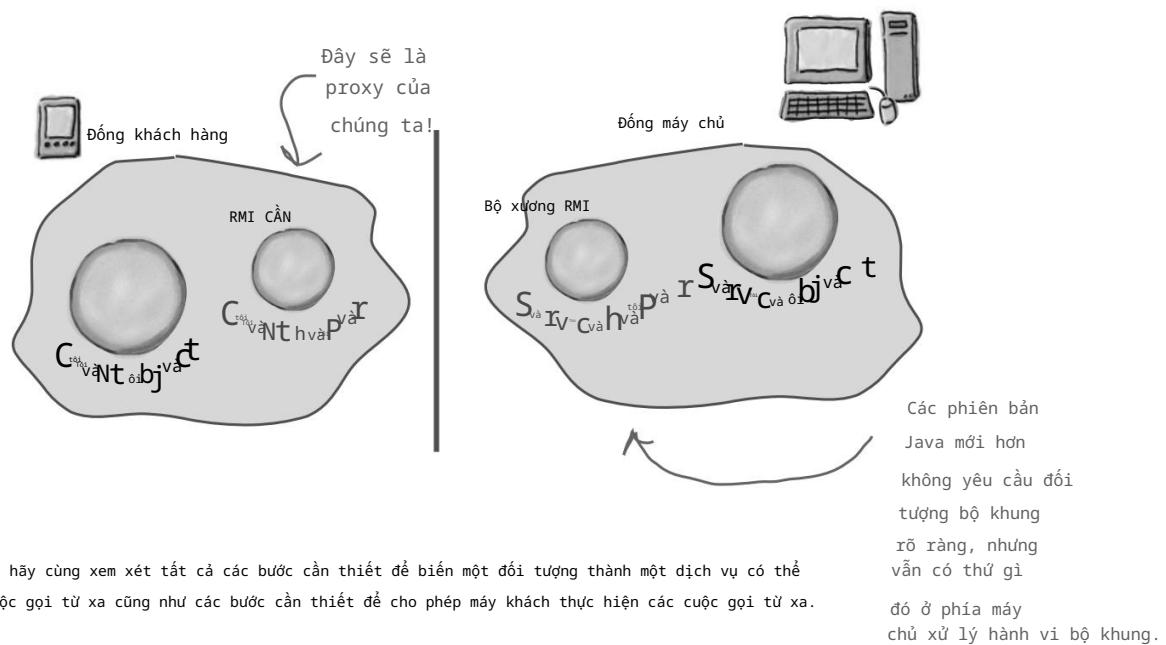
Java RMI, bức tranh toàn cảnh

Được rồi, bạn đã hiểu cơ bản về cách thức hoạt động của phương thức từ xa; bây giờ bạn chỉ cần hiểu cách sử dụng RMI để kích hoạt lệnh gọi phương thức từ xa.

RMI thực hiện cho bạn là xây dựng các đối tượng trợ giúp dịch vụ và máy khách, cho đến việc tạo ra một đối tượng trợ giúp máy khách với cùng phương thức như dịch vụ từ xa. Điểm tuyệt vời của RMI là bạn không phải tự viết bất kỳ mã mạng hoặc I/O nào. Với máy khách, bạn gọi các phương thức từ xa (tức là các phương thức mà Dịch vụ thực có) giống như các lệnh gọi phương thức thông thường trên các đối tượng đang chạy trong JVM cục bộ của máy khách.

RMI cũng cung cấp toàn bộ cơ sở hạ tầng thời gian chạy để mọi thứ hoạt động, bao gồm dịch vụ tra cứu mà máy khách có thể sử dụng để tìm và truy cập các đối tượng từ xa.

Danh pháp RMI: trong RMI, trình trộn giúp máy khách là một 'phản thô' và trình trộn giúp dịch vụ là một 'bộ khung'.



Bây giờ chúng ta hãy cùng xem xét tất cả các bước cần thiết để biến một đối tượng thành một dịch vụ có thể chấp nhận các cuộc gọi từ xa cũng như các bước cần thiết để cho phép máy khách thực hiện các cuộc gọi từ xa.

Bạn có thể muốn đảm bảo rằng mình đã thắt dây an toàn; đường đi có rất nhiều bậc thang và một vài chỗ gò ghè và khúc cua - nhưng không có gì đáng lo ngại cả.

mẫu proxy

Tạo dịch vụ từ xa

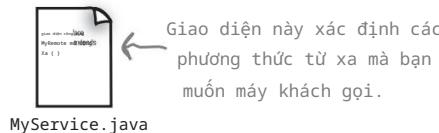
Đây là tổng quan về năm bước để tạo dịch vụ từ xa. Nói cách khác, các bước cần thiết để lấy một đối tượng thông thường và siêu nạp nó để nó có thể được gọi bởi một máy khách từ xa. Chúng ta sẽ thực hiện điều này sau với GumballMachine của mình. Bây giờ, hãy cùng tìm hiểu các bước và sau đó chúng ta sẽ giải thích chi tiết từng bước.



Bước một:

Tạo một giao diện từ xa

Giao diện từ xa xác định các phương thức mà máy khách có thể gọi từ xa. Đó là những gì máy khách sẽ sử dụng làm loại lớp cho dịch vụ của bạn. Cả Stub và dịch vụ thực tế sẽ triển khai điều này!

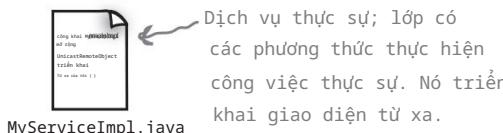


Giao diện này xác định các phương thức từ xa mà bạn muốn máy khách gọi.

Bước hai:

Thực hiện triển khai từ xa

Đây là lớp thực hiện Công việc thực sự. Nó có triển khai thực sự của các phương thức từ xa được định nghĩa trong giao diện từ xa.



Dịch vụ thực sự; lớp có các phương thức thực hiện công việc thực sự. Nó triển khai giao diện từ xa.

Bước ba:

Tạo stub và skeleton bằng rmic

Đây là các 'trình trợ giúp' của máy khách và máy chủ. Bạn không cần phải tạo các lớp này hoặc xem mã nguồn tạo ra chúng. Tất cả đều được xử lý tự động khi bạn chạy công cụ rmic đi kèm với bộ công cụ phát triển Java của bạn.

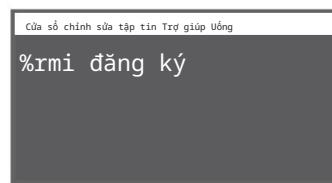
Chạy rmic trên lớp triển khai dịch vụ thực tế...
...đưa ra hai lớp mới cho các đối tượng trợ giúp giúp.



Bước bốn:

Bắt đầu đăng ký RMI (rmiregistry)

rmiregistry giống như những trang trắng của một cuốn danh bạ điện thoại. Đó là nơi máy khách đến để lấy proxy (đối tượng stub/helper của máy khách).



Chạy lệnh này trong một thiết bị đầu cuối riêng biệt.



bạn đang ở đây 4 441

tạo một giao diện từ xa

Bước một: tạo một giao diện từ xa

① Mở rộng java.rmi.Remote

Remote là giao diện 'marker', nghĩa là nó không có phương thức nào. Tuy nhiên, nó có ý nghĩa đặc biệt đối với RMI, vì vậy bạn phải tuân theo quy tắc này. Lưu ý rằng chúng tôi nói 'extends' ở đây. Một giao diện được phép mở rộng giao diện khác.

giao diện công khai MyRemote mở rộng Remote {

Điều này cho chúng ta biết rằng giao diện sẽ được sử dụng để hỗ trợ các cuộc gọi từ xa.

② Khai báo rằng tất cả các phương thức đều ném RemoteException

Giao diện từ xa là giao diện mà máy khách sử dụng làm kiểu cho dịch vụ. Nói cách khác, máy khách gọi các phương thức trên một thứ gì đó triển khai giao diện từ xa. Tất nhiên, thứ đó là stub, và vì stub đang thực hiện mạng và I/O, nên mọi loại lỗi tồi tệ đều có thể xảy ra. Máy khách phải thừa nhận rủi ro bằng cách xử lý hoặc khai báo các ngoại lệ từ xa. Nếu các phương thức trong một giao diện khai báo các ngoại lệ, thì bất kỳ phương thức gọi mã nào trên một tham chiếu của kiểu đó (kiểu giao diện) phải xử lý hoặc khai báo các ngoại lệ.

nhập java.rmi.*;

← Giao diện từ xa nằm trong java.rmi

giao diện công khai MyRemote mở rộng Remote {
 public String sayHello() ném RemoteException;
}

Mỗi lệnh gọi phương thức từ xa đều được coi là 'rủi ro'. Khai báo RemoteException trên mọi phương thức buộc máy khách phải chú ý và thừa nhận rằng mọi thứ có thể không hoạt động.

③ Đảm bảo các đối số và giá trị trả về là nguyên thủy hoặc có thể tuần tự hóa

Các đối số và giá trị trả về của một phương thức từ xa phải là nguyên thủy hoặc có thể tuần tự hóa. Hãy nghĩ về điều đó. Bất kỳ đối số nào cho một phương thức từ xa đều phải được đóng gói và vận chuyển qua mạng, và điều đó được thực hiện thông qua tuần tự hóa. Tương tự với các giá trị trả về. Nếu bạn sử dụng các nguyên thủy, chuỗi và phần lớn các kiểu trong API (bao gồm mảng và bộ sưu tập), bạn sẽ ôn Java nếu bạn cần làm điều đó. Nếu bạn đang truyền các kiểu dữ liệu của riêng mình, hãy đảm bảo rằng các lớp của bạn triển khai Serializable.

Kiểm tra Head First

mới trí nhớ của mình trên Serializable.

public String sayHello() ném RemoteException;

← Giá trị trả về này sẽ được chuyển qua đường dây từ máy chủ trả lại máy khách, vì vậy nó phải có thể Serializable. Đó là cách các đối số và giá trị trả về được đóng gói và gửi đi.

Bước hai: thực hiện triển khai từ xa



① Triển khai giao diện từ xa

Dịch vụ của bạn phải triển khai giao diện từ xa—giao diện có các phương thức mà máy khách của bạn sẽ gọi.

```
lớp công khai MyRemoteImpl mở rộng UnicastRemoteObject triển khai MyRemote {
    công khai String sayHello() {
        trả về "Máy chủ nói, 'Này'";
    }
    // thêm mã trong lớp
}
```



Trình biên dịch sẽ đảm bảo rằng bạn đã triển khai tất cả các phương thức từ giao diện mà bạn triển khai.
Trong trường hợp này, chỉ có một.

② Mở rộng UnicastRemoteObject

Để hoạt động như một đối tượng dịch vụ từ xa, đối tượng của bạn cần một số chức năng liên quan đến 'ở xa'. Cách đơn giản nhất là mở rộng UnicastRemoteObject (từ gói `java.rmi.server`) và để lớp đó (siêu lớp của bạn) thực hiện công việc cho bạn.

```
lớp công khai MyRemoteImpl mở rộng UnicastRemoteObject triển khai MyRemote {
```

③ Viết một hàm tạo không có đối số khai báo RemoteException

Siêu lớp mới của bạn, `UnicastRemoteObject`, có một vấn đề nhỏ—hàm khởi tạo của nó ném một `RemoteException`. Cách duy nhất để giải quyết vấn đề này là khai báo một hàm khởi tạo cho triển khai từ xa của bạn, chỉ để bạn có một nơi để khai báo `RemoteException`. Hãy nhớ rằng, khi một lớp được khởi tạo, hàm khởi tạo siêu lớp của nó luôn được gọi. Nếu hàm khởi tạo siêu lớp của bạn ném một ngoại lệ, bạn không có lựa chọn nào khác ngoài việc khai báo rằng hàm khởi tạo của bạn cũng ném một ngoại lệ.

```
công khai MyRemoteImpl() ném RemoteException { }
```



Bạn không cần phải đưa bất cứ thứ gì vào constructor. Bạn chỉ cần một cách để khai báo rằng constructor siêu lớp của bạn sẽ ném một ngoại lệ.

④ Đăng ký dịch vụ với sổ đăng ký RMI

Bây giờ bạn đã có một dịch vụ từ xa, bạn phải làm cho nó khả dụng với các máy khách từ xa. Bạn thực hiện việc này bằng cách khởi tạo nó và đưa nó vào sổ đăng ký RMI (phải chạy hoặc dòng mã này sẽ không thành công). Khi bạn đăng ký đối tượng triển khai, hệ thống RMI thực sự đặt stub vào sổ đăng ký, vì đó là những gì máy khách thực sự cần. Đăng ký dịch vụ của bạn bằng phương thức `static rebind()` của lớp `java.rmi.Naming`.

```
thử {
```

```
    Dịch vụ MyRemote = new MyRemoteImpl();
    Naming.rebind("RemoteHello", dịch vụ);
} catch(Ngoại lệ ex) {...}
```

Đặt tên cho dịch vụ của bạn (mà khách hàng có thể sử dụng để tra cứu trong sổ đăng ký) và đăng ký với sổ đăng ký RMI. Khi bạn liên kết đối tượng dịch vụ, RMI hoàn đổi dịch vụ cho stub và đặt stub vào sổ đăng ký.

gốc và bộ xương

Bước ba: tạo stub và skeleton

- Chạy rmic trên lớp triển khai từ xa (không phải giao diện từ xa)

Công cụ rmic, đi kèm với bộ công cụ phát triển phần mềm Java, lấy một triển khai dịch vụ và tạo ra hai lớp mới, stub và skeleton. Nó sử dụng quy ước đặt tên là tên của triển khai từ xa của bạn, với _Stub hoặc _Skel được thêm vào cuối. Có các tùy chọn khác với rmic, bao gồm không tạo skeleton, xem mã nguồn cho các lớp này trông như thế nào và thậm chí sử dụng IIOP làm giao thức. Cách chúng tôi thực hiện ở đây là cách bạn thường làm. Các lớp sẽ nằm trong thư mục hiện tại (tức là bất kỳ thứ gì bạn đã thực hiện lệnh cd). Hãy nhớ rằng, rmic phải có thể nhìn thấy lớp triển khai của bạn, vì vậy bạn có thể sẽ chạy rmic từ thư mục nơi triển khai từ xa của bạn nằm. (Chúng tôi cố tình không sử dụng các gói ở đây để đơn giản hóa. Trong Thế giới thực, bạn sẽ cần tính đến các cấu trúc thư mục gói và tên đủ điều kiện).

RMIC tạo ra hai
mới lớp học cho
Lưu ý rằng bạn không nói
đối tượng trợ giúp:
".class" ở cuối. Chỉ cần tên lớp.

```
Cửa sổ chính sửa tập tin Trợ giúp Whuffie
%rmic MyRemoteImpl
```



Lớp MyRemoteImpl_Stub



Lớp MyRemoteImpl_Skel

Bước bốn: chạy rmiregistry

- Mở terminal và khởi động rmiregistry.

Hãy chắc chắn rằng bạn bắt đầu nó từ một thư mục có quyền truy cập vào các lớp của bạn. Cách đơn giản nhất là bắt đầu nó từ thư mục 'classes' của bạn.

Trợ giúp về cửa sổ chính sửa tập tin Hà?

%rmi đăng ký

Bước năm: bắt đầu dịch vụ

- Mở một thiết bị đầu cuối khác và bắt đầu dịch vụ của bạn

Điều này có thể đến từ phương thức main() trong lớp triển khai từ xa của bạn hoặc từ một lớp khởi chạy riêng biệt. Trong ví dụ đơn giản này, chúng ta đặt mã khởi động vào lớp triển khai, trong phương thức chính khởi tạo đối tượng và đăng ký nó với số đăng ký RMI.

Trợ giúp về cửa sổ chính sửa tập tin Hà?

%java MyRemoteImpl

mẫu proxy

Mã hoàn chỉnh cho phía máy chủ



Giao diện từ xa:

```

nhập java.rmi.*;           ← RemoteException và giao
                            dien Remote nằm trong gói java.rmi.

giao diện công khai MyRemote mở rộng Remote {
    public String sayHello() ném RemoteException;
}

```

← Giao diện của bạn PHẢI mở rộng java.rmi.Remote

← Tất cả các phương thức từ xa của bạn
phải khai báo RemoteException.

Dịch vụ từ xa (việc triển khai):

```

nhập java.rmi.*;           ← UnicastRemoteObject nằm
nhập java.rmi.server.*;   ← trong gói java.rmi.server. Mỗi rộng UnicastRemoteObject là cách
                            dễ nhất để tạo một đối tượng từ xa.

lớp công khai MyRemoteImpl mở rộng UnicastRemoteObject triển khai MyRemote {
    công khai String sayHello() {           ← Tất nhiên, bạn phải triển
        trả về "Máy chủ nói, 'Này'";
    }
}

công khai MyRemoteImpl() ném RemoteException { }           ← Trình xây dựng siêu lớp của bạn (đối
                                                               với UnicastRemoteObject) khai báo một ngoại lệ, do đó
                                                               BẠN PHẢI triển khai giao
                                                               dien từ xa của mình!!

public static void main (String[] args) {
    thử {
        Dịch vụ MyRemote = new MyRemoteImpl();
        Naming.rebind("RemoteHello", dịch vụ);
    } catch(Ngoại lệ ex) {
        ví dụ: printStackTrace();
    }
}

```

← BẠN PHẢI triển khai giao
dien từ xa của mình!!

← Tất nhiên, bạn phải triển
khai tất cả các phương thức giao
diện. Nhưng lưu ý rằng bạn
KHÔNG phải khai báo RemoteException.

← Trình xây dựng siêu lớp của bạn (đối
với UnicastRemoteObject) khai báo một ngoại lệ, do đó
BẠN phải viết một trình xây dựng, vì điều đó có nghĩa
là trình xây dựng của bạn đang gọi mā rủi ro (trình
xây dựng siêu lớp của nó).

← Tạo đối tượng từ xa, sau đó 'liên kết' nó với
rmiregistry bằng cách sử dụng Naming.rebind()
tĩnh. Tên bạn đăng ký nó là tên mà máy khách sẽ
sử dụng để tra cứu nó trong sổ đăng ký RMI.

bạn đang ở đây 4 445

làm thế nào để có được đối tượng stub

Làm thế nào để máy khách có được
đối tượng stub?

Máy khách phải lấy đối tượng stub (proxy của chúng tôi),
vì đó là thứ mà máy khách sẽ gọi phương thức. Và đó là nơi
số đăng ký RMI xuất hiện. Máy khách thực hiện 'tra
cứu', giống như việc tìm đến các trang trắng của danh bạ
diện thoại, và về cơ bản là nói, "Đây là một cái tên, và
tôi muốn stub đi kèm với cái tên đó."

Chúng ta hãy xem xét đoạn mã cần thiết để tra cứu và
lấy một đối tượng stub.

Dưới đây là cách thức hoạt động.



Mã Gắn

Máy khách luôn sử dụng giao diện
từ xa làm loại dịch vụ.
Trên thực tế, khách hàng không bao
giờ cần biết tên lớp thực tế của bạn
dịch vụ từ xa.

Dịch vụ MyRemote = (MyRemote)

```
Naming.lookup("rmi://127.0.0.1/RemoteHello");
```

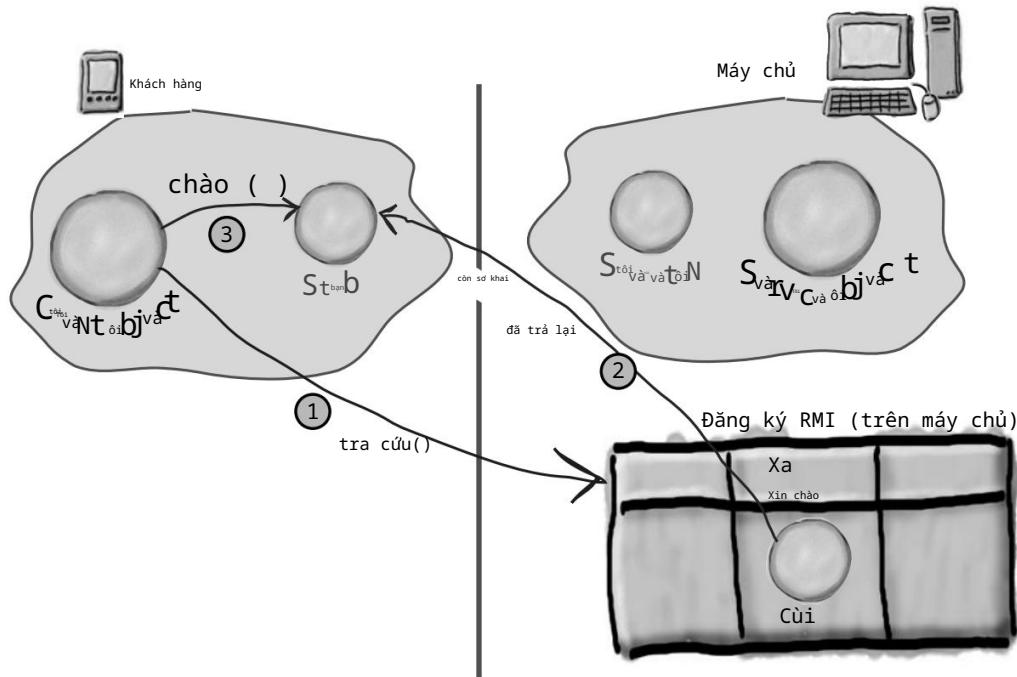
Bạn phải ép kiểu nó thành
giao diện vì phương thức
tra cứu trả về kiểu Object.

lookup() là phương thức
tĩnh của lớp Naming.

Đây phải là tên
mà dịch vụ đã
được đăng ký.

Tên máy chủ hoặc
địa chỉ IP nơi
dịch vụ đang chạy.

mẫu proxy



Nó hoạt động như thế nào...

- 1 Khách hàng thực hiện tra cứu trên sổ đăng ký RMI

```
Đặt tên.lookup("rmi://127.0.0.1/RemoteHello");
```

- 2 Đăng ký RMI trả về đối tượng stub

(là giá trị trả về của phương thức tra cứu) và RMI tự động hủy tuần tự hóa stub. Bạn PHẢI có lớp stub (mà rmic tạo ra cho bạn) trên máy khách, nếu không stub sẽ không được hủy tuần tự hóa.

- 3 Khách hàng gọi một phương thức trên stub, như thể stub là dịch vụ thực sự

khách hàng từ xa

Mã khách hàng hoàn chỉnh

nhập java.rmi.*;

Lớp Naming (để thực hiện tra cứu rmiregistry) nằm trong gói java.rmi.

```

lớp công khai MyRemoteClient {
    public static void main (String[] args) {
        mới MyRemoteClient().go();
    }

    công khai void go() {
        thử {
            Dịch vụ MyRemote = (MyRemote) Naming.lookup("rmi://127.0.0.1/RemoteHello");

            Chuỗi s = service.sayHello();
            Hệ thống.out.println(s);
        } catch(Ngoại lệ ex) {
            ví dụ: printStackTrace();
        }
    }
}

```

Nó được lấy ra khỏi số đăng ký dưới dạng kiểu Object, vì vậy đừng quên ép kiểu.

Bạn cần địa chỉ IP hoặc tên máy chủ.

và tên được sử dụng để liên kết/liên kết lại dịch vụ.

Nó trông giống như một lệnh gọi phương thức thông thường! (Ngoại trừ việc nó phải xác nhận RemoteException.)



Những điều thú vị

Làm thế nào để máy khách có được lớp stub?

Bây giờ chúng ta đến với câu hỏi thú vị. Bằng cách nào đó, theo một cách nào đó, máy khách phải có lớp stub (mà bạn đã tạo trước đó bằng rmic) tại thời điểm máy khách thực hiện tra cứu, nếu không stub sẽ không được giải mã tự động trên máy khách và toàn bộ mọi thứ sẽ nổ tung. Máy khách cũng cần các lớp cho bất kỳ đối tượng được mã hóa nào được trả về bởi các lệnh gọi phương thức đến đối tượng từ xa. Trong một hệ thống đơn giản, bạn có thể chỉ cần giao thủ công các lớp này cho máy khách.

Có một cách thú vị hơn nhiều, mặc dù nằm ngoài phạm vi của cuốn sách này. Nhưng trong trường hợp bạn quan tâm, cách thú vị hơn được gọi là "tải xuống lớp động". Với tải xuống lớp động, các đối tượng được mã hóa (như stub) được "đóng dấu" bằng một URL cho hệ thống RMI trên máy khách biết nơi tìm tệp lớp cho đối tượng đó. Sau đó, trong quá trình hủy mã hóa một đối tượng, nếu RMI không thể tìm thấy lớp cục bộ, nó sẽ sử dụng URL đó để thực hiện HTTP Get để truy xuất tệp lớp. Vì vậy, bạn sẽ cần một máy chủ web đơn giản để phục vụ các tệp lớp và bạn cũng cần thay đổi một số tham số bảo mật trên máy khách. Có một số vấn đề khó khăn khác khi tải xuống lớp động, nhưng đó là tổng quan.

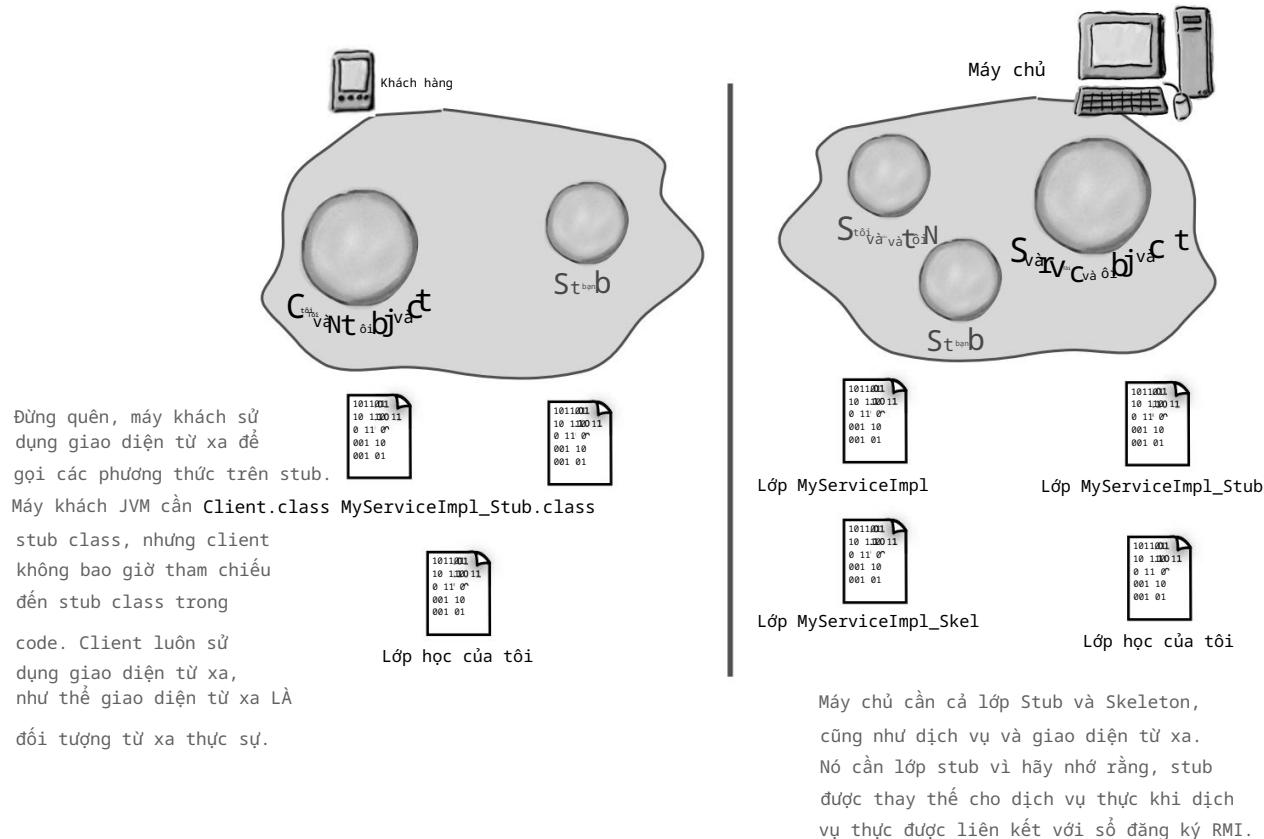
Riêng đối với stub object, có một cách khác để client có thể lấy được class. Tuy nhiên, cách này chỉ khả dụng trong Java 5. Chúng ta sẽ nói sơ qua về cách này ở gần cuối chương.

mẫu proxy



Ba điều sai lầm lớn nhất mà các lập trình viên mắc phải với RMI là:

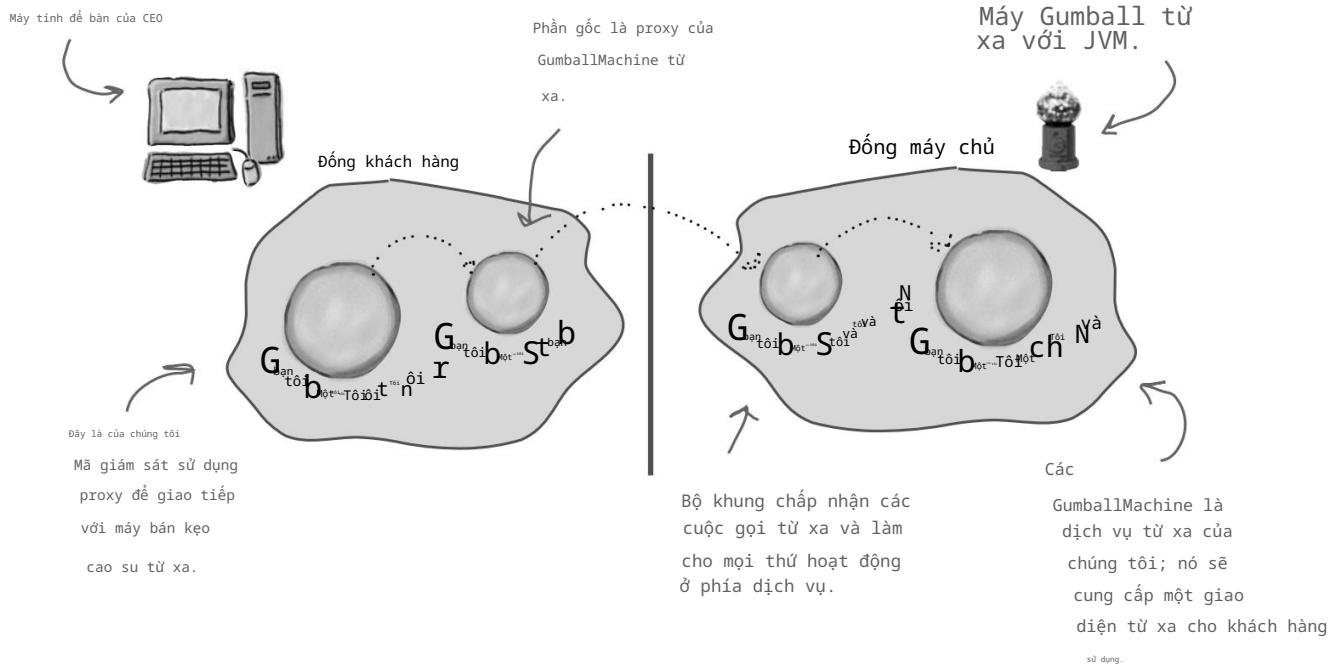
- 1) Quên khởi động rmiregistry trước khi khởi động dịch vụ từ xa (khi dịch vụ được đăng ký bằng Naming.rebind(), rmiregistry phải đang chạy!)
- 2) Quên tạo các đối số và kiểu trả về có thể tuân tự hóa (bạn sẽ không biết cho đến khi chạy; trình biên dịch sẽ không phát hiện ra điều này.)
- 3) Quên cung cấp lớp stub cho máy khách.



màn hình gumball từ xa

Quay lại proxy từ xa GumballMachine của chúng tôi

Được rồi, giờ bạn đã nắm được những kiến thức cơ bản về RMI, bạn đã có các công cụ cần thiết để triển khai proxy từ xa của máy Gumball. Hãy cùng xem GumballMachine phù hợp với khuôn khổ này như thế nào:



Chuẩn bị GumballMachine để trở thành dịch vụ từ xa

Bước đầu tiên trong việc chuyển đổi mã của chúng ta để sử dụng proxy từ xa là cho phép GumballMachine phục vụ các yêu cầu từ xa từ máy khách. Nói cách khác, chúng ta sẽ biến nó thành một dịch vụ. Để làm điều đó, chúng ta cần:

- 1) Tạo giao diện từ xa cho GumballMachine. Giao diện này sẽ cung cấp một tập hợp các phương thức có thể được gọi từ xa.
- 2) Đảm bảo tất cả các kiểu trả về trong giao diện đều có thể tuần tự hóa được.
- 3) Triển khai giao diện trong một lớp cụ thể.

Chúng ta sẽ bắt đầu với giao diện từ xa:

```
Đừng quên import java.rmi.*  
nhập java.rmi.*;
```

Đây là giao diện từ xa.

```
giao diện công khai GumballMachineRemote mở rộng Remote {  
    public int getCount() ném RemoteException;  
    public String getLocation() ném RemoteException;  
    public State getState() ném RemoteException;  
}
```



Sau đây là những phương pháp chúng tôi sẽ hỗ trợ.
Tất cả các kiểu trả về phải là
kiểu nguyên thủy hoặc có
thể tuần tự hóa...



Chúng ta có một kiểu trả về không thể Serializable: lớp State. Hãy sửa nó...

```
nhập java.io.*;
```



Serializable nằm trong gói java.io.

```
Giao diện công khai State mở rộng Serializable {  
    công khai void insertQuarter();  
    công khai void ejectQuarter();  
    công khai void turnCrank();  
    công khai void dispense();  
}
```

Sau đó chúng ta chỉ cần mở rộng giao diện Serializable (không có phương thức nào trong đó).
Và bây giờ State trong tất cả các lớp con có
thể được chuyển qua mạng.

giao diện từ xa cho máy gumball

Trên thực tế, chúng ta vẫn chưa hoàn thành Serializable; chúng ta có một vấn đề với State. Như bạn có thể nhớ, mỗi đối tượng State duy trì một tham chiếu đến một máy gumball để nó có thể gọi các phương thức của máy gumball và thay đổi trạng thái của nó. Chúng ta không muốn toàn bộ máy gumball được tuần tự hóa và chuyển giao với đối tượng State. Có một cách dễ dàng để khắc phục điều này:

```
lớp công khai NoQuarterState thực hiện State {
    tạm thời GumballMachine gumballMachine;

    // tắt cả các phương pháp khác ở đây
}
```

Trong mỗi lần triển khai State, chúng tôi thêm từ khóa transient vào biến
 thể hiện GumballMachine. Điều này cho biết JVM không tuần tự hóa trường này.
 Lưu ý rằng điều này có thể hơi nguy hiểm
 nếu bạn cố truy cập vào trường này sau khi
 nó đã được tuần tự hóa và chuyển giao.

Chúng tôi đã triển khai GumballMachine của mình, nhưng chúng tôi cần đảm bảo rằng nó có thể hoạt động như một dịch vụ và xử lý các yêu cầu đến từ mạng. Để làm được điều đó, chúng tôi phải đảm bảo GumballMachine đang thực hiện mọi thứ cần thiết để triển khai giao diện GumballMachineRemote.

Như bạn đã thấy trong phần vòng vo của RMI, điều này khá đơn giản, tắt cả những gì chúng ta cần làm là thêm một vài thứ...

Đầu tiên, chúng ta cần nhập các gói rmi.

nhập java.rmi.*;
nhập java.rmi.server.*;

GumballMachine sẽ phân lớp UnicastRemoteObject; điều này giúp nó có khả năng hoạt động như một dịch vụ từ xa.

GumballMachine cũng cần triển khai giao diện từ xa...

```
lớp công khai GumballMachine
    mở rộng UnicastRemoteObject triển khai GumballMachineRemote
{
    // các biến thể hiện ở đây

    công khai GumballMachine(Chuỗi vị trí, int sốGumballs) ném RemoteException {
        // mã ở đây
    }

    công khai int getCount() {
        số lần trả về;
    }

    công khai State getState() {
        trả về trạng thái;
    }

    công khai String getLocation() {
        vị trí trả về;
    }

    // các phương pháp khác ở đây
}
```

...và hàm tạo cần phải đưa ra một ngoại lệ từ xa, vì lớp cha sẽ thực hiện điều đó.

Vậy thôi! Không có gì thay đổi ở đây cả!

Đăng ký với cơ quan đăng ký RMI...

Như vậy là hoàn thành dịch vụ máy gumball. Bây giờ chúng ta chỉ cần khởi động nó để nó có thể nhận được yêu cầu.

Đầu tiên, chúng ta cần đảm bảo rằng chúng ta đã đăng ký nó với sổ đăng ký RMI để khách hàng có thể định vị được nó.

Chúng tôi sẽ thêm một đoạn mã nhỏ vào ổ đĩa thử nghiệm để thực hiện việc này giúp chúng tôi:

```
lớp công khai GumballMachineTestDrive {

    public static void main(String[] args) {
        GumballMachineRemote gumballMachine = null;
        số nguyên;
        nếu (args.length < 2) {
            System.out.println("GumballMachine <tên> <hang tồn kho>");
            Hệ thống. thoát(1);
        }
        thử {
            đếm = Integer.parseInt(args[1]);
            gumballMachine = new
                GumballMachine(args[0], count);
            Naming.rebind("//" + args[0] + "/gumballmachine", gumballMachine);
        } catch (Ngoại lề e) {
            e.printStackTrace();
        }
    }
}
```

Chúng ta hãy tiếp tục và bắt đầu thực hiện nhé...

Chạy cái này trước.

Thao tác này sẽ giúp
dịch vụ đăng ký RMI
hoạt động.

Đầu tiên, chúng ta cần thêm khối try/catch xung quanh phiên bản gumball vì hàm tạo của chúng ta hiện có thể đưa ra ngoại lệ.

Chúng tôi cũng thêm lệnh gọi đến
Naming.rebind, lệnh này sẽ xuất bản phần mở
rộng GumballMachine dưới tên gumballmachine.

Chúng tôi đang sử dụng "chính thức"
Máy Gumball Mighty, bạn nên
thay tên máy của mình vào
đây.

Trợ giúp về cửa sổ chính sửa tập tin Hỗ?

% đăng ký rmi

Trợ giúp về cửa sổ chính sửa tập tin Hỗ?

% java GumballMachineTestDrive seattle.mightygumball.com 100

Chạy ngay giây này.

Thao tác này sẽ khởi động và chạy GumballMachine và đăng
ký nó với sổ đăng ký RMI.

khách hàng giám sát gumball

Bây giờ là phần mềm khách GumballMonitor...

Bạn còn nhớ GumballMonitor không? Chúng tôi muốn sử dụng lại mà không cần phải viết lại để hoạt động trên mạng. Vâng, chúng tôi sẽ làm như vậy, nhưng chúng tôi cần thực hiện một vài thay đổi.

```

nhập java.rmi.*;
lớp công khai GumballMonitor {
    GumballMachineMáy từ xa;
    public GumballMonitor(Máy GumballMachineRemote) {
        this.machine = máy móc;
    }
    công khai void báo cáo() {
        thử {
            System.out.println("Máy Gumball: " + machine.getLocation());
            System.out.println("Số lượng hàng tồn kho hiện tại: " + machine.getCount() + " gumballs");
            System.out.println("Trạng thái hiện tại: " + máy.getStat());
        catch (RemoteException e) {
            e.printStackTrace();
        }
    }
}

Chúng ta cần nhập gói RMI vì chúng ta đang sử dụng lớp RemoteException bên dưới...
Bây giờ chúng ta sẽ dựa vào giao diện từ xa thay vì lớp GumballMachine cụ thể.

Chúng ta cũng cần phát hiện mọi ngoại lệ từ xa có thể xảy ra khi chúng ta cố gắng gọi các phương thức cuối cùng diễn ra trên mạng.

```



mẫu proxy

Viết ỏ đĩa thử nghiệm Monitor

Bây giờ chúng ta đã có tất cả các phần cần thiết. Chúng ta chỉ cần viết một số mã để CEO có thể giám sát một loạt máy bán kẹo cao su:

```

Đây là bản thử nghiệm màn hình. Tổng
giám đốc điều hành sẽ chạy thử!

nhập java.rmi.*;
lớp công khai GumballMonitorTestDrive {
    public static void main(String[] args) {
        Chuỗi[] vị trí = {"rmi://santafe.mightygumball.com/gumballmachine",
                          "rmi://boulder.mightygumball.com/gumballmachine",
                          "rmi://seattle.mightygumball.com/gumballmachine"};
        GumballMonitor[] monitor = new GumballMonitor[vị trí.chiều dài];
        dối với (int i=0; i < vị trí.chiều dài; i++) {
            thử {
                GumballMachineMáy từ xa =
                    (GumballMachineRemote) Đặt tên.lookup(vị trí[i]);
                monitor[i] = new GumballMonitor(máy);
                System.out.println(màn hình[i]);
            } catch (Ngoại lệ e) {
                e.printStackTrace();
            }
        }
        dối với (int i = 0; i < chiều dài màn hình; i++) {
            giám sát[i].báo cáo();
        }
    }
}

```

Đây là bản thử nghiệm màn hình. Tổng
giám đốc điều hành sẽ chạy thử!

Sau đây là tất cả các

địa điểm sẽ được giám sát.

Chúng tôi tạo
ra một mảng các vị
trí, mỗi vị
trí cho một máy.

```

        Chuỗi[] vị trí = {"rmi://santafe.mightygumball.com/gumballmachine",
                          "rmi://boulder.mightygumball.com/gumballmachine",
                          "rmi://seattle.mightygumball.com/gumballmachine"};

```

GumballMonitor[] monitor = new GumballMonitor[vị trí.chiều dài];

```

        dối với (int i=0; i < vị trí.chiều dài; i++) {
            thử {
                GumballMachineMáy từ xa =
                    (GumballMachineRemote) Đặt tên.lookup(vị trí[i]);
                monitor[i] = new GumballMonitor(máy);
                System.out.println(màn hình[i]);
            } catch (Ngoại lệ e) {
                e.printStackTrace();
            }
        }

```

dối với (int i = 0; i < chiều dài màn hình; i++) {

giám sát[i].báo cáo();

}

}

}

Chúng tôi tạo
ra một mảng các vị
trí, mỗi vị
trí cho một máy.

Chúng tôi cũng tạo
ra một loạt màn hình.

Bây giờ chúng ta cần có một
proxy cho mỗi máy từ xa.

Sau đó, chúng tôi lặp lại từng
máy và in báo cáo của máy đó.

máy kẹo cao su proxy



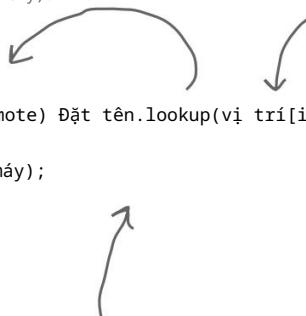
Mã Gắn

Lệnh này sẽ trả về một proxy cho Gumball Machine từ xa (hoặc đưa ra một ngoại lệ nếu không thể xác định được proxy).

```
thử {
    GumballMachineMáy từ xa =
        (GumballMachineRemote) Đặt tên.lookup(vị trí[i]);
    monitor[i] = new GumballMonitor(máy);

} catch (Ngoại lệ e) {
    e.printStackTrace();
}
```

Hãy nhớ rằng, Naming.lookup() là phương thức tĩnh trong gói RMI, phương thức này lấy tên dịch vụ và vị trí rồi tra cứu trong rmiregistry tại vị trí đó.



Khi chúng ta có được proxy cho máy tính từ xa, chúng ta tạo một GumballMonitor mới và truyền cho máy tính đó để giám sát.

Một bản demo khác dành cho CEO của Mighty Gumball...

Được rồi, đã đến lúc tập hợp tất cả công việc này lại và thực hiện một bản demo khác. Trước tiên, hãy đảm bảo một vài máy bán kẹo cao su đang chạy mã mới:

Trên mỗi máy, chạy rmiregistry ở chế độ nền hoặc từ một máy riêng biệt của số terminal...

...và sau đó chạy GumballMachine, cung cấp vị trí và số lượng gumball ban đầu.

```
Trợ giúp về cửa sổ chính sửa tập tin Hỗ trợ?
% rmiregistry &
% java GumballMachine santafe.mightygumball.com 100
```

```
Trợ giúp về cửa sổ chính sửa tập tin Hỗ trợ?
% rmiregistry &
% java GumballMachine boulder.mightygumball.com 100
```

```
Trợ giúp về cửa sổ chính sửa tập tin Hỗ trợ?
% rmiregistry &
% java GumballMachine seattle.mightygumball.com 250
```

máy phô biến! ↗

Và bây giờ chúng ta hãy đưa màn hình vào tay CEO.
Hy vọng lần này anh ấy sẽ thích nó:

```
Cửa sổ chỉnh sửa tệp Trợ giúp GumballsAndBeyond
% java GumballMonitor

Máy Gumball: santafe.mightygumball.com
Số lượng hiện tại: 99 viên kẹo cao su
Trạng thái hiện tại: chờ quý

Máy Gumball: boulder.mightygumball.com
Số lượng hiện tại: 44 viên kẹo cao su
Trạng thái hiện tại: đang chờ quay tay quay

Máy Gumball: seattle.mightygumball.com
Tồn kho hiện tại: 187 viên kẹo cao su
Trạng thái hiện tại: chờ quý
%
```



Trình giám sát sẽ lắp
lại trên từng máy
từ xa và gọi các
phương thức
getLocation(),
getCount() và getState() của máy đó.

Thật tuyệt vời;
nó sẽ cách mạng hóa doanh
nghiệp của tôi và đánh bại
đối thủ cạnh tranh!



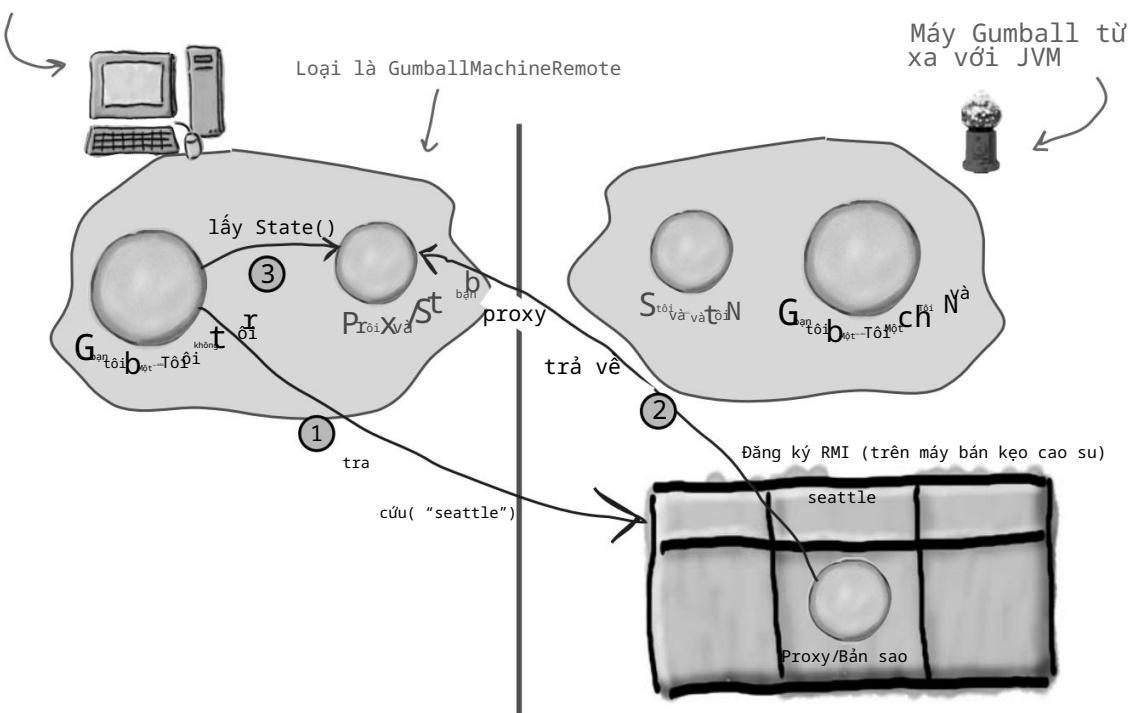
Bằng cách gọi các phương thức trên proxy, một lệnh
gọi từ xa được thực hiện qua dây và một String, một
số nguyên và một đối tượng State được trả về. Vì
chúng ta đang sử dụng proxy, GumballMonitor không
biết hoặc không quan tâm rằng các lệnh gọi là từ xa
(ngoại trừ việc phải lo lắng về các ngoại lệ từ xa).

proxy đằng sau hậu trường



- CEO chạy màn hình, đầu tiên lấy các proxy đến các máy bán kẹo cao su từ xa và sau đó gọi getState() trên mỗi máy (cùng với getCount() và getLocation()).

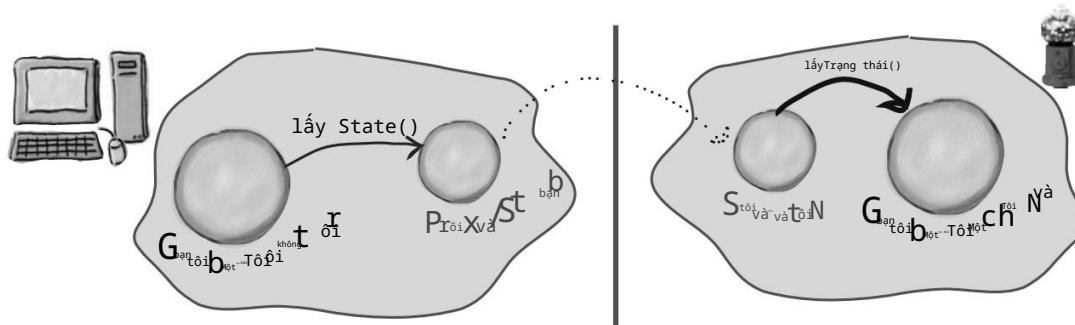
Máy tính để bàn của CEO



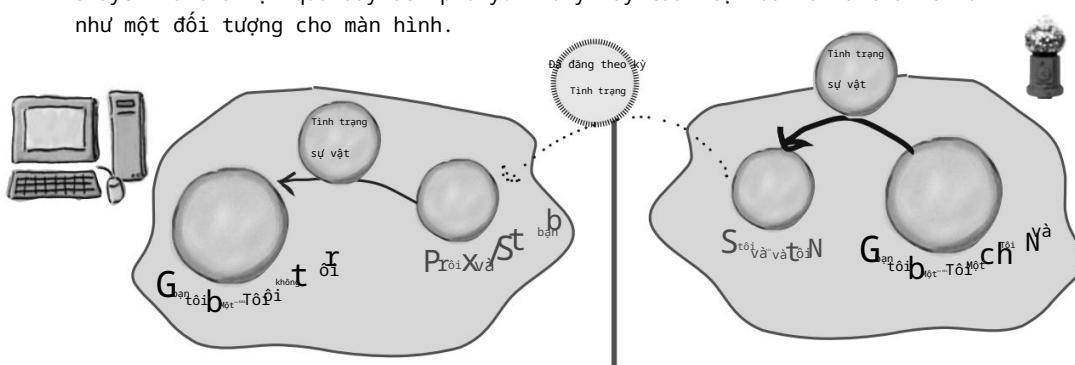
mẫu proxy

- ② `getState()` được gọi trên proxy, chuyển tiếp cuộc gọi đến dịch vụ từ xa.

Bộ khung nhận được yêu cầu và sau đó chuyển tiếp đến máy gumball.



- ③ GumballMachine trả lại trạng thái cho bộ xương, bộ xương này sẽ tuần tự hóa trạng thái đó và chuyển nó trở lại qua dây đến proxy. Proxy hủy tuần tự hóa nó và trả về nó như một đối tượng cho màn hình.



Màn hình không thay đổi gì cả, ngoại trừ việc nó biết rằng nó có thể gặp phải các ngoại lệ từ xa. Nó cũng sử dụng giao diện GumballMachineRemote thay vì một triển khai cụ thể.

Tương tự như vậy, GumballMachine triển khai một giao diện khác và có thể đưa ra một ngoại lệ từ xa trong hàm tạo của nó, nhưng ngoài ra, mã không thay đổi gì.

Chúng tôi cũng có một đoạn mã nhỏ để đăng ký và định vị stub bằng cách sử dụng sổ đăng ký RMI. Nhưng dù thế nào đi nữa, nếu chúng tôi viết thứ gì đó để hoạt động qua Internet, chúng tôi sẽ cần một số loại dịch vụ định vị.

mẫu proxy được xác định

Mẫu Proxy được định nghĩa

Chúng tôi đã bỏ qua rất nhiều trang trong chương này; như bạn có thể thấy, việc giải thích về Proxy từ xa khá phức tạp. Mặc dù vậy, bạn sẽ thấy rằng định nghĩa và sơ đồ lớp cho Proxy Pattern thực sự khá đơn giản. Lưu ý rằng Remote Proxy là một triển khai của Proxy Pattern chung; thực tế có khá nhiều biến thể của mẫu này và chúng ta sẽ nói về chúng sau.

Bây giờ, chúng ta hãy cùng tìm hiểu chi tiết về mô hình chung.

Sau đây là định nghĩa về Mẫu Proxy:

Mẫu Proxy cung cấp đối tượng thay thế hoặc đối tượng giữ chỗ cho đối tượng khác để kiểm soát quyền truy cập vào đối tượng đó.

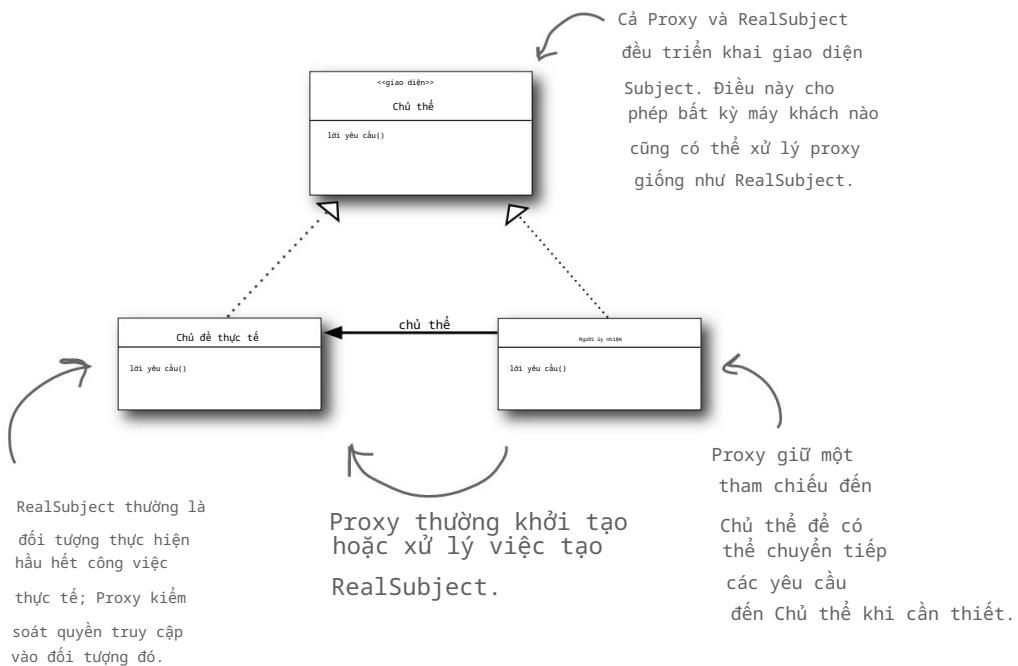
Vâng, chúng ta đã thấy Proxy Pattern cung cấp một đại diện hoặc trình giữ chỗ cho một đối tượng khác. Chúng ta cũng đã mô tả proxy là "đại diện" cho một đối tượng khác.

Nhưng còn proxy kiểm soát quyền truy cập thì sao? Nghe có vẻ hơi lạ. Đừng lo. Trong trường hợp của máy bán kẹo cao su, chỉ cần nghĩ đến proxy kiểm soát quyền truy cập vào đối tượng từ xa. Proxy cần kiểm soát quyền truy cập vì máy khách của chúng ta, màn hình, không biết cách giao tiếp với đối tượng từ xa. Vì vậy, theo một nghĩa nào đó, proxy từ xa kiểm soát quyền truy cập để có thể xử lý thông tin chi tiết về mạng cho chúng ta. Như chúng ta vừa thảo luận, có nhiều biến thể của Mẫu Proxy và các biến thể này thường xoay quanh cách proxy "kiểm soát quyền truy cập". Chúng ta sẽ nói thêm về điều này sau, nhưng hiện tại, đây là một số cách proxy kiểm soát quyền truy cập:

- β Như chúng ta đã biết, proxy từ xa kiểm soát quyền truy cập vào một đối tượng từ xa.
- β Một proxy ảo kiểm soát quyền truy cập vào một tài nguyên tồn kém để tạo ra.
- β Proxy bảo vệ kiểm soát quyền truy cập vào tài nguyên dựa trên quyền truy cập.

Bây giờ bạn đã nắm được cốt lõi của mô hình chung, hãy xem sơ đồ lớp...

Sử dụng Mẫu
Proxy để tạo một
đối tượng đại diện
kiểm soát quyền
truy cập vào một
đối tượng khác, có
thể ở xa, tồn kém để
tạo hoặc cần được bảo mật.



Chúng ta hãy cùng xem sơ đồ nhé...

Đầu tiên chúng ta có một Subject, cung cấp giao diện cho RealSubject và Proxy.

Bằng cách triển khai cùng một giao diện, Proxy có thể được thay thế cho RealSubject ở bất kỳ nơi nào nó xuất hiện.

RealSubject là đối tượng thực hiện công việc thực sự. Đây là đối tượng mà Proxy biểu diễn và kiểm soát quyền truy cập.

Proxy giữ tham chiếu đến RealSubject. Trong một số trường hợp, Proxy có thể chịu trách nhiệm tạo và hủy RealSubject. Máy khách tương tác với RealSubject thông qua Proxy. Vì Proxy và RealSubject triển khai cùng một giao diện (Subject), Proxy có thể được thay thế ở bất kỳ nơi nào có thể sử dụng chủ đề. Proxy cũng kiểm soát quyền truy cập vào RealSubject; quyền kiểm soát này có thể cần thiết nếu Subject đang chạy trên máy từ xa, nếu Subject tồn kén để tạo theo một cách nào đó hoặc nếu quyền truy cập vào chủ đề cần được bảo vệ theo một cách nào đó.

Bây giờ bạn đã hiểu về mô hình chung, hãy cùng xem một số cách khác để sử dụng proxy ngoài Proxy từ xa...

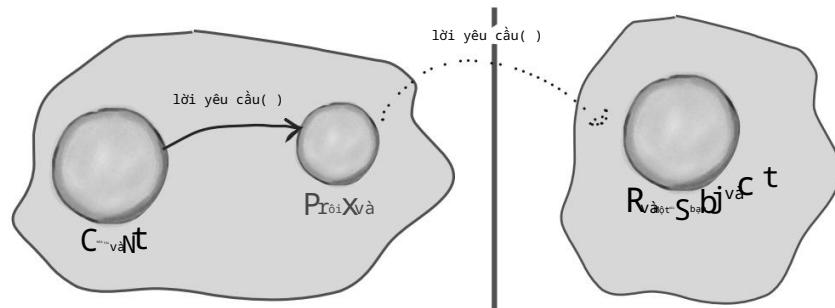
proxy ảo

Chuẩn bị cho Proxy ảo

Được rồi, cho đến giờ bạn đã thấy định nghĩa của Proxy Pattern và bạn đã xem xét một ví dụ cụ thể: Remote Proxy. Vậy giờ chúng ta sẽ xem xét một loại proxy khác, Virtual Proxy. Như bạn sẽ khám phá, Proxy Pattern có thể biểu hiện dưới nhiều hình thức, nhưng tất cả các hình thức đều tuân theo thiết kế proxy chung. Tại sao lại có nhiều hình thức như vậy? Bởi vì proxy pattern có thể được áp dụng cho nhiều trường hợp sử dụng khác nhau. Hãy cùng xem Virtual Proxy và so sánh nó với Remote Proxy:

Proxy từ xa Với

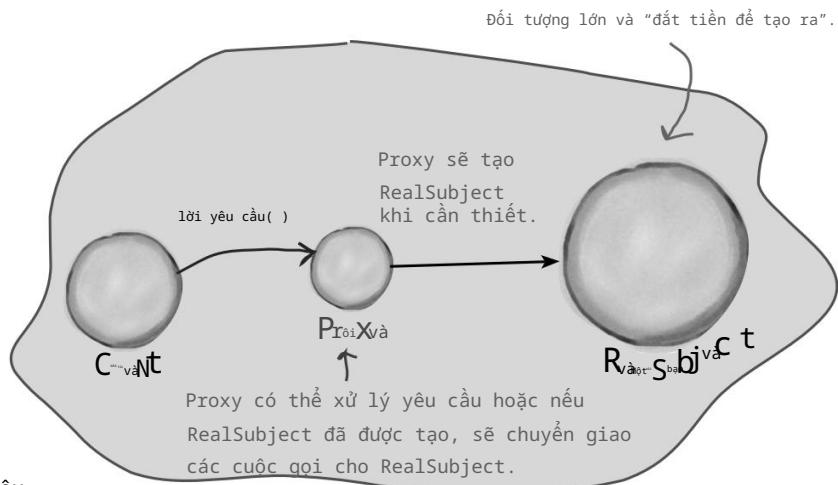
Proxy từ xa, proxy hoạt động như một đại diện cục bộ cho một đối tượng nằm trong một JVM khác. Một lệnh gọi phương thức trên proxy dẫn đến lệnh gọi được chuyển qua dây, được gọi từ xa và kết quả được trả về proxy rồi đến Client.



Bây giờ chúng ta đã biết
khá rõ sơ đồ này rồi...

Proxy ảo

Proxy ảo hoạt động như một đại diện cho một đối tượng có thể tồn kém để tạo ra. Proxy ảo thường trì hoãn việc tạo ra đối tượng cho đến khi cần thiết; Proxy ảo cũng hoạt động như một đại diện cho đối tượng trước và trong khi nó đang được tạo ra. Sau đó, proxy ủy quyền các yêu cầu trực tiếp cho RealSubject.

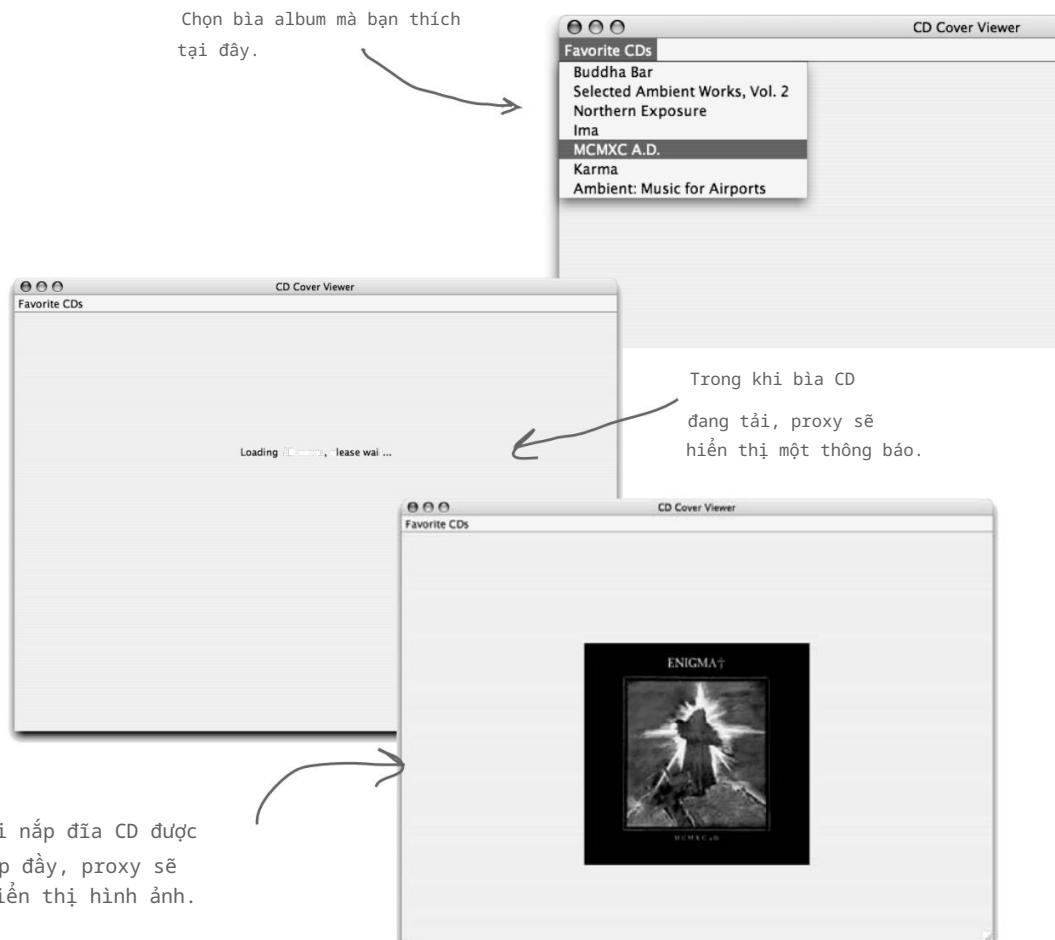


mẫu proxy

Hiển thị bìa CD

Giả sử bạn muốn viết một ứng dụng hiển thị bìa đĩa CD yêu thích của bạn. Bạn có thể tạo một menu các tiêu đề CD và sau đó lấy hình ảnh từ một dịch vụ trực tuyến như Amazon.com. Nếu bạn đang sử dụng Swing, bạn có thể tạo một Biểu tượng và yêu cầu nó tải hình ảnh từ mạng. Vấn đề duy nhất là, tùy thuộc vào tải mạng và băng thông kết nối của bạn, việc lấy bìa CD có thể mất một chút thời gian, vì vậy ứng dụng của bạn phải hiển thị một cái gì đó trong khi bạn đang chờ hình ảnh tải. Chúng tôi cũng không muốn treo toàn bộ ứng dụng trong khi nó đang chờ hình ảnh. Sau khi hình ảnh được tải, thông báo sẽ biến mất và bạn sẽ thấy hình ảnh.

Một cách dễ dàng để đạt được điều này là thông qua proxy áo. Proxy áo có thể thay thế biểu tượng, quản lý việc tải nền và trước khi hình ảnh được tải hoàn toàn từ mạng, hiển thị “Đang tải bìa CD, vui lòng đợi...”. Sau khi hình ảnh được tải, proxy sẽ chuyển giao việc hiển thị cho Biểu tượng.



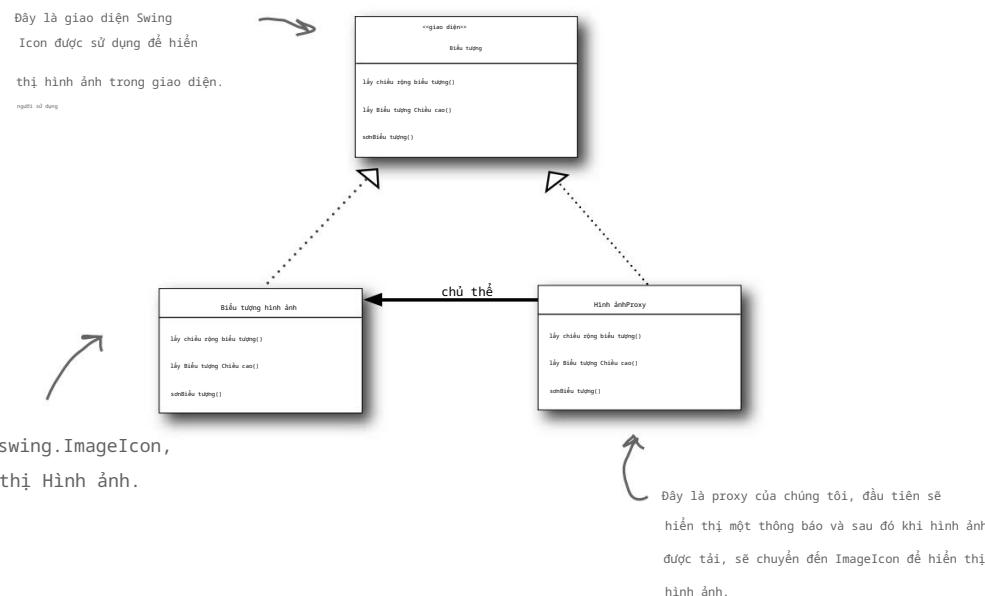
bạn đang ở đây 4 463

proxy hình ảnh kiểm soát truy cập

Thiết kế bìa CD Virtual Proxy

Trước khi viết mã cho CD Cover Viewer, chúng ta hãy xem sơ đồ lớp.

Bạn sẽ thấy nó trông giống hệt lớp Remote Proxy của chúng ta, nhưng ở đây proxy được sử dụng để ẩn một đối tượng tồn kém khi tạo (vì chúng ta cần truy xuất dữ liệu cho Biểu tượng qua mạng) thay vì một đối tượng thực sự nằm ở đâu đó khác trên mạng.



ImageProxy hoạt động như thế nào:

- ➊ Đầu tiên, ImageProxy tạo một ImageIcon và bắt đầu tải nó từ URL mạng.
- ➋ Trong khi các byte của hình ảnh đang được lấy, ImageProxy hiển thị "Đang tải bìa CD, vui lòng đợi...".
- ➌ Khi hình ảnh được tải đầy đủ, ImageProxy sẽ chuyển tất cả các lệnh gọi phương thức đến biểu tượng hình ảnh, bao gồm paintIcon(), getWidth() và getHeight().
- ➍ Nếu người dùng yêu cầu một hình ảnh mới, chúng tôi sẽ tạo một proxy mới và bắt đầu lại quy trình.

mẫu proxy

Viết Proxy hình ảnh

```
lớp ImageProxy thực hiện Icon {
    Biểu tượng hình ảnh;
    URL hình ảnhURL;
    Truy xuất luồngThread; boolean truy
    xuất = false;
```

```
công khai ImageProxy(URL url) { imageURL = url; }
```

```
công khai int getIconWidth() {
    nếu (imageIcon != null) { trả về
        imageIcon.getIconWidth(); } nếu không { trả về 800;
```

```
}
```

```
công khai int getIconHeight() { nếu (imageIcon !
    = null) { trả về
        imageIcon.getIconHeight(); } nếu không { trả về 600;
```

```
}
```

```
public void paintIcon(Thành phần cuối cùng c, Đồ họa g, int x, int y) {
    if (imageIcon != null)
        { imageIcon.paintIcon(c, g, x, y); } else

    { g.drawString("Đang tải bìa CD, vui lòng đợi...", x+300, y+190); if (!retrieving) { retrieving = true;
        retrievalThread = new
            Thread(new Runnable()
            { public void run() { try { imageIcon = new ImageIcon(imageURL, "Bìa
                CD"); c.repaint();

                } catch (Ngoại lệ e) {
                    e.printStackTrace();
                }
            });
        retrievalThread.start();
    }
}
```

ImageProxy
triển khai giao
diện Icon.



imageIcon là biểu tượng THỰC SỰ mà cuối cùng chúng ta muốn hiển thị khi nó được tải.

↑

Chúng ta truyền URL của hình ảnh vào hàm tạo. Đây là hình ảnh chúng ta cần hiển thị sau khi tải xong!

↑
↙

Chúng tôi trả về chiều rộng và chiều cao mặc định cho đến khi imageIcon được tải; sau đó chúng tôi chuyển nó sang imageIcon.

↗

Đây chính là lúc mọi chuyện trở nên thú vị. Mã này vẽ biểu tượng trên màn hình (bằng cách ủy quyền cho imageIcon). Tuy nhiên, nếu chúng ta không có ImageIcon được tạo đầy đủ, thì chúng ta sẽ tạo một cái. Hãy xem xét kỹ hơn ở trang tiếp theo...

```
}
```

hình ảnh proxy cận cảnh



. Mã Gắn

Phương pháp này được gọi khi đến lúc vẽ biểu tượng trên màn hình.

```
public void paintIcon(Thành phần cuối cùng c, Đồ họa g, int x, int y) {
    nếu (imageIcon != null) {
```

```
        imageIcon.paintIcon(c, g, x, y);
```

```
} khác {
```

```
    g.drawString("Đang tải bìa CD, vui lòng đợi...", x+300, y+190);
    nếu (! đang truy xuất) {
```

Nếu đã có biểu tượng, chúng ta hãy bảo nó tự vẽ chính nó.

```
        đang truy xuất = đúng;
        retrievalThread = new Thread(new Runnable() {
            công khai void run() {
                thử {
                    imageIcon = new ImageIcon(imageURL, "Bìa CD");
                    c. sơn lại();
                } catch (Ngoại lệ e) {
                    e.printStackTrace();
                }
            }
        });
    }
```

Nếu không, chúng tôi sẽ hiển thị thông báo "đang tải".

Đây là nơi chúng ta tải hình ảnh biểu tượng THỰC. Lưu ý rằng việc tải hình ảnh bằng ImageIcon là đồng bộ: hàm tạo ImageIcon không trả về cho đến khi hình ảnh được tải. Điều đó không cho chúng ta nhiều cơ hội để thực hiện cập nhật màn hình và hiển thị thông báo của mình, vì vậy chúng ta sẽ sẽ thực hiện điều này một cách không đồng bộ. Xem "Code Way Up "Đóng" ở trang tiếp theo để biết thêm..."



Mã Cách Gần Hơn

Nếu chúng ta chưa có găng lấy lại hình ảnh...

```

nếu (! đang truy xuất) {
    đang truy xuất = đúng;

    retrievalThread = new Thread(new Runnable() {
        công khai void run() {
            thử {
                imageIcon = new ImageIcon(imageURL, "Bìa CD");
                c. sơn lại();
            } catch (Ngoại lệ e) {
                e.printStackTrace();
            }
        }
    });
    retrievalThread.start();
}

```

...sau đó là lúc bắt đầu truy xuất nó (trong trường hợp bạn thắc mắc, chỉ có một luồng gọi paint, vì vậy chúng ta sẽ ổn về mặt an toàn của luồng).

Chúng tôi không muốn treo toàn bộ giao diện người dùng, vì vậy chúng tôi sẽ sử dụng một luồng khác để lấy lại hình ảnh.

Trong luồng của chúng ta, chúng ta khởi tạo đối tượng Icon. Hàm tạo của nó sẽ không trả về cho đến khi hình ảnh được tải.

Khi đã có hình ảnh, chúng ta sẽ nói với Swing rằng chúng ta cần phải sơn lại.

Vì vậy, lần tiếp theo khi màn hình được vẽ sau khi ImageIcon được khởi tạo, phương thức paintIcon sẽ vẽ hình ảnh chứ không phải thông báo đang tải.

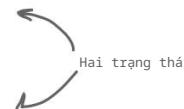
thiết kế câu đố

Thiết kế câu đố

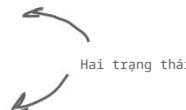
Lớp ImageProxy dường như có hai trạng thái được điều khiển bởi các câu lệnh có điều kiện. Bạn có thể nghĩ ra một mẫu khác có thể đơn giản hơn không? Bạn sẽ thiết kế lại ImageProxy như thế nào?

```
lớp ImageProxy triển khai Icon { // biến thể hiện & hàm  
    tạo ở đây
```

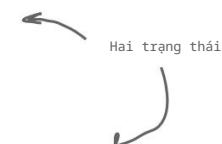
```
công khai int getIconWidth() {  
    nếu (imageIcon != null) { trả về  
        imageIcon.getIconWidth(); } nếu không { trả về 800;  
    }  
}
```



```
công khai int getIconHeight() { nếu (imageIcon !  
    = null) { trả về  
        imageIcon.getIconHeight(); } nếu không { trả về 600;  
    }  
}
```



```
public void paintIcon(Thành phần cuối cùng c, Đồ họa g, int x, int y) {  
    if (imageIcon != null)  
        { imageIcon.paintIcon(c, g, x, y); } else  
  
        { g.drawString("Đang tải bìa CD, vui lòng đợi...", x+300, y+190); // thêm mã ở đây  
    }  
}
```



Kiểm tra Trình xem bìa CD



Sẵn sàng nướng
Mã số

Được rồi, đã đến lúc thử nghiệm proxy ảo mới lạ này. Đằng sau hậu trường, chúng tôi đã tạo ra một ImageProxyTestDrive mới để thiết lập cửa sổ, tạo khung, cài đặt menu và tạo proxy của chúng tôi.

Chúng tôi không đi sâu vào tất cả các mã đó ở đây, nhưng bạn luôn có thể lấy mã nguồn và xem thử hoặc kiểm tra ở cuối chương, nơi chúng tôi liệt kê tất cả mã nguồn cho Proxy ảo.

Sau đây là một phần của mã lệnh lái thử:

```
lớp công khai ImageProxyTestDrive {
    Thành phần hình ảnh Thành phần hình ảnh;
    public static void main (String[] args) ném Ngoại lệ {
        ImageProxyTestDrive testDrive = new ImageProxyTestDrive();
    }

    public ImageProxyTestDrive() ném ra Exception{
        // thiết lập khung và menu

        Biểu tượng icon = new ImageProxy(initialURL);
        imageComponent = new ImageComponent(biểu tượng);
        frame.getContentPane().add(imageComponent);
    }
}
```

Cuối cùng chúng ta thêm proxy vào khung để có thể hiển thị.

Bây giờ chúng ta hãy chạy thử nghiệm:

```
Cửa sổ chính sửa tên Trợ giúp JustSomeOfTheCDsThatGotUsThroughThisBook
% java ImageProxyTestDrive
```

Chạy ImageProxyTestDrive sẽ cho bạn một cửa sổ như thế này.

Những điều cần thử...

- 1 Sử dụng menu để tải các bìa CD khác nhau; xem màn hình proxy hiển thị "đang tải" cho đến khi hình ảnh xuất hiện.
- 2 Thay đổi kích thước cửa sổ khi thông báo "đang tải" được hiển thị. Lưu ý rằng proxy đang xử lý việc tải mà không treo cửa sổ Swing.
- 3 Thêm đĩa CD yêu thích của bạn vào ImageProxyTestDrive.



Ở đây chúng ta tạo một proxy hình ảnh và đặt nó thành một URL ban đầu. Bất cứ khi nào bạn chọn một lựa chọn từ CD, bạn sẽ nhận được một proxy hình ảnh mới.

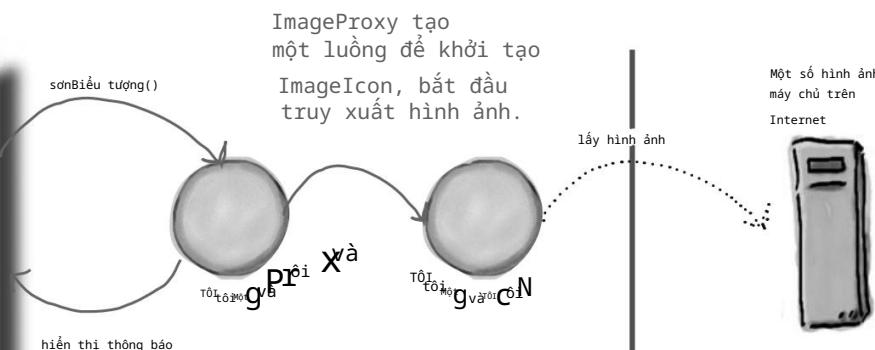
thực đơn,

Tiếp theo, chúng ta gói proxy của mình trong một thành phần để có thể thêm vào khung. Thành phần này sẽ xử lý chiều rộng, chiều cao và các chi tiết tương tự của proxy.

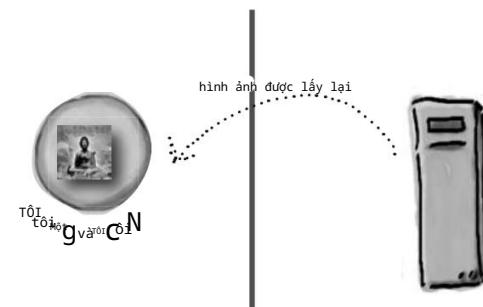
hậu trường với proxy hình ảnh

Chúng tôi đã làm gì?

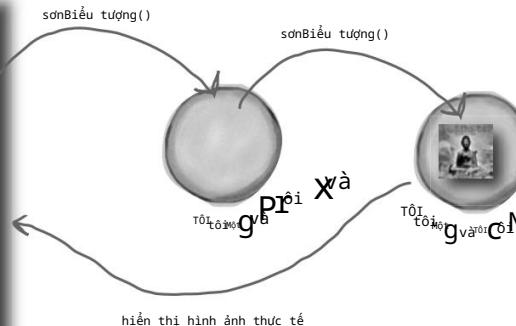
- Chúng tôi đã tạo một ImageProxy để hiển thị. Phương thức paintIcon() được gọi và ImageProxy sẽ khởi chạy một luồng để lấy hình ảnh và tạo ImageIcon.



- Đến một thời điểm nào đó, hình ảnh sẽ được trả về và ImageIcon được khởi tạo hoàn toàn.



- Sau khi ImageIcon được tạo, lần tiếp theo khi paintIcon() được gọi, proxy sẽ chuyển tiếp đến ImageIcon.



không có Những câu hỏi ngắn

Q: Proxy từ xa và ào

Với tôi, Proxy có vẻ rất khác biệt; liệu chúng có thực sự là MỘT mẫu không?

A: Bạn sẽ tìm thấy rất nhiều biến thể của Mẫu Proxy trong thế giới thực; điểm chung của chúng là chặn lời gọi phương thức mà máy khách đang thực hiện trên đối tượng.

Mức độ gián tiếp này cho phép chúng ta thực hiện nhiều việc, bao gồm gửi yêu cầu đến một chủ thể từ xa, cung cấp một đại diện cho một đối tượng đất tiền khi nó được tạo ra hoặc, như bạn sẽ thấy, cung cấp một số mức độ bảo vệ có thể xác định những máy khách nào nên được gọi phương thức nào. Đó chỉ là khởi đầu; Proxy Pattern chung có thể được áp dụng theo nhiều cách khác nhau và chúng tôi sẽ đề cập đến một số cách khác vào cuối chương.

Q: ImageProxy có vẻ giống như một Decorator đối với tôi. Ý tôi là, về cơ bản chúng ta đang gói một đối tượng bằng một đối tượng khác và sau đó chuyển giao các lệnh gọi cho ImageIcon. Tôi đang bỗng nhiên?

A: Đôi khi Proxy và Decorator

trong rất giống nhau, nhưng mục đích của chúng thì khác nhau: một trình trang trí thêm hành vi vào một lớp, trong khi một proxy kiểm soát quyền truy cập vào lớp đó. Bạn có thể nói, "Thông báo tài không phải là thêm hành vi sao?" Trong một số

cách nó là; tuy nhiên, quan trọng hơn, ImageProxy đang kiểm soát quyền truy cập vào ImageIcon. Nó kiểm soát quyền truy cập như thế nào? Vâng, hãy nghĩ về nó theo cách này: proxy đang tách máy khách khỏi ImageIcon. Nếu chúng được ghép nối, máy khách sẽ phải đợi cho đến khi từng hình ảnh được truy xuất trước khi nó có thể vẽ toàn bộ giao diện của nó. Proxy kiểm soát quyền truy cập vào ImageIcon để trước khi nó được tạo hoàn toàn, proxy cung cấp một biểu diễn khác trên màn hình.

Sau khi ImageIcon được tạo, proxy sẽ cho phép truy cập.

Q: Làm thế nào để tôi khiến khách hàng sử dụng

Người đại diện chứ không phải là chủ thể thực sự?

A: Câu hỏi hay. Một câu hỏi phổ biến

Kỹ thuật này là cung cấp một nhà máy có thể khởi tạo và trả về chủ thể. Vì điều này xảy ra trong phương thức nhà máy, chúng ta có thể bao bọc chủ thể bằng proxy trước khi trả về. Máy khách không bao giờ biết hoặc quan tâm rằng nó đang sử dụng proxy thay vì thứ thực sự.

Q: Tôi nhận thấy trong ImageProxy

Ví dụ, bạn luôn tạo một ImageIcon mới để lấy hình ảnh, ngay cả khi hình ảnh đã được truy xuất. Bạn có thể triển khai một cái gì đó tương tự như ImageProxy để lưu trữ các lần truy xuất trước đó không?

A: Bạn đang nói về một điều đặc biệt-

dạng được mã hóa của Proxy ào được gọi là Proxy lưu trữ đệm. Proxy lưu trữ đệm duy trì bộ nhớ đệm của các đối tượng đã tạo trước đó và khi có yêu cầu, nó sẽ trả về đối tượng đã lưu trữ đệm, nếu có thể.

Chúng ta sẽ xem xét điều này và một số biến thể khác của Mẫu Proxy ở cuối chương.

H: Tôi thấy Decorator và Proxy

liên quan, nhưng còn Adapter thì sao? Một adapter cũng có vẻ rất giống.

A: Cả Proxy và Adapter đều nằm trong

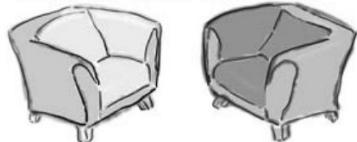
phía trước các đối tượng khác và chuyển tiếp yêu cầu đến chúng. Hãy nhớ rằng Adapter thay đổi giao diện của các đối tượng mà nó điều chỉnh, trong khi Proxy triển khai cùng một giao diện.

Có một điểm tương đồng bổ sung liên quan đến Protection Proxy. Protection Proxy có thể cho phép hoặc không cho phép máy khách truy cập vào các phương thức cụ thể trong một đối tượng dựa trên vai trò của máy khách. Theo cách này, Protection Proxy chỉ có thể cung cấp một giao diện một phần cho máy khách, khá giống với một số Adapter. Chúng ta sẽ xem xét Protection Proxy trong một vài trang.

trò chuyện bên lò sưởi: người đại diện và người trang trí

Fireside Chats

Bài nói chuyện tôi nay: Proxy và Decorator có chủ đích.



Người ủy nhiệm

Xin chào, Decorator. Tôi cho rằng bạn ở đây vì đôi khi
mọi người khiến chúng tôi bối rối?

Tôi sao chép ý tưởng của bạn à? Làm ơn. Tôi kiểm soát quyền
truy cập vào các đồ vật. Bạn chỉ cần trang trí chúng. Công
việc của tôi quan trọng hơn công việc của bạn rất nhiều, thậm
chí không hề buồn cười.

Được thôi, có lẽ bạn không hoàn toàn phù phiếm... nhưng tôi
vẫn không hiểu tại sao bạn lại nghĩ tôi sao chép tất cả ý
tưởng của bạn. Tôi chỉ muốn đại diện cho chủ thể của mình,
không phải trang trí chúng.

Tôi không nghĩ là bạn hiểu đâu, Decorator. Tôi đứng ra thay thế
cho Subjects của tôi; tôi không chỉ thêm hành vi. Khách hàng
sử dụng tôi như một người thay thế cho một Real Subject, vì tôi
có thể bảo vệ họ khỏi sự truy cập không mong muốn, hoặc giữ cho GUI
của họ không bị treo trong khi họ đang chờ các đối tượng lớn tải,
hoặc ẩn sự thật rằng Subjects của họ đang chạy trên các máy từ
xa. Tôi cho rằng đó là một ý định rất khác so với bạn!

Người trang trí

Vâng, tôi nghĩ lý do khiến mọi người nhầm lẫn là vì bạn đi khắp
nơi giả vờ là một mẫu hoàn toàn khác, trong khi thực tế,
bạn chỉ là một Decorator trá hình. Tôi thực sự không nghĩ bạn
nên sao chép tất cả các ý tưởng của tôi.

"Chỉ" trang trí thôi sao? Bạn nghĩ trang trí là một kiểu mẫu
phù phiếm không quan trọng sao? Để tôi nói cho bạn biết,
tôi thêm hành vi. Đó là điều quan trọng nhất về đồ vật -
chúng làm gì!

Bạn có thể gọi nó là "biểu diễn" nhưng nếu nó trông giống như
một con vịt và dì như một con vịt... Ý tôi là, hãy nhìn vào Proxy
ảo của bạn; nó chỉ là một cách khác để thêm hành vi để thực
hiện một cái gì đó trong khi một số đối tượng lớn đắt tiền đang
tải, và Proxy từ xa của bạn là một cách để nói chuyện với các
đối tượng từ xa để các máy khách của bạn không phải bận tâm
đến điều đó. Tất cả là về hành vi, giống như tôi đã nói.

Gọi nó là gì tùy bạn. Tôi triển khai cùng một giao diện như
các đối tượng tôi gói; bạn cũng vậy.

mẫu proxy

Người ủy nhiệm

Được rồi, chúng ta hãy xem lại tuyên bố đó. Bạn gói một đối tượng. Mặc dù đôi khi chúng ta nói một cách không chính thức rằng một proxy gói Chủ thể của nó, nhưng thực ra đó không phải là thuật ngữ chính xác.

Người trang trí

Ô vây sao? Tại sao không?

Hãy nghĩ về một proxy từ xa... tôi đang gói đối tượng nào? Đối tượng tôi đang biểu diễn và kiểm soát quyền truy cập nằm trên một máy khác!

Chúng ta hãy cùng xem bạn làm điều đó nhé.

Được thôi, nhưng chúng ta đều biết proxy từ xa khá kỳ lạ. Bạn có ví dụ thứ hai không? Tôi nghĩ ngờ điều đó.

Được thôi, hãy lấy một proxy ào... hãy nghĩ về ví dụ về trình xem CD. Khi máy khách lần đầu tiên sử dụng tôi làm proxy thì chủ thẻ thậm chí không tồn tại! Vậy tôi đang gói cái gì ở đó?

Ừ, và điều tiếp theo bạn sẽ nói là bạn thực sự có thể tạo ra các đối tượng.

Tôi không bao giờ biết rằng các trình trang trí lại ngốc nghếch đến vậy! Tất nhiên đôi khi tôi tạo ra các đối tượng, bạn nghĩ một proxy ào lấy chủ thẻ của nó như thế nào! Được rồi, bạn vừa chỉ ra một sự khác biệt lớn giữa chúng ta: cả hai chúng ta đều biết các trình trang trí chỉ thêm vào phần trang trí cửa sổ; chúng không bao giờ có thể khởi tạo bất cứ thứ gì.

Ô thế à? Hãy thực hiện điều này!

Này, sau cuộc nói chuyện này tôi tin rằng anh chỉ là một kẻ úy nhiệm ngu ngốc thôi!

Proxy ngớ ngẩn ư? Tôi muốn thấy bạn có thể gói một đối tượng theo cách đệ quy bằng 10 trình trang trí và đồng thời vẫn giữ được đầu óc minh mẫn.

Rất hiếm khi bạn thấy một proxy bao bọc một chủ đề nhiều lần; trên thực tế, nếu bạn bao bọc một thứ gì đó 10 lần, tốt hơn hết bạn nên xem xét lại thiết kế của mình.

Giống như một proxy, hành động như thật khi thực tế bạn chỉ đứng thay cho các đối tượng đang thực hiện công việc thực sự. Bạn biết không, thực ra tôi thấy tiếc cho bạn.

bạn đang ở đây 4 473

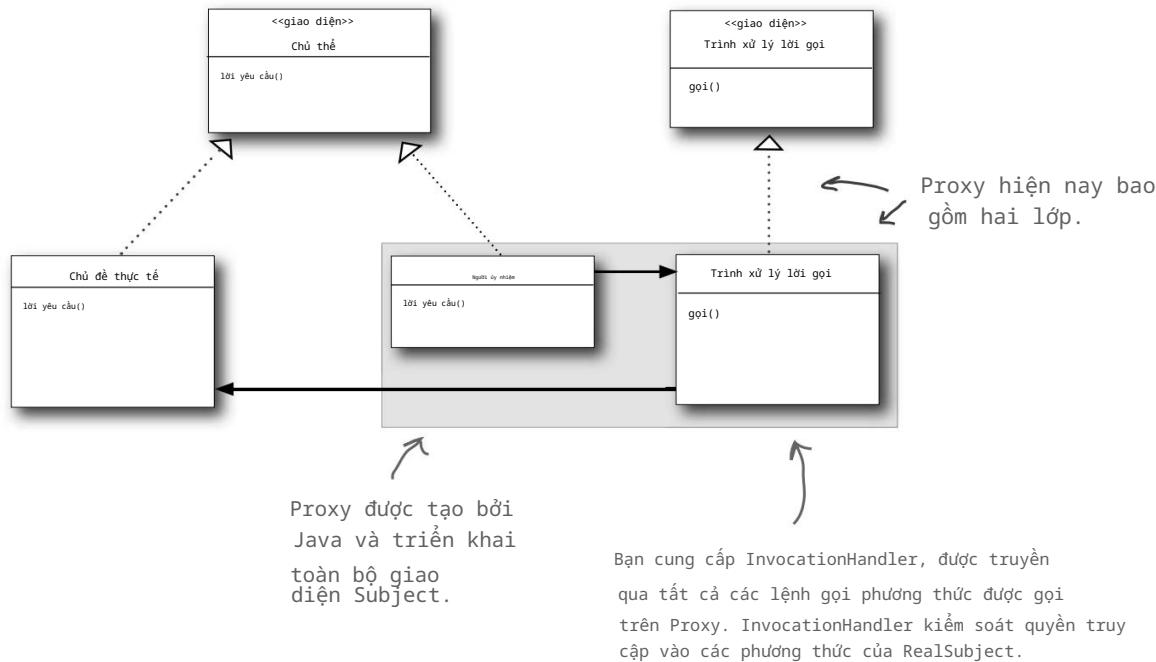
proxy bảo vệ

Sử dụng Proxy của Java API để tạo proxy bảo vệ



Java có hỗ trợ proxy riêng ngay trong gói `java.lang.reflect`. Với gói này, Java cho phép bạn tạo một lớp proxy ngay lập tức để triển khai một hoặc nhiều giao diện và chuyển tiếp các lệnh gọi phương thức đến một lớp mà bạn chỉ định. Vì lớp proxy thực tế được tạo khi chạy, chúng tôi gọi công nghệ Java này là proxy động.

Chúng ta sẽ sử dụng proxy động của Java để tạo triển khai proxy tiếp theo (proxy bảo vệ), nhưng trước khi thực hiện, hãy nhanh chóng xem sơ đồ lớp cho thấy cách các proxy động được kết hợp với nhau. Giống như hầu hết mọi thứ trong thế giới thực, nó hơi khác so với định nghĩa cổ điển của mẫu:



Vì Java tạo lớp Proxy cho bạn, bạn cần một cách để cho lớp Proxy biết phải làm gì. Bạn không thể đưa mã đó vào lớp Proxy như chúng ta đã làm trước đây, vì bạn không triển khai trực tiếp một lớp. Vì vậy, nếu bạn không thể đưa mã này vào lớp Proxy, bạn sẽ đặt nó ở đâu? Trong `InvocationHandler`. Công việc của `InvocationHandler` là phản hồi bất kỳ lệnh gọi phương thức nào trên proxy. Hãy nghĩ về `InvocationHandler` như đối tượng mà Proxy yêu cầu thực hiện tất cả công việc thực sự sau khi nhận được lệnh gọi phương thức.

Được rồi, chúng ta hãy cùng tìm hiểu cách sử dụng proxy động...

mẫu proxy

Ghép đôi ở Objectville



Mọi thị trấn đều cần một dịch vụ mai mối, đúng không? Bạn đã thực hiện nhiệm vụ và triển khai dịch vụ hẹn hò cho Objectville. Bạn cũng đã cố gắng sáng tạo bằng cách đưa tính năng "Hot or Not" vào dịch vụ, nơi những người tham gia có thể đánh giá lẫn nhau - bạn cho rằng điều này giúp khách hàng của bạn luôn tham gia và tìm kiếm những đôi tượng phù hợp; nó cũng làm cho mọi thứ trở nên thú vị hơn rất nhiều.

Dịch vụ của bạn xoay quanh bean Person cho phép bạn thiết lập và nhận thông tin về một người:

Đây là giao diện; chúng

ta sẽ thực hiện

chỉ trong một giây...

giao diện công khai PersonBean {

```
Chuỗi getName();
Chuỗi getGender();
Chuỗi getInterests();
int getHotOrNotRating();
```

```
void setName(Chuỗi tên);
void setGender(Chuỗi giới tính);
void setInterests(Chuỗi sở thích);
void setHotOrNotRating(int rating);
```

}

Chúng ta cũng có thể thiết lập
thông tin tương tự thông qua
các lệnh gọi phương thức tương ứng.

Tại đây chúng ta có thể lấy

thông tin về tên, giới
tính, sở thích và xếp

hạng HotOrNot của người đó (1-10).

setHotOrNotRating() lấy
một số nguyên và thêm vào số
trung bình đang chạy của người này.

Bây giờ chúng ta hãy kiểm tra việc thực hiện...

personbean cần được bảo vệ

Việc triển khai PersonBean

PersonBeanImpl triển khai giao diện PersonBean



lớp công khai PersonBeanImpl triển khai PersonBean {

```

    Tên chuỗi;
    Chuỗi giới tính;
    Lãi suất chuỗi;
    Xếp hạng int;
    int ratingCount = 0;
```



Các biến thể hiện.

```

    công khai String getName() {
        trả về tên;
    }
```



Tất cả các phương thức getter; mỗi phương

thức đều trả về biến thể hiện thích hợp...

```

    công khai String getGender() {
        trả về giới tính;
    }
```

```

    công khai String getInterests() {
        trả lại lãi suất;
    }
```

...ngoại trừ

```

    công khai int getHotOrNotRating() {
        nếu (ratingCount == 0) trả về 0;
        trả về (đánh giá/danh giáCount);
    }
```



getHotOrNotRating(), tính toán
mức xếp hạng trung bình bằng
cách chia xếp hạng cho ratingCount.

```

    public void setName(String name) {
        this.name = tên;
    }
```



Và đây là tất cả các phương
thức thiết lập, dùng để
thiết lập biến thể hiện tương ứng.

```

    public void setGender(String giới tính) {
        this.gender = giới tính;
    }
```

```

    public void setInterests(Chuỗi sở thích) {
        this.interests = sở thích;
    }
```

```

    công khai void setHotOrNotRating(int rating) {
        this.rating += rating; ratingCount+
        +;
    }
}
```



Cuối cùng,
phương thức
setHotOrNotRating() tăng
tổng ratingCount và
thêm xếp hạng vào tổng đang chạy.

Tôi không thành công lắm trong việc tìm kiếm ngày. Sau đó, tôi nhận thấy có người đã thay đổi sở thích của tôi. Tôi cũng nhận thấy rằng rất nhiều người đang tăng điểm HotOrNot của họ bằng cách tự cho mình xếp hạng cao. Bạn không nên có thể thay đổi sở thích của người khác hoặc tự cho mình xếp hạng!



Trong khi chúng tôi nghĩ ngờ các yếu tố khác có thể ngăn Elroy nhận được ngày, anh ấy nói đúng: bạn không thể tự bỏ phiếu hoặc thay đổi dữ liệu của khách hàng khác. Theo cách PersonBean của chúng tôi được định nghĩa, bất kỳ khách hàng nào cũng có thể gọi bất kỳ phương thức nào.

Đây là một ví dụ hoàn hảo về nơi chúng ta có thể sử dụng Proxy bảo vệ. Proxy bảo vệ là gì? Đó là một proxy kiểm soát quyền truy cập vào một đối tượng dựa trên quyền truy cập. Ví dụ, nếu chúng ta có một đối tượng nhân viên, proxy bảo vệ có thể cho phép nhân viên gọi một số phương thức nhất định trên đối tượng, người quản lý gọi các phương thức bổ sung (như setSalary()) và nhân viên nhân sự gọi bất kỳ phương thức nào trên đối tượng.

Trong dịch vụ hẹn hò của chúng tôi, chúng tôi muốn đảm bảo rằng khách hàng có thể thiết lập thông tin của riêng mình trong khi ngăn chặn người khác thay đổi thông tin đó. Chúng tôi cũng muốn cho phép điều ngược lại với xếp hạng HotOrNot: chúng tôi muốn những khách hàng khác có thể thiết lập xếp hạng, nhưng không phải khách hàng cụ thể đó. Chúng tôi cũng có một số phương thức lấy dữ liệu trong PersonBean và vì không có phương thức nào trong số này trả về thông tin riêng tư nên bất kỳ khách hàng nào cũng có thể gọi chúng.

↑

Elroy

kịch năm phút



Kịch năm phút: bảo vệ chủ thẻ

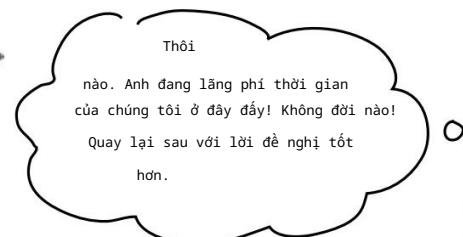
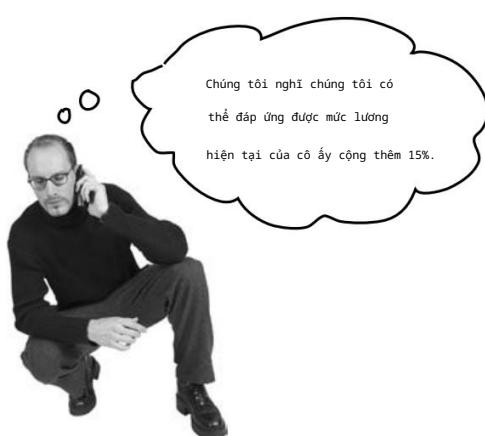
Bong bóng Internet dường như là một ký ức xa vời; đó là những ngày mà tất cả những gì bạn cần làm để tìm một công việc tốt hơn, lương cao hơn là đi bộ qua đường. Ngay cả các đại lý cho các nhà phát triển phần mềm cũng thịnh hành...



Joe DotCom



Giống như một proxy bảo vệ, tác nhân bảo vệ quyền truy cập vào chủ thẻ của mình, chỉ cho phép một số cuộc gọi nhất định đi qua...



mẫu proxy

Bức tranh toàn cảnh: tạo Proxy động cho PersonBean

Chúng tôi có một vài vấn đề cần khắc phục: khách hàng không nên thay đổi xếp hạng HotOrNot của riêng họ và khách hàng không có thể thay đổi thông tin cá nhân của khách hàng khác. Để khắc phục những vấn đề này, chúng tôi sẽ tạo hai proxy: một để truy cập đối tượng PersonBean của riêng bạn và một để truy cập đối tượng PersonBean của khách hàng khác. Theo cách đó, các proxy có thể kiểm soát những yêu cầu có thể được thực hiện trong từng trường hợp.

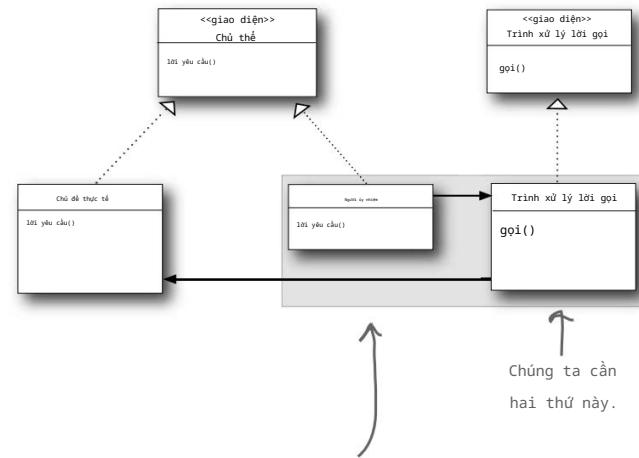
Bạn còn nhớ sơ đồ này
ở vài trang trước không...

Để tạo các proxy này, chúng ta sẽ sử dụng proxy động của Java API mà bạn đã thấy ở một vài trang trước. Java sẽ tạo hai proxy cho chúng ta; tất cả những gì chúng ta cần làm là cung cấp trình xử lý biết phải làm gì khi một phương thức được gọi trên proxy.

Bước một:

Tạo hai InvocationHandler.

InvocationHandlers triển khai hành vi của proxy. Như bạn sẽ thấy, Java sẽ đảm nhiệm việc tạo lớp và đối tượng proxy thực tế, chúng ta chỉ cần cung cấp một trình xử lý biết phải làm gì khi một phương thức được gọi trên đó.



Bước hai:

Viết mã tạo proxy động.

Chúng ta cần viết một chút mã để tạo lớp proxy và khởi tạo nó.

Chúng ta sẽ tìm hiểu từng bước trong đoạn mã này sau.

Chúng tôi tạo ra

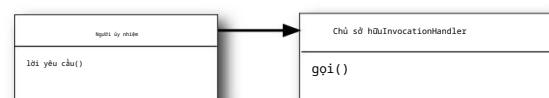
proxy tại
thời gian chạy.

Bước ba:

Bao bọc bất kỳ đối tượng PersonBean nào bằng proxy thích hợp.

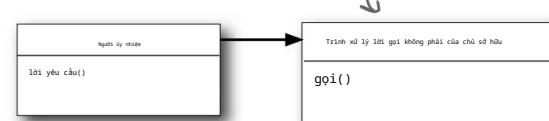
Khi chúng ta cần sử dụng đối tượng PersonBean, thì đối tượng đó có thể là chính khách hàng (trong trường hợp đó, sẽ gọi đối tượng đó là "chủ sở hữu"), hoặc là một người dùng khác của dịch vụ mà khách hàng đang kiểm tra (trong trường hợp đó, chúng ta sẽ gọi đối tượng đó là "không phải chủ sở hữu").

Trong cả hai trường hợp, chúng tôi đều tạo proxy thích hợp cho PersonBean.



Khi khách hàng đang xem hật đậu của mình

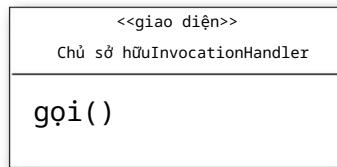
Khi một khách hàng đang xem ai đó đậu của người khác



tạo một trình xử lý lệnh gọi

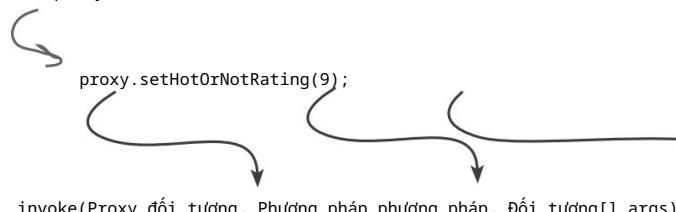
Bước một: tạo Trình xử lý lệnh gọi

Chúng ta biết rằng chúng ta cần viết hai trình xử lý gọi, một cho chủ sở hữu và một cho người không phải chủ sở hữu. Nhưng trình xử lý gọi là gì? Đây là cách để suy nghĩ về chúng: khi một cuộc gọi phương thức được thực hiện trên proxy, proxy sẽ chuyển tiếp cuộc gọi đó đến trình xử lý gọi của bạn, nhưng không phải bằng cách gọi phương thức tương ứng của trình xử lý gọi. Vậy, nó gọi cái gì? Hãy xem giao diện InvocationHandler:



Chỉ có một phương thức, invoke(), và bất kể phương thức nào được gọi trên proxy, phương thức invoke() là phương thức được gọi trên trình xử lý. Hãy cùng xem cách thức hoạt động của nó:

- ① Giả sử phương thức setHotOrNotRating()
được gọi trên proxy.



- ② Sau đó, proxy sẽ
quay lại và gọi
invoke() trên
InvocationHandler.

Lớp Method, một phần của API phản
chiếu, cho chúng ta biết phương
thức nào được gọi trên proxy thông
qua phương thức getName() của nó.

- ③ Trình xử lý quyết định
những gì cần làm với
yêu cầu và có thể
chuyển tiếp yêu
cầu đó đến RealSubject.

Người xử lý quyết
định như thế nào?
Chúng ta sẽ tìm hiểu sau.

phương thức trả về.invoke(person, args);

Ở đây chúng ta gọi
phương thức gốc được gọi
trên proxy. Đối tượng này
được truyền cho chúng ta
trong lệnh gọi invoke.

Bây giờ chúng
ta mới gọi nó
trên RealSubject...

với các lập luận
ban đầu.

Tiếp tục tạo Trình xử lý lời gọi...

Khi invoke() được gọi bởi proxy, làm sao bạn biết phải làm gì với lệnh gọi đó?

Thông thường, bạn sẽ kiểm tra phương thức được gọi trên proxy và đưa ra quyết định dựa trên tên phương thức và có thể là các đối số của phương thức đó. Hãy triển khai OwnerInvocationHandler để xem cách thức hoạt động của nó:

```

nhập java.lang.reflect.*;

lớp công khai OwnerInvocationHandler triển khai InvocationHandler {
    PersonBean người;

    công khai OwnerInvocationHandler(PersonBean người) {
        this.person = người;
    }

    public Object invoke(Object proxy, Method method, Object[] args) throws IllegalAccessException {
        thử {
            nếu (phương thức getName().startsWith("get")) {
                phương thức trả về.invoke(person, args);
            } nếu không thì nếu (phương thức.getName().bằng("setHotOrNotRating")) {
                ném ngoại lệ IllegalAccessException mới();
            } else if (phương thức.getName().startsWith("set")) {
                phương thức trả về.invoke(person, args);
            }
        } bắt (InvocationTargetException e) {
            e.printStackTrace();
        }
    }
}

```

Nếu gọi bất kỳ phương thức nào khác, chúng ta sẽ trả về giá trị null thay vì thử.

Tất cả trình
xử lý lệnh gọi đều
triển khai giao diện
InvocationHandler.

Chúng ta truyền
Chủ ngữ thực vào hàm
tạo và giữ một tham
chiếu đến nó.

Sau đây là phương
thức invoke được gọi
mỗi khi một phương thức
được gọi trên proxy.

Nếu phương thức là một
getter, chúng ta sẽ tiếp
tục và gọi nó trên
chủ đề thực tế.

Ngược lại, nếu đó là phương
thức setHotOrNotRating()
thì chúng ta sẽ không
cho phép bằng cách
đưa ra lỗi IllegalAccessException.

Điều này sẽ xảy ra
nếu chủ thể thực
sự đưa ra ngoại lệ.

Bởi vì chúng ta là
chủ sở hữu nên bắt
kỳ phương pháp thiết lập
nào khác cũng được và
chúng ta sẽ tiếp tục
và áp dụng nó vào
chủ thể thực tế.

tạo trình xử lý lệnh gọi của riêng bạn



Bài tập

NonOwnerInvocationHandler hoạt động giống như
OwnerInvocationHandler ngoại trừ việc nó cho phép gọi đến setHotOrNotRating()
và không cho phép gọi đến bất kỳ phương thức set nào khác. Hãy tiếp tục
và tự viết trình xử lý này:

mẫu proxy

Bước hai: tạo lớp Proxy và khởi tạo đối tượng Proxy

Bây giờ, tất cả những gì chúng ta còn lại là tạo động lớp proxy và khởi tạo đối tượng proxy. Hãy bắt đầu bằng cách viết một phương thức lấy PersonBean và biết cách tạo proxy chủ sở hữu cho nó. Nghĩa là, chúng ta sẽ tạo loại proxy chuyển tiếp các lệnh gọi phương thức của nó đến OwnerInvocationHandler. Đây là mã:

```

Phương pháp này lấy một đối tượng người (chủ
thể thực) và trả về một proxy cho nó. Bởi vì proxy
có cùng giao diện với chủ thể, chúng ta
trả về một PersonBean.

PersonBean getOwnerProxy(PersonBean người) {
    trả về (PersonBean)
        Proxy.newProxyInstance( person.getClass().getClassLoader(),
            người.getClass().getInterfaces(),
            new OwnerInvocationHandler(người));
}

```

Mã này tạo ra proxy.

Đây là một mã khá xấu
xí, vì vậy hãy cùng xem
xét kỹ lưỡng.

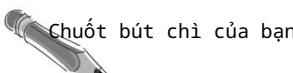
Để tạo proxy, chúng ta sử
dụng phương thức tĩnh
newProxyInstance trên lớp Proxy...

Chúng ta truyền cho nó classloader
cho môn học của chúng ta...

...và tập hợp các giao diện mà
proxy cần triển khai...

...và trình xử lý lệnh gọi, trong
 trường hợp này là OwnerInvocationHandler.

Chúng ta truyền chủ thể thực vào hàm tạo của trình
xử lý lệnh gọi. Nếu bạn xem lại hai trang, bạn
sẽ thấy đây là cách trình xử lý truy cập vào chủ thể
thực.



Mặc dù có chút phức tạp, nhưng không cần quá nhiều thao tác để tạo một proxy động.
Tại sao bạn không viết getNonOwnerProxy(), hàm này trả về một proxy cho
NonOwnerInvocationHandler:

Hãy nói xa hơn: bạn có thể viết một phương thức getProxy() sử dụng một
trình xử lý và một người rồi trả về một proxy sử dụng trình xử lý đó không?

tim thấy sự phù hợp của bạn

Kiểm tra dịch vụ mai mối

Hãy chạy thử dịch vụ ghép đôi và xem cách nó kiểm soát quyền truy cập vào phương thức thiết lập dựa trên proxy được sử dụng.

```

lớp công khai MatchMakingTestDrive {
    // các biến thể hiện ở đây

    public static void main(String[] args) {
        Kiểm tra MatchMakingTestDrive = new MatchMakingTestDrive();
        kiểm tra.ở đĩa();
    }

    công khai MatchMakingTestDrive() {
        khởi tạo cơ sở dữ liệu();
    }

    công khai void drive() {
        PersonBean joe = getPersonFromDatabase("Joe Javabean"); PersonBean ownerProxy =
            getOwnerProxy(joe);
        System.out.println("Tên là " + ownerProxy.getName());
        " ownerProxy.setInterests("bowling, Go");
        System.out.println("Lợi ích được thiết lập từ proxy của chủ sở hữu");
        thử {
            chủ sở hữuProxy.setHotOrNotRating(10);
        } catch (Ngoại lệ e) {
            System.out.println("Không thể thiết lập xếp hạng từ proxy của chủ sở hữu");
        }
        System.out.println("Xếp hạng là " + ownerProxy.getHotOrNotRating());

        PersonBean nonOwnerProxy = getNonOwnerProxy(joe);
        System.out.println("Tên là " + nonOwnerProxy.getName());
        thử {
            nonOwnerProxy.setInterests("bowling, Go");
        } catch (Ngoại lệ e) {
            System.out.println("Không thể thiết lập quyền lợi từ proxy không phải chủ sở hữu");
        }
        nonOwnerProxy.setHotOrNotRating(3);
        System.out.println("Đánh giá được đặt từ proxy không phải chủ sở hữu");
        System.out.println("Xếp hạng là " + nonOwnerProxy.getHotOrNotRating());
    }

    // các phương pháp khác như getOwnerProxy và getNonOwnerProxy ở đây
}

```

Main chỉ tạo ở đĩa thử nghiệm và gọi phương thức drive() để mọi thứ hoạt động.



Trình xây dựng khởi tạo cơ sở dữ liệu về mọi người trong dịch vụ mai mối.



Chúng ta hãy lấy lại một người từ DB ...và tạo proxy cho chủ sở hữu. Gọi người nhận và sau đó là một người thiết lập đổi xếp hạng.

và sau đó thử thay điều này không hiệu quả!

Bây giờ hãy tạo một proxy không phải của chủ sở hữu

...và gọi một getter theo sau là một setter ↑

Điều này không có tác dụng!

Sau đó thử đặt xếp hạng

Cách này sẽ hiệu quả!

mẫu proxy

Chạy mã...

```
Cửa sổ chỉnh sửa tệp Trợ giúp Born2BDynamic
% java MatchMakingTestDrive

Tên là Joe Javabean
Quyền lợi được thiết lập từ người đại diện chủ sở hữu
Không thể thiết lập xếp hạng từ proxy của chủ sở hữu
Đánh giá là 7

Tên là Joe Javabean
Không thể thiết lập quyền lợi từ proxy không phải chủ sở hữu
Xếp hạng được thiết lập từ proxy không phải của chủ sở hữu
Đánh giá là 5
%  Xếp hạng mới là mức trung bình của xếp hạng trước đó là 7 và giá
tri do người ủy quyền không phải chủ sở hữu đặt ra là 3.
```

[hỏi đáp về proxy](#)

không có Những câu hỏi ngắn

Q: Vậy chính xác thì cái gì là khía cạnh "động" của proxy động? Có phải là tôi đang khởi tạo proxy và thiết lập nó thành trình xử lý khi chạy không?

A: Không, proxy là động bởi vì lớp của nó được tạo ra khi chạy. Hãy nghĩ về điều này: trước khi mã của bạn chạy, không có lớp proxy nào cả; nó được tạo theo yêu cầu từ tập hợp các giao diện mà bạn truyền cho nó.

Q: InvocationHandler của tôi có vẻ như giống như một proxy rất lạ, nó không triển khai bất kỳ phương thức nào của lớp mà nó đang proxy.

A: Đó là bởi vì InvocationHandler không phải là proxy nó là một lớp mà proxy phân phối đến để xử lý các cuộc gọi phương thức. Bản thân proxy được tạo động khi chạy bằng phương thức tĩnh Proxy.newInstance().

Q: Có cách nào để biết liệu một lớp này là lớp Proxy phải không?

A: Có. Lớp Proxy có một lớp tĩnh phương thức được gọi là isProxyClass(). Gọi phương thức này với một lớp sẽ trả về đúng nếu lớp là lớp proxy động. Ngoài ra, lớp proxy sẽ hoạt động giống như bất kỳ lớp nào khác triển khai một tập hợp giao diện cụ thể.

Q: Có bất kỳ hạn chế nào không? các loại giao diện tôi có thể truyền vào newProxyInstance()?

A: Vâng, có một vài. Đầu tiên, nó là đáng lưu ý là chúng ta luôn truyền cho newProxyInstance() một mảng các giao diện - chỉ cho phép các giao diện, không cho phép các lớp. Các hạn chế chính là tất cả các giao diện không công khai cần phải từ cùng một gói. Bạn cũng không thể có các giao diện có tên phương thức xung đột (tức là hai giao diện có một phương thức có cùng

chữ ký). Ngoài ra còn có một vài đặc điểm nhỏ khác nữa, vì vậy tại một thời điểm nào đó, bạn nên xem qua phần chữ nhô về proxy động trong javadoc.

H: Tại sao bạn lại sử dụng bộ xương? Tôi tưởng chúng ta đã loại bỏ những thứ đó trong Java 1.2 rồi chứ.

A: Bạn nói đúng; chúng ta không cần để thực sự tạo ra bộ khung. Kể từ Java 1.2, thời gian chạy RMI có thể phân phối các cuộc gọi của máy khách trực tiếp đến dịch vụ từ xa bằng cách sử dụng phản chiếu. Nhưng chúng tôi muốn trình bày sơ lược vì về mặt khái niệm, điều này giúp bạn hiểu rằng có điều gì đó ẩn sau giúp quá trình giao tiếp giữa máy khách và dịch vụ từ xa diễn ra.

H: Tôi nghe nói rằng trong Java 5, tôi thậm chí không cần phải tạo stub nữa. Có đúng vậy không?

A: Chắc chắn là vậy. Trong Java 5, RMI và Dynamic Proxy đã tập hợp lại và bây giờ stub được tạo động bằng Dynamic Proxy. Stub của đối tượng từ xa là một thê hiện java.lang.reflect.Proxy (có trình xử lý lệnh gọi) được tạo tự động để xử lý tất cả các chi tiết về việc nhận các lệnh gọi phương thức cục bộ của máy khách đến đối tượng từ xa. Vì vậy, bây giờ bạn không cần phải sử dụng rmic nữa; mọi thứ bạn cần để kết nối máy khách với một đối tượng từ xa đều được xử lý ở chế độ nền.

mẫu proxy



Ghép mỗi mẫu với mô tả của nó:

Mẫu

Sự miêu tả

Người trang trí

Bao bọc một đối tượng
khác và cung cấp một giao
diện khác cho nó

Mặt tiền

Bao bọc một đối tượng
khác và cung cấp hành
vi bổ sung cho nó

Người ủy nhiệm

Bao bọc một đối tượng khác để kiềm
soát quyền truy cập vào nó

Bộ chuyển đổi

Bao bọc một loạt các
đối tượng để đơn giản hóa giao
diện của chúng

sở thú proxy

Vườn thú Proxy

Chào mừng đến với Sở thú Objectville!

Bây giờ bạn đã biết về proxy từ xa, áo và bảo vệ, nhưng ngoài thực tế, bạn sẽ thấy rất nhiều đột biến của mô hình này.

Ở góc Proxy của sở thú này, chúng tôi có một bộ sưu tập các mẫu proxy hoang dã tuyệt đẹp mà chúng tôi đã chụp lại để bạn nghiên cứu.

Công việc của chúng tôi vẫn chưa xong; chúng tôi chắc chắn rằng bạn sẽ thấy nhiều biến thể của mẫu này hơn nữa trong thế giới thực, vì vậy hãy giúp chúng tôi lập danh mục nhiều proxy hơn. Hãy cùng xem bộ sưu tập hiện có:



Tướng lửa Proxy
kiểm soát quyền truy cập
vào một tập hợp các
tài nguyên mạng, bảo vệ chủ
thể khỏi các máy khách "xấu".

Môi trường sống: thường thấy ở vị
trí hệ thống tường lửa của công ty.

Giúp tìm môi trường sống

Smart Reference Proxy cung cấp
các hành động bổ sung bắt cứ khi
nào một chủ đề được tham
chiếu, chẳng hạn như đếm số lượng tham
chiếu đến



một vật thể.



Caching Proxy cung cấp bộ nhớ
tạm thời cho kết quả của các
hành động tốn kém. Nó cũng
có thể cho phép nhiều máy
khách chia sẻ kết quả để giảm độ trễ tính toán hoặc
mạng.



Môi trường sống: thường thấy trong các proxy máy chủ
web cũng như các hệ thống quản lý và xuất bản nội dung.

mẫu proxy

Proxy đồng bộ hóa
cung cấp quyền truy cập an
toàn vào một chủ đề từ
nhiều luồng.

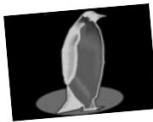


Được thấy xung quanh JavaSpaces, nơi nó
kiểm soát quyền truy cập được đồng bộ
hóa vào một tập hợp các đối tượng cơ
bản trong môi trường phân tán.

Giúp tìm môi trường sống



Proxy ẩn độ phức tạp
ẩn đi sự phức tạp và kiểm
soát quyền truy cập vào một
tập hợp các lớp phức tạp.
Đôi khi, điều này được gọi
là Facade Proxy vì những lý do hiển nhiên.
Proxy ẩn độ phức tạp khác với Facade
Pattern ở chỗ proxy kiểm soát quyền truy
cập, trong khi Facade Pattern chỉ cung cấp
một giao diện thay thế.



Copy-On-Write Proxy kiểm
soát việc sao chép một
đối tượng bằng cách trì
hoãn việc sao chép
một đối tượng cho đến khi máy
khách yêu cầu. Đây là một biến
thể của Virtual Proxy.



Môi trường sống: được nhìn thấy ở
gần CopyOnArrayList của Java 5.

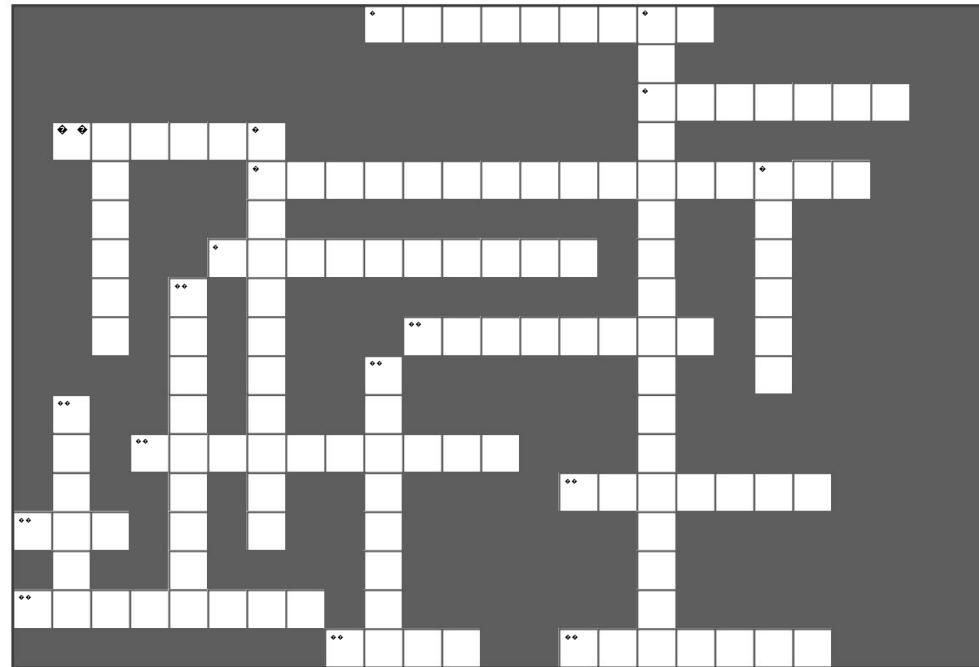
Ghi chú thực địa: vui lòng thêm quan sát của bạn về các proxy khác trong tự nhiên tại đây:

bạn đang ở đây 4 489

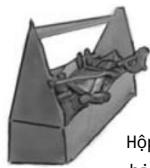
trò chơi ô chữ



Đây là một chương DÀI. Tại sao không thu giãn bằng cách giải ô chữ trước khi kết thúc?

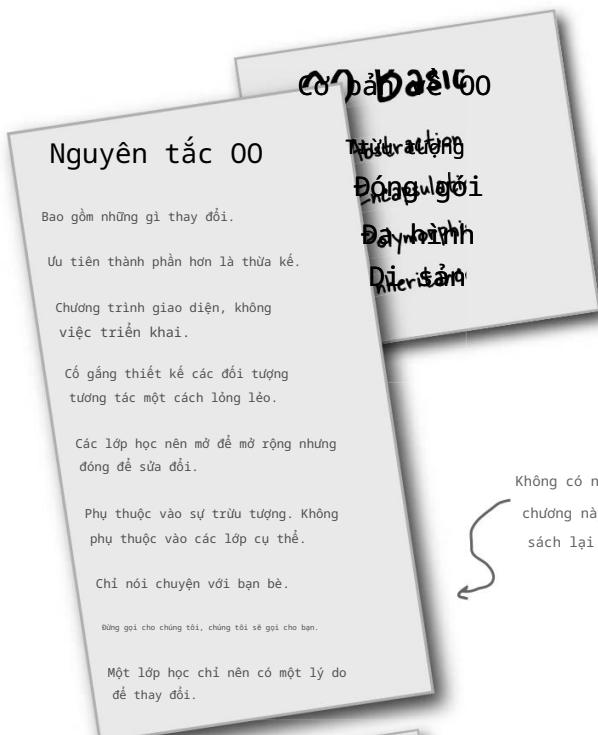


mẫu proxy



Công cụ cho hộp công cụ thiết kế của bạn

Hộp công cụ thiết kế của bạn gần như đã đầy đủ; bạn đã chuẩn bị sẵn sàng cho hầu hết mọi vấn đề thiết kế có thể gặp phải.



Không có nguyên tắc mới nào trong chương này, bạn có thể đóng sách lại và nhớ hết không?



Mẫu mới của chúng tôi.
Proxy đóng vai trò là người đại diện cho một đối tượng khác.

ĐIỂM ĐẦU TIÊN

β Proxy Pattern cung cấp một đại diện cho một đối tượng khác để kiểm soát quyền truy cập của máy khách vào đối tượng đó. Có một số cách để nó có thể quản lý quyền truy cập đó.

β Một Proxy từ xa quản lý tương tác giữa máy khách và đối tượng từ xa.

β Proxy áo kiểm soát quyền truy cập vào một đối tượng tồn kém để khởi tạo.

β Proxy bảo vệ kiểm soát quyền truy cập vào các phương thức của đối tượng dựa trên người gọi.

β Có nhiều biến thể khác nhau của Proxy Pattern bao gồm proxy lưu trữ đệm, proxy đồng bộ hóa, proxy tường lửa, proxy sao chép khi ghi, v.v.

β Proxy có cấu trúc tương tự như Decorator, nhưng mục đích của hai cái này lại khác nhau.

β Mẫu trang trí thêm vào hành vi của một đối tượng, trong khi Proxy kiểm soát quyền truy cập.

β Hỗ trợ tích hợp của Java cho Proxy có thể xây dựng một lớp proxy động theo yêu cầu và phân phối tất cả các cuộc gọi đến lớp đó với trình xử lý mà bạn chọn.

β Giống như bất kỳ wrapper nào, proxy sẽ tăng số lượng lớp và đối tượng trong thiết kế của bạn.

giải bài tập



Giải pháp bài tập



Bài tập

NonOwnerInvocationHandler hoạt động giống như OwnerInvocationHandler, ngoại trừ việc nó cho phép gọi đến setHotOrNotRating() và không cho phép gọi đến bất kỳ phương thức set nào khác. Hãy tiếp tục và tự viết trình xử lý này:

```

nhập java.lang.reflect.*;

lớp công khai NonOwnerInvocationHandler triễn khai InvocationHandler {
    PersonBean người;

    công khai NonOwnerInvocationHandler(PersonBean người) {
        this.person = người;
    }

    public Object invoke(Object proxy, Method method, Object[] args) throws IllegalAccessException {
        thủ {
            nếu (phương thức getName().startsWith("get")) {
                phương thức trả về.invoke(person, args);
            } nếu không thì nếu (phương thức.getName().bằng("setHotOrNotRating")) {
                phương thức trả về.invoke(person, args);
            } else if (phương thức.getName().startsWith("set")) {
                ném ngoại lệ IllegalAccessException mới();
            }
        } bắt (InvocationTargetException e) {
            e.printStackTrace();
        } trả về null;
    }
}

```

Lớp thiết kế

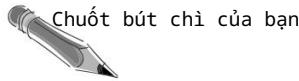
Lớp ImageProxy của chúng ta dường như có hai trạng thái được điều khiển bởi các câu lệnh có điều kiện. Bạn có thể nghĩ ra một mẫu khác có thể dọn dẹp mã này không? Bạn sẽ thiết kế lại ImageProxy như thế nào?

Sử dụng State Pattern: triển khai hai trạng thái, ImageLoaded và ImageNotLoaded. Sau đó, đưa mã từ các câu lệnh if vào các trạng thái tương ứng của chúng. Bắt đầu ở trạng thái ImageNotLoaded và sau đó chuyển sang trạng thái ImageLoaded sau khi ImageIcon đã được truy xuất.

mẫu proxy

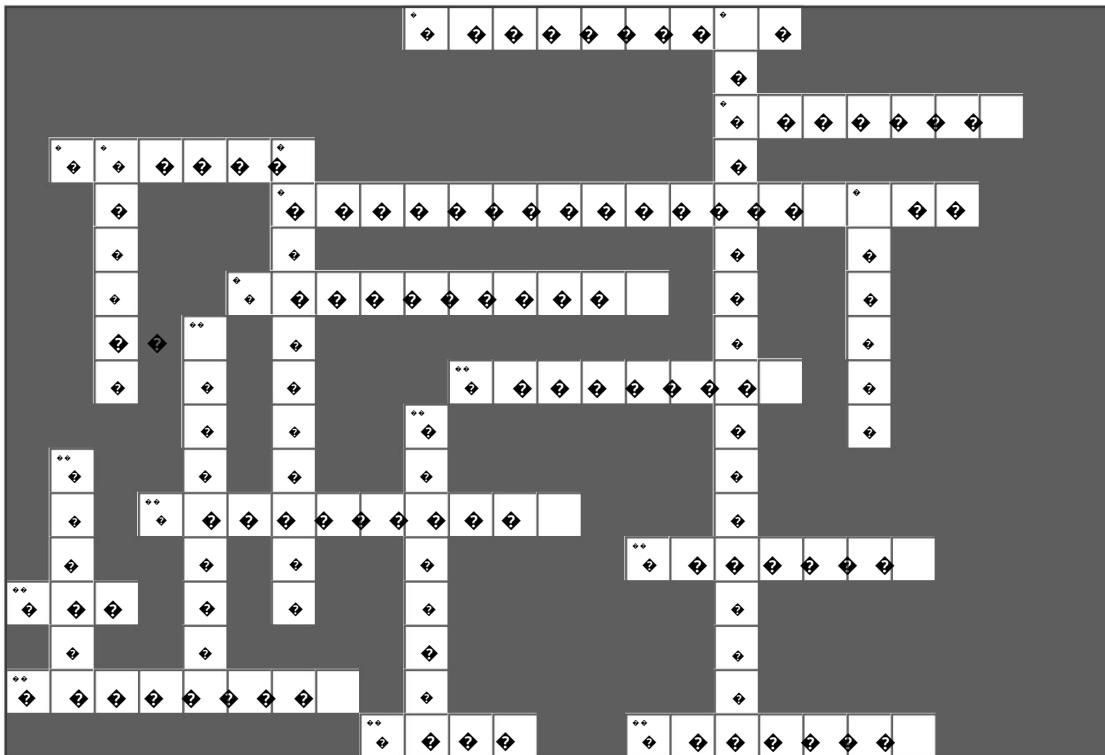


Giải pháp bài tập



Mặc dù hơi phức tạp, nhưng không có nhiều thứ để tạo proxy động. Tại sao bạn không viết `getNonOwnerProxy()`, trả về proxy cho `NonOwnerInvocationHandler`:

```
PersonBean getNonOwnerProxy(PersonBean người) {
    trả về (PersonBean) Proxy.newProxyInstance(
        người.getClass().getClassLoader(),
        người.getClass().getInterfaces(),
        new NonOwnerInvocationHandler(người));
}
```



bạn đang ở đây 4 493

mã ready-bak: trình xem bìa đĩa cd



Sẵn sàng nướng
Mã số

Mã cho Trình xem bìa CD

```

gói headfi rst.proxy.virtualproxy; nhập java.net.*;
nhập java.awt.*; nhập
java.awt.event.*; nhập
javax.swing.*; nhập java.util.*;
lớp công khai ImageProxyTestDrive
{ ImageComponent
imageComponent; JFrame frame = new JFrame("CD
Cover Viewer"); JMenuBar menuBar;

Trình đơn JMenu;
Bảng băm cds = new Hashtable();

public static void main (String[] args) ném Ngoại lệ {
    ImageProxyTestDrive testDrive = new ImageProxyTestDrive();
}

public ImageProxyTestDrive() throws Exception{ cds.put("Ambient:
Âm nhạc cho Sân bay","http://images.amazon.com/images/P/
B000003S2K.01.LZZZZZZZ.jpg");
cds.put ("Thanh Phật","http://images.amazon.com/images/P/B00009XBYK.01.LZZZZZZZ.
jpg");
cds.put ("Ima","http://images.amazon.com/images/P/B000005IRM.01.LZZZZZZZ.jpg"); cds.put("Karma","http://
images.amazon.com/images/P/B000005DCB.01.LZZZZZZZ.gif"); cds.put("MCMXC AD","http://images.amazon.com/images/P/
B000002URV.01.LZZZZZZZ.
jpg");
cds.put ("Northern Exposure","http://images.amazon.com/images/P/B000003SFN.01.
LZZZZZZZ.jpg");
cds.put ("Các tác phẩm Ambient được chọn, Tập 2","http://images.amazon.com/images/P/
B000002MNZ.01.LZZZZZZZ.jpg");
cds.put("oliver","http://www.cs.yale.edu/homes/freeman-elisabeth/2004/9/Oliver_ sm.jpg");

URL initialURL = new URL((String)cds.get("Selected Ambient Works, Vol. 2")); menuBar = new JMenuBar(); menu =
new JMenu("Các đĩa CD yêu thích");
menuBar.add(menu); frame.setJMenuBar(menuBar);

```

mẫu proxy

```
dưới với (Enumeration e = cds.keys(); e.hasMoreElements();) { Tên chuỗi =
    (Chuỗi)e.nextElement(); JMenuItem menuItem = new JMenuItem(tên);
    menu.add(menuItem); menuItem.addActionListener(ActionListener())
mới {
    public void actionPerformed(ActionEvent event) { imageComponent.setIcon(new
        ImageProxy(getCDUrl(event.getActionCom-
lệnh())));
    frame. sơn lại();
}
});
}

// thiết lập khung và menu

Biểu tượng icon = new ImageProxy(initialURL); imageComponent =
new ImageComponent(icon); frame.getContentPane().add(imageComponent);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setSize(800,600); frame.setVisible(true);

}

URL getCDUrl(String name) { thử { trả về
    URL
    mới((String)cds.get(name));
} catch (MalformedURLException e) { e.printStackTrace();
    trả về null;
}
}
}
```

bạn đang ở đây 4 495

mã ready-bak: trình xem bìa đĩa cd



Sẵn sàng nướng
Mã số

Mã cho Trình xem bìa CD, tiếp
theo...

```

gói headfi rst.proxy.virtualproxy; nhập java.net.*;
nhập java.awt.*; nhập
java.awt.event.*; nhập
javax.swing.*;

lớp ImageProxy thực hiện Icon {
    Biểu tượng hình ảnh;
    URL hình ảnhURL;
    Truy xuất luồngThread; boolean truy
    xuất = false;

    công khai ImageProxy(URL url) { imageURL = url; }

    công khai int getIconWidth() { nếu (imageIcon !
        = null) { trả về
            imageIcon.getIconWidth(); } nếu không { trả về 800;

        }
    }

    công khai int getIconHeight() { nếu (imageIcon !
        = null) { trả về
            imageIcon.getIconHeight(); } nếu không { trả về 600;

        }
    }

    public void paintIcon(Thành phần cuối cùng c, Đồ họa g, int x, int y) {
        if (imageIcon != null)
            { imageIcon.paintIcon(c, g, x, y); } else

            { g.drawString("Đang tải bìa CD, vui lòng đợi...", x+300, y+190); if (!retrieving) { retrieving = true;

                retrievalThread = new Thread(new Runnable() { public void run() { try
                    { imageIcon = new
                        ImageIcon(imageURL, "CD Cover"); c.repaint();

                    } catch (Ngoại lệ e) {

                }
            }
        }
    }
}

```

mẫu proxy

```
        e.printStackTrace();
    }
}
}
);
retrievalThread.start();
}
}
}
}
```

```
gói headfirst.proxy.virtualproxy; nhập java.awt.*;  
nhập javax.swing.*;
```

```
lớp ImageComponent mở rộng JComponent {
    biểu tượng riêng tư;

    public ImageComponent(Biểu tượng biểu tượng) { this.icon
        = biểu tượng;
    }

    public void setIcon(Biểu tượng icon) { this.icon = icon;
    }

    public void paintComponent(Graphics g) { super.paintComponent(g);
        int w = icon.getIconWidth(); int h
        = icon.getIconHeight(); int x = (800 - w)/
        2; int y = (600 - h)/2; icon.paintIcon(this,
        g, x, y);

    }
}
```

bạn đang ở đây 4 497

12 Mẫu hợp chất

h Các mẫu
của các mẫu gg



Ai có thể ngờ rằng các Mẫu có thể hoạt động cùng nhau?

Bạn đã chứng kiến Fireside Chats gay gắt (và bạn thậm chí còn chưa xem các trang Pattern Death Match mà biên tập viên đã buộc chúng tôi xóa khỏi cuốn sách*), vậy ai có thể nghĩ rằng các mẫu thực sự có thể hòa hợp tốt với nhau? Vâng, tin hay không thì tùy, một số thiết kế OO mạnh mẽ nhất sử dụng nhiều mẫu cùng nhau. Hãy chuẩn bị đưa kỹ năng về mẫu của bạn lên một tầm cao mới; đã đến lúc sử dụng các mẫu hợp chất.

*

gửi email cho chúng tôi để xin một bản sao.

đây là một chương mới

499

các mẫu có thể làm việc cùng nhau

Làm việc cùng nhau

Một trong những cách tốt nhất để sử dụng các mẫu là đưa chúng ra khỏi nhà để chúng có thể tương tác với các mẫu khác. Bạn càng sử dụng nhiều mẫu, bạn càng thấy chúng xuất hiện cùng nhau trong các thiết kế của mình. Chúng tôi có một tên gọi đặc biệt cho một tập hợp các mẫu hoạt động cùng nhau trong một thiết kế có thể áp dụng cho nhiều vấn đề: mẫu ghép. Đúng vậy, bây giờ chúng ta đang nói về các mẫu được tạo thành từ các mẫu!

Bạn sẽ thấy rất nhiều mẫu hợp chất được sử dụng trong thế giới thực. Bây giờ bạn đã có các mẫu trong não, bạn sẽ thấy rằng chúng thực sự chỉ là các mẫu hoạt động cùng nhau và điều đó giúp bạn dễ hiểu hơn.

Chúng ta sẽ bắt đầu chương này bằng cách xem lại những chú vịt thân thiện của chúng ta trong trình mô phỏng vịt SimUDuck. Thật hợp lý khi những chú vịt ở đây khi chúng ta kết hợp các mẫu; sau cùng, chúng đã ở bên chúng ta trong suốt toàn bộ cuốn sách và chúng rất thích tham gia vào nhiều mẫu.

Những chú vịt sẽ giúp bạn hiểu cách các mẫu có thể hoạt động cùng nhau trong cùng một giải pháp. Nhưng chỉ vì chúng ta đã kết hợp một số mẫu không có nghĩa là chúng ta có một giải pháp đủ điều kiện là một mẫu hợp chất. Đối với điều đó, nó phải là một giải pháp mục đích chung có thể áp dụng cho nhiều vấn đề.

Vì vậy, trong nửa sau của chương, chúng ta sẽ xem xét một mẫu hợp chất thực sự: đúng vậy, chính là ông Model-View-Controller. Nếu bạn chưa từng nghe đến ông ấy, bạn sẽ biết, và bạn sẽ thấy mẫu hợp chất này là một trong những mẫu mạnh mẽ nhất trong hộp công cụ thiết kế của bạn.



Các mẫu thường được sử dụng cùng nhau
và kết hợp trong cùng một giải pháp thiết kế.

Một mô hình hợp chất kết hợp hai hoặc
nhiều mô hình thành một giải pháp giải quyết
một vấn đề chung hoặc thường gặp.

Đoàn tụ vịt

Như bạn đã nghe, chúng ta sẽ lại làm việc với những chú vịt. Lần này, những chú vịt sẽ cho bạn thấy cách các mô hình có thể cùng tồn tại và thậm chí hợp tác trong cùng một giải pháp.

Chúng ta sẽ xây dựng lại trình mô phỏng vịt của mình từ đầu và cung cấp cho nó một số khả năng thú vị bằng cách sử dụng một loạt các mẫu. Được rồi, hãy bắt đầu nào...

① Đầu tiên, chúng ta sẽ tạo giao diện Quackable.

Như chúng tôi đã nói, chúng tôi đang bắt đầu từ con số không. Lần này, Ducks sẽ triển khai giao diện Quackable. Theo cách đó, chúng tôi sẽ biết những thứ nào trong trình mô phỏng có thể kêu quack() - như Mallard Ducks, Redhead Ducks, Duck Calls, và chúng tôi thậm chí có thể thấy Rubber Duck lén vào.

```
giao diện công cộng Quackable {
    công khai void quack();
}
```

Những kè hay kêu quạc quạc chỉ cần
làm tốt một việc: Quạc quạc!

② Nay giờ, một số Ducks thực hiện Quackable

Giao diện mà không có một số lớp để triển khai thì có ích gì? Đã đến lúc tạo ra một số con vịt bê tông (nhưng không phải loại "nghệ thuật bãi cỏ", nếu bạn hiểu ý chúng tôi).

```
lớp công khai MallardDuck triển khai Quackable {
    công khai void quack() {
        System.out.println("Quack");
    }
}
```

Vịt trời tiêu chuẩn
của bạn.

```
lớp công khai RedheadDuck triển khai Quackable {
    công khai void quack() {
        System.out.println("Quack");
    }
}
```

Chúng ta phải có một số loài
đa dạng nếu muốn đây là một trò
chơi mô phỏng thú vị.

thêm nhiều vịt hơn

Sẽ chẳng vui chút nào nếu chúng ta không thêm vào những loài vịt khác.

Bạn còn nhớ lần trước không? Chúng ta đã nghe tiếng vịt kêu (thú mà thợ săn thường dùng, chúng chắc chắn có tiếng kêu) và tiếng vịt cao su.

```
lớp công khai DuckCall thực hiện Quackable {
    công khai void quack() {
        System.out.println("Kwak");
    }
}
```

Một chú vịt kêu nhưng nghe không giống tiếng kêu thật.

```
lớp công khai RubberDuck thực hiện Quackable {
    công khai void quack() {
        System.out.println("Tiếng kêu cót két");
    }
}
```

Một chú vịt cao su kêu cót két khi kêu.

3 Được rồi, chúng ta đã có sự chuẩn bị; bây giờ tất cả những gì chúng ta cần là một trình mô phỏng.

Chúng ta hãy cùng tạo ra một chương trình mô phỏng tạo ra một vài chú vịt và đảm bảo tiếng kêu của chúng vẫn hoạt động...

```
lớp công khai DuckSimulator {
    public static void main(String[] args) {
        Trình mô phỏng DuckSimulator = new DuckSimulator();
        mô phỏng.mô phỏng();
    }
}
```

Đây là phương pháp chính của chúng tôi để thực hiện mọi việc.

Chúng ta tạo một trình mô phỏng và sau đó gọi phương thức simulate() của nó.

```
void mô phỏng() {
    Vịt trời có thể quack = new MallardDuck();
    Quackable redheadDuck = new RedheadDuck();
    Quackable duckCall = new DuckCall();
    Quackable rubberDuck = new RubberDuck();
```

Chúng ta cần một số con vịt, vì vậy chúng ta sẽ tạo ra mỗi loại một con Quackable...

```
System.out.println("\nTrình mô phỏng vịt");
```

```
    mô phỏng(vịt trời);
    mô phỏng(redheadDuck);
    mô phỏng(duckCall);
    mô phỏng(rubberDuck);
```

...sau đó chúng ta mô phỏng từng cái một.

```
}
```

Ở đây chúng ta sử dụng phương thức mô phỏng quá mức để mô phỏng chỉ một con vịt.

```
}
```



Ở đây chúng ta để đa hình thực hiện phép thuật của nó: bắt kể loại Quackable nào được truyền vào, phương thức simulate() đều yêu cầu nó quack.

mẫu hợp chất

Chưa thực sự thú vị nhưng
chúng tôi vẫn chưa thêm mẫu!



```
Cửa sổ chỉnh sửa tệp Trợ giúp ItBetterGetBetterThanThis
% java DuckSimulator

Mô phỏng vịt
Lang bäm
Lang bäm
Kwak

Tiếng kêu cát két

%
```

Tất cả chúng đều triển khai cùng một giao diện Quackable, nhưng cách triển khai của chúng cho phép chúng quack theo cách riêng của mình.

Có vẻ như mọi thứ đều ổn; cho đến giờ thì mọi thứ đều tốt.

④ Khi có vịt ở xung quanh, ngỗng không thể đi xa.

Nơi nào có một loài chim nước, có lẽ sẽ có hai. Đây là lớp Ngỗng đã xuất hiện xung quanh trình mô phỏng.

```
lớp công khai Ngỗng {
    công khai void honk() {
        System.out.println("Bấm còi");
    }
}
```



Ngỗng là loài kêu chứ không phải
là loài kêu quác quác.

não Apower

Giả sử chúng ta muốn có thể sử dụng một con ngỗng ở bất cứ nơi nào chúng ta muốn sử dụng một con vịt. Rốt cuộc, ngỗng tạo ra tiếng động; ngỗng bay; ngỗng bơi. Tại sao chúng ta không thể có ngỗng trong trình mô phỏng?

Kiểu mẫu nào cho phép loài Ngỗng dễ dàng hòa nhập với loài Vịt?

bạn đang ở đây 4 503

bộ chuyển đổi ngỗng

5 Chúng ta cần một bộ chuyển đổi ngỗng.

Trình mô phỏng của chúng tôi mong đợi thấy giao diện Quackable. Vì ngỗng không phải là loài kê (chúng là loài kê the thé), chúng ta có thể sử dụng bộ điều hợp để chuyển đổi ngỗng thành vịt.

```

lớp công khai GooseAdapter triết khai Quackable {
    Ngỗng ngỗng;

    công khai GooseAdapter(Ngỗng ngỗng) {
        this.goose = ngỗng;
    }

    công khai void quack() {
        ngỗng.kêu ();
    }
}

```

Hãy nhớ rằng, Bộ điều hợp triết khai giao diện mục tiêu, trong trường hợp này là Quackable.

Người xây dựng sẽ lấy con ngỗng mà chúng ta sẽ thích nghi.

Khi gọi quack, lệnh gọi sẽ được chuyển đến phương thức honk() của con ngỗng.

6 Bây giờ, ngỗng cũng có thể chơi trong trình mô phỏng.

Tất cả những gì chúng ta cần làm là tạo một Goose, bọc nó trong một bộ điều hợp triết khai Quackable và chúng ta sẽ ổn thôi.

```

lớp công khai DuckSimulator {
    public static void main(String[] args) {
        Trình mô phỏng DuckSimulator = new DuckSimulator();
        mô phỏng.mô phỏng();
    }

    void mô phỏng() {
        Vịt trời có thể quack = new MallardDuck();
        Quackable redheadDuck = new RedheadDuck();
        Quackable duckCall = new DuckCall();
        Quackable rubberDuck = new RubberDuck();
        Quackable gooseDuck = new GooseAdapter(new Goose());
    }
}

```

Chúng ta tạo ra một con Ngỗng hoạt động như một con Vịt bằng cách bọc con Ngỗng trong GooseAdapter.

System.out.println("\nTrình mô phỏng vịt: Với bộ điều hợp ngỗng");

```

mô phỏng(vịt trời);
mô phỏng(redheadDuck);
mô phỏng(duckCall);
mô phỏng(rubberDuck);
mô phỏng(ngỗngVịt);
}

```

Sau khi đã quản xong con ngỗng, chúng ta có thể xử lý nó giống như những con vịt Quackable khác.

```

void mô phỏng(Vịt có thể quack) {
    vịt.quack();
}
}

```

7 Bây giờ chúng ta hãy chạy thử một chút....

Lần này khi chúng ta chạy trình mô phỏng, danh sách các đối tượng được truyền cho phương thức `simulate()` bao gồm một `Goose` được bọc trong một bộ điều hợp `duck`. Kết quả là gì? Chúng ta sẽ thấy một số tiếng kêu!

Đây rồi, con ngỗng! Bây giờ
con ngỗng có thể kêu cùng
với những con vịt khác.

```
Cửa sổ chỉnh sửa tập tin Trợ giúp GoldenEggs
% java DuckSimulator
Duck Simulator: Với Bộ Chuyển Đổi Ngỗng
Lang băm
Lang băm
Lang băm
Kwak
Tiếng kêu cót kèt
Tiếng còi

%
```



Quackology

Các nhà nghiên cứu về Quack rất thích thú với mọi khía cạnh của hành vi `Quackable`. Một điều mà các nhà nghiên cứu về Quack luôn muốn nghiên cứu là tổng số tiếng kêu của một đàn vịt.

Làm thế nào chúng ta có thể bổ sung khả năng đếm tiếng kêu của vịt mà không cần phải thay đổi các lớp vịt?

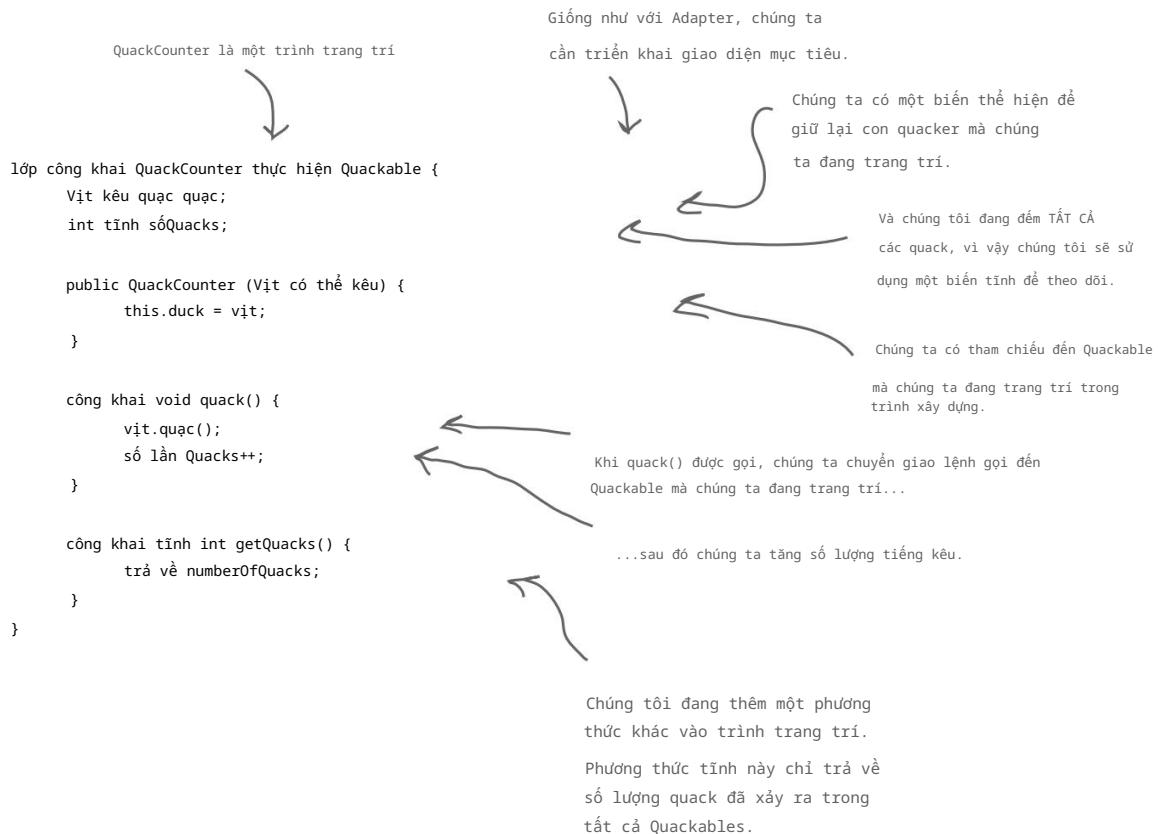
Bạn có thể nghĩ ra một mô hình nào có thể giúp ích không?



người trang trí vịt

⑧ Chúng ta sẽ làm cho những nhà Quackologist vui vẻ và
đếm số lượng quack cho họ.

Làm sao? Hãy tạo một trình trang trí cung cấp cho vịt một số hành vi mới (hành vi đếm) bằng cách bao bọc chúng bằng một đối tượng trang trí. Chúng ta sẽ không phải thay đổi mã Duck chút nào.



9 Chúng ta cần cập nhật trình mô phỏng để tạo ra những chú vịt trang trí.

Bây giờ, chúng ta phải bao bọc từng đối tượng Quackable mà chúng ta khởi tạo trong một trình trang trí QuackCounter. Nếu không, chúng ta sẽ có những con vịt chạy xung quanh và tạo ra những tiếng kêu không đếm được.

```

lớp công khai DuckSimulator {
    public static void main(String[] args) {
        Trình mô phỏng DuckSimulator = new DuckSimulator();
        mô phỏng.mô phỏng();
    }
    void mô phỏng() {
        Quackable mallardDuck = new QuackCounter(new MallardDuck());
        Quackable redheadDuck = new QuackCounter(new RedheadDuck());
        Quackable duckCall = new QuackCounter(new DuckCall());
        Quackable rubberDuck = new QuackCounter(new RubberDuck());
        Quackable gooseDuck = new GooseAdapter(new Goose());

        System.out.println("\nTrình mô phỏng vịt: Với trình trang trí");

        mô phỏng(vịt trời);
        mô phỏng(redheadDuck);
        mô phỏng(duckCall);
        mô phỏng(rubberDuck);
        mô phỏng(ngỗngVịt);

        System.out.println("Những con vịt kêu
                           " + QuackCounter.getQuacks() +
                           " lần");
    }
}

void mô phỏng(Vịt có thể quack) {
    vịt.quack();
}
}

```

Mỗi lần chúng ta tạo một Quackable, chúng ta sẽ bao bọc nó bằng một trình trang trí mới.

Người kiểm lâm nói với chúng tôi rằng ông ấy không muốn đếm tiếng kêu của ngỗng nên chúng tôi không trang trí nó.

Đây là nơi chúng tôi thu thập hành vi kêu quacking cho các nhà Quackologist.

Không có gì thay đổi ở đây; các đồ vật được trang trí vẫn là Quackables.

Đây là kết quả!

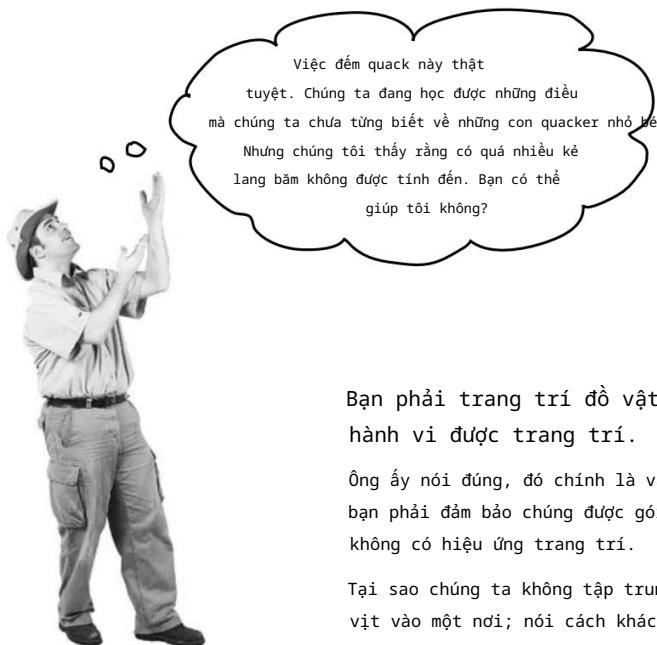
Hãy nhớ rằng chúng ta không tính số ngỗng.

```

Cửa sổ chính sửa tệp Trợ giúp DecoratedEggs
% java DuckSimulator
Duck Simulator: Với Decorator
Lang băm
Lang băm
Kwak
Tiếng kêu cót két
Tiếng còi
Đã đếm được 4 con quack
%

```

nhà máy vịt



⑩ Chúng ta cần một nhà máy để sản xuất vịt!

Được rồi, chúng ta cần kiểm soát chất lượng để đảm bảo vịt của chúng ta được đóng gói.

Chúng ta sẽ xây dựng toàn bộ một nhà máy chỉ để sản xuất chúng. Nhà máy sẽ sản xuất một nhóm sản phẩm bao gồm nhiều loại vịt khác nhau, vì vậy chúng ta sẽ sử dụng Mẫu nhà máy trừu tượng.

Chúng ta hãy bắt đầu với định nghĩa của `AbstractDuckFactory`:

```
lớp trừu tượng công khai AbstractDuckFactory {
```

```
    tóm tắt công khai Quackable createMallardDuck();
    tóm tắt công khai Quackable createRedheadDuck();
    tóm tắt công khai Quackable createDuckCall();
    tóm tắt công khai Quackable createRubberDuck();
}
```

Chúng tôi đang định nghĩa một nhà máy trừu tượng mà các lớp con sẽ triển khai để tạo ra các họ khác nhau.

Mỗi phương pháp tạo ra một loại vịt.

mẫu hợp chất

Chúng ta hãy bắt đầu bằng cách tạo một nhà máy tạo ra vịt mà không cần trình trang trí, chỉ để hiểu rõ hơn về nhà máy:

```
lớp công khai DuckFactory mở rộng AbstractDuckFactory {

    công khai Quackable createMallardDuck() {
        trả về MallardDuck mới();
    }

    công khai Quackable createRedheadDuck() {
        trả về RedheadDuck mới();
    }

    công khai Quackable createDuckCall() {
        trả về DuckCall() mới();
    }

    công khai Quackable createRubberDuck() {
        trả về RubberDuck() mới();
    }
}
```

DuckFactory mở rộng
nhà máy trừu tượng.

Mỗi phương pháp tạo ra một sản phẩm:
một loại Quackable cụ thể.
Trình mô phỏng không biết sản phẩm
thực tế là gì - nó chỉ biết rằng nó
đang nhận được một tiếng Quackable.

Bây giờ chúng ta hãy tạo nhà máy mà chúng ta thực sự muốn, CountingDuckFactory:

```
lớp công khai CountingDuckFactory mở rộng AbstractDuckFactory {

    công khai Quackable createMallardDuck() {
        trả về QuackCounter mới(MallardDuck() mới());
    }

    công khai Quackable createRedheadDuck() {
        trả về QuackCounter mới(RedheadDuck mới());
    }

    công khai Quackable createDuckCall() {
        trả về QuackCounter mới(DuckCall() mới());
    }

    công khai Quackable createRubberDuck() {
        trả về QuackCounter mới(RubberDuck mới());
    }
}
```

CountingDuckFactory
cũng mở rộng
nhà máy trừu tượng.

Mỗi phương pháp bao bọc
Quackable bằng trình trang
trí đếm quack. Trình mô
phỏng sẽ không bao giờ biết
sự khác biệt; nó chỉ nhận
lại được một Quackable.
Nhưng bây giờ các kiểm lâm của
chúng tôi có thể chắc chắn rằng tất cả
các con quạ đều được đếm.

các gia đình vịt

(11) Chúng ta hãy thiết lập trình mô phỏng để sử dụng nhà máy.

Bạn còn nhớ Abstract Factory hoạt động như thế nào không? Chúng ta tạo ra một phương thức đa hình lấy một nhà máy và sử dụng nó để tạo ra các đối tượng. Bằng cách truyền vào các nhà máy khác nhau, chúng ta có thể sử dụng các họ sản phẩm khác nhau trong phương thức.

Chúng ta sẽ thay đổi phương thức simulate() để nó lấy một nhà máy và sử dụng nó để tạo ra vịt.

```

lớp công khai DuckSimulator {
    public static void main(String[] args) {
        Trình mô phỏng DuckSimulator = new DuckSimulator();
        Tóm tắtDuckFactory duckFactory = new CountingDuckFactory();

        simulator.simulate(duckFactory);
    }

    void mô phỏng(AbstractDuckFactory duckFactory) {
        Vịt trời có thể kêu quack = duckFactory.createMallardDuck();
        Quackable redheadDuck = duckFactory.createRedheadDuck();
        DuckCall có thể gọi = duckFactory.createDuckCall();
        Quackable rubberDuck = duckFactory.createRubberDuck();
        Quackable gooseDuck = new GooseAdapter(new Goose());

        System.out.println("\nTrình mô phỏng vịt: Với Nhà máy trừu tượng");

        mô phỏng(vịt trời);
        mô phỏng(redheadDuck);
        mô phỏng(duckCall);
        mô phỏng(rubberDuck);
        mô phỏng(ngỗngVịt);

        System.out.println("Những con vịt kêu " + QuackCounter.getQuacks() + " times");
    }

    void mô phỏng(Vịt có thể quack) {
        vịt.quack();
    }
}

```

Đầu tiên chúng ta tạo nhà máy mà chúng ta sẽ truyền vào phương thức simulate().

Phương thức mô phỏng() lấy một AbstractDuckFactory và sử dụng nó để tạo ra vịt thay vì khởi tạo chúng trực tiếp.

Không có gì thay đổi ở đây cả! Vẫn là mã cũ.

Đây là kết quả đầu ra khi sử dụng nhà máy...

Giống như lần trước,
nhưng lần này chúng ta đảm
bảo rằng tất cả các
chú vịt đều được trang
trí vì chúng ta
đang sử dụng CountingDuckFactory.

Cửa sổ chỉnh sửa tệp Trợ giúp EggFactory

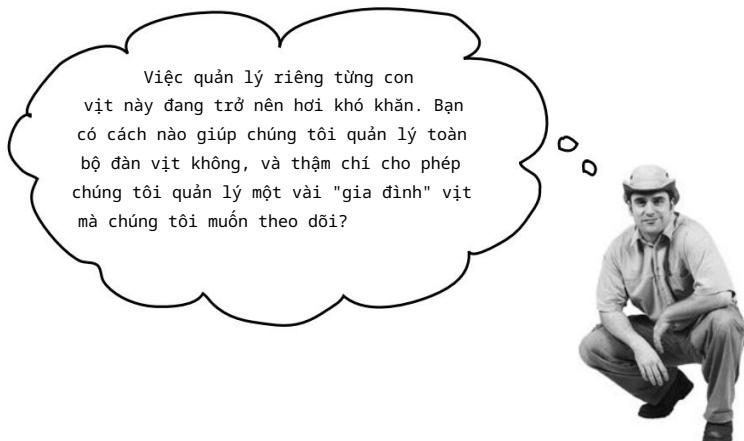
```
% java DuckSimulator
Duck Simulator: Với Abstract Factory
Lang băm
Lang băm
Kwak
Tiếng kêu cót kèt
Tiếng còi
Đã đếm được 4 con quack
%
```



Chuốt bút chì của bạn

Chúng ta vẫn đang trực tiếp khởi tạo Geese bằng cách dựa vào các lớp cụ thể. Bạn có thể
viết Abstract Factory cho Geese không? Nó sẽ xử lý việc tạo ra "goose ducks" như thế nào?

đàn vịt



À, anh ta muốn quản lý một đàn vịt.

Đây là một câu hỏi hay khác từ kiểm lâm Brewer: Tại sao chúng ta phải quản lý từng con vịt riêng lẻ?

Việc này không dễ
quản lý chút nào!

```

Vịt trời có thể kèu quack = duckFactory.createMallardDuck();
Quackable redheadDuck = duckFactory.createRedheadDuck();
DuckCall có thể gọi = duckFactory.createDuckCall();
Quackable rubberDuck = duckFactory.createRubberDuck();
Quackable gooseDuck = new GooseAdapter(new Goose());

mô phỏng(vịt trời);
mô phỏng(redheadDuck);
mô phỏng(duckCall);
mô phỏng(rubberDuck);
mô phỏng(ngỗngVịt);

```

Điều chúng ta cần là một cách để nói về các bộ sưu tập vịt và thậm chí là các bộ sưu tập con vịt (để giải quyết yêu cầu của gia đình từ Ranger Brewer). Sẽ thật tuyệt nếu chúng ta có thể áp dụng các hoạt động trên toàn bộ bộ vịt.

Mẫu nào có thể giúp chúng ta?

(12) Chúng ta hãy tạo ra một đàn vịt (thực ra là một đàn Quackables).

Bạn còn nhớ Composite Pattern cho phép chúng ta xử lý một tập hợp các đối tượng theo cùng một cách như các đối tượng riêng lẻ không? Còn gì tuyệt vời hơn một đàn Quackables!

Chúng ta hãy cùng tìm hiểu cách thức hoạt động của nó:

```

lớp công khai Flock thực hiện Quackable {
    ArrayList quackers = new ArrayList();

    public void add(Quackable quacker) {
        quackers.add(quacker);
    }

    công khai void quack() {
        Trình lặp iterator = quackers.iterator();
        trong khi (iterator.hasNext()) {
            Quackable quacker = (Quackable)iterator.next();
            quacker. quack();
        }
    }
}

```

Hãy nhớ rằng, thành phần tổng hợp cần phải triển khai cùng một giao diện như các phần tử lá. Các phần tử lá của chúng ta là Quackables.

↑ Bây giờ là phương thức quack() - xét cho cùng, Flock cũng là Quackable.

Phương thức quack() trong Flock cần phải hoạt động trên toàn bộ Flock. Ở đây chúng ta lặp qua ArrayList và gọi quack() trên mỗi phần tử.

Chúng tôi sử dụng ArrayList bên trong mỗi Flock để lưu trữ các Quackable thuộc về Flock đó.

← Phương thức add() thêm một Quackable vào Flock.

Mã Gắn

Bạn có nhận thấy chúng tôi đã cố gắng lén đưa cho bạn một Mẫu thiết kế mà không để cập đến nó không?

```

công khai void quack() {
    Trình lặp iterator = quackers.iterator();
    trong khi (iterator.hasNext()) {
        Quackable quacker = (Quackable)iterator.next();
        quacker. quack();
    }
}

```

← → Đây rồi! Mẫu Iterator đang hoạt động!

vịt tổng hợp

(13) Vậy giờ chúng ta cần thay đổi trình mô phỏng.

Cấu trúc tổng hợp của chúng ta đã sẵn sàng; chúng ta chỉ cần một số mã để đưa những chú vịt vào cấu trúc tổng hợp.

```

lớp công khai DuckSimulator {
    // phương pháp chính ở đây

    void mô phỏng(AbstractDuckFactory duckFactory) {
        Quackable redheadDuck = duckFactory.createRedheadDuck();
        DuckCall có thể gọi = duckFactory.createDuckCall();
        Quackable rubberDuck = duckFactory.createRubberDuck();
        Quackable gooseDuck = new GooseAdapter(new Goose());
        System.out.println("\nTrình mô phỏng vịt: Với hồn hợp - Đàm vịt");

        Đàm flockOfDucks = đàm new Flock();

        flockOfDucks.add(vịt đầu đỏ);
        flockOfDucks.add(duckCall);
        flockOfDucks.add(rubberDuck);
        flockOfDucks.add(gooseDuck);

        Đàm flockOfMallards = đàm new Flock();

        Vịt trời có thể quackOne = duckFactory.createMallardDuck();
        Quackable mallardTwo = duckFactory.createMallardDuck();
        Vịt trời có thể kêu quackThree = duckFactory.createMallardDuck();
        Vịt trời QuackableFour = duckFactory.createMallardDuck();

        flockOfMallards.add(mallardOne);
        flockOfMallards.add(mallardTwo);
        flockOfMallards.add(mallardThree);
        flockOfMallards.add(mallardFour);

        đàm vịt.add(đàm vịt trời);

        System.out.println("\nTrình mô phỏng vịt: Mô phỏng toàn bộ đàm vịt");
        mô phỏng(bầy vịt);

        System.out.println("\nTrình mô phỏng vịt: Mô phỏng đàm vịt trời");
        mô phỏng(flockOfMallards);

        System.out.println("\nNhững con vịt kêu "
            + QuackCounter.getQuacks() + " lần");

    }

    void mô phỏng(Vịt có thể quack) {
        vịt.quack();
    }
}

```

Tạo tất cả
Quackables
giống như trước.

Đầu tiên chúng ta tạo một Flock và
nạp Quackable vào đó.
Sau đó, chúng ta tạo ra
một Đàm Vịt Trời mới.

Ở đây chúng
ta đang tạo
ra một gia đình nhỏ
vịt trời...

...và thêm chúng vào Đàm vịt
trời.
Sau đó, chúng ta thêm đàm vịt trời
vào đàm chính.

Hãy cùng kiểm tra toàn bộ Flock nhé!

Vậy thì chúng ta hãy thử nghiệm đàm vịt trời nhé.

Cuối cùng, chúng ta hãy
cung cấp dữ liệu cho Quackologist.

Không có gì cần phải thay đổi ở đây cả, Flock là một Quackable!

Hãy thử xem nhé...

```
Cửa sổ trợ giúp chỉnh sửa tập tin FlockADuck
% java DuckSimulator
Duck Simulator: Với Composite - Đàm
Duck Simulator: Mô phỏng toàn bộ đàn vịt
Lang băm
Kwak
Tiếng kêu cót két
Tiếng còi
Lang băm
Lang băm
Lang băm
Lang băm

Duck Simulator: Mô phỏng đàn vịt trời
Lang băm
Lang băm
Lang băm
Lang băm
Con vịt kêu 11 lần

Đây là đàn đầu tiên.
Và bây giờ là vịt trời.

Dữ liệu có vẻ
tốt (hãy nhớ
rằng con ngỗng
không được tính).
```



An toàn so với minh bạch

Bạn có thể nhớ rằng trong chương Composite Pattern, các thành phần hợp thành (Menu) và các nút lá (MenuItem) có cùng một tập hợp các phương thức chính xác, bao gồm cả phương thức add(). Vì chúng có cùng một tập hợp các phương thức, chúng ta có thể gọi các phương thức trên MenuItem mà thực sự không có ý nghĩa (như cố gắng thêm thứ gì đó vào MenuItem bằng cách gọi add()). Lợi ích của việc này là sự khác biệt giữa lá và thành phần hợp thành là minh bạch: máy khách không cần phải biết liệu nó đang xử lý lá hay thành phần hợp thành; nó chỉ cần gọi cùng một phương thức trên cả hai.

Ở đây, chúng tôi quyết định giữ các phương thức bảo trì con của hợp chất tách biệt với các nút lá: nghĩa là chỉ có Flock mới có phương thức add(). Chúng tôi biết rằng việc cố gắng thêm thứ gì đó vào Duck là vô nghĩa và trong triển khai này, bạn không thể làm vậy. Bạn chỉ có thể add() vào Flock. Vì vậy, thiết kế này an toàn hơn - bạn không thể gọi các phương thức không có ý nghĩa trên các thành phần - nhưng nó ít minh bạch hơn. Bây giờ, máy khách phải biết rằng Quackable là Flock để thêm Quackable vào nó.

Như thường lệ, sẽ có sự đánh đổi khi bạn thiết kế OO và bạn cần cân nhắc chúng khi tạo ra sản phẩm tổng hợp của riêng mình.

người quan sát vịt



Bạn có thể nói "người quan sát" không?

Nghe có vẻ như Quackologist muốn quan sát hành vi của từng con
vịt. Điều đó dẫn chúng ta đến một mô hình được tạo ra để quan sát
hành vi của các vật thể: Mô hình quan sát.

⑯

Đầu tiên chúng ta cần một giao diện Observable.

Hãy nhớ rằng Observable là đối tượng đang được quan sát. Observable cần có phương thức để
đăng ký và thông báo cho người quan sát. Chúng ta cũng có thể có phương thức để xóa người
quan sát, nhưng chúng ta sẽ giữ cho việc triển khai đơn giản ở đây và bỏ qua điều đó.

```
giao diện công khai QuackObservable {  
    public void registerObserver(Người quan sát quan sát);  
    công khai void notifyObservers();  
}
```

QuackObservable là giao diện
mà Quackables phải triển
khai nếu muốn được quan sát.

Nó cũng có phương pháp để
thông báo cho người quan sát.

Nó có một phương pháp để đăng ký
Observer. bất kỳ đối tượng nào triển khai
giao diện Observer đều có thể lắng
nghe tiếng kêu. Chúng ta sẽ định nghĩa
giao diện Observer sau một giây.

Bây giờ chúng ta cần đảm bảo tất cả Quackables đều triển khai giao diện này...

```
giao diện công cộng Quackable mở rộng QuackObservable {  
    công khai void quack();  
}
```

Vì vậy, chúng tôi mở rộng giao
diện Quackable bằng QuackObserver.

mẫu hợp chất

- 15) Nay giờ, chúng ta cần đảm bảo rằng tất cả các lớp cụ thể triển khai Quackable đều có thể xử lý được vai trò là QuackObservable.

Chúng ta có thể tiếp cận vấn đề này bằng cách triển khai đăng ký và thông báo trong từng lớp (như chúng ta đã làm trong Chương 2). Nhưng lần này chúng ta sẽ làm hơi khác một chút: chúng ta sẽ đóng gói mã đăng ký và thông báo trong một lớp khác, gọi là Observable và biên soạn nó với QuackObservable. Theo cách đó, chúng ta chỉ viết mã thực một lần và QuackObservable chỉ cần đưa mã để chuyển giao cho lớp trợ giúp Observable.

Chúng ta hãy bắt đầu với lớp trợ giúp Observable...



QuackCó thể quan sát

Observable triển khai tất cả các chức năng mà Quackable cần để trở thành một observable.

Chúng ta chỉ cần đưa nó vào một lớp và để lớp đó chuyển giao cho Observable.

```
lớp công khai Observable triển khai QuackObservable {
    Người quan sát ArrayList = new ArrayList();
    QuackVịt quan sát được;

    công khai Observable(QuackObservable vịt) {
        this.duck = vịt;
    }

    public void registerObserver(Người quan sát quan sát) {
        observers.add(người quan sát);
    }

    công khai void thông báo cho người quan sát () {
        Trình lặp iterator = observers.iterator();
        trong khi (iterator.hasNext()) {
            Người quan sát observer = (Người quan sát) iterator.next();
            observer.update(vịt);
        }
    }
}
```

Observable phải triển khai QuackObservable vì đây là những lệnh gọi phương thức

giống nhau sẽ được chuyển giao cho nó.

Tronh hàm tạo, chúng ta truyền vào QuackObservable đang sử dụng đối tượng này để quản lý hành vi quan sát được của nó. Hãy xem phương thức notify() bên dưới; bạn sẽ thấy rằng khi có thông báo, Observable sẽ truyền đối tượng này để người quan sát biết đối tượng nào đang kêu.

Sau đây là mã để đăng ký người quan sát.

Và mã để thực hiện thông báo.

Nay giờ chúng ta hãy xem lớp Quackable sử dụng trình tự này như thế nào...

quack decorators cũng là những thứ có thể quan sát được

⑯ Tích hợp trình trợ giúp Observable với các lớp Quackable.

Điều này không quá tệ. Tất cả những gì chúng ta cần làm là đảm bảo các lớp Quackable được tạo thành từ một Observable và chúng biết cách chuyển giao cho Observable. Sau đó, chúng đã sẵn sàng để trở thành Observable. Đây là triển khai của MallardDuck; các duck khác cũng giống vậy.

```
lớp công khai MallardDuck triển khai Quackable {  
    Có thể quan sát được;
```

Mỗi Quackable có một biến thể hiện Observable.

```
    công khai MallardDuck() {  
        observable = new Observable(cái này);  
    }
```

Trong hàm tạo, chúng ta tạo một Observable và truyền cho nó một tham chiếu đến đối tượng MallardDuck.

```
    công khai void quack() {  
        System.out.println("Quack");  
        thông báo cho người quan sát();  
    }
```

Khi chúng ta kêu quacking, chúng ta cần phải cho người quan sát biết về điều đó.

```
    public void registerObserver(Người quan sát quan sát) {  
        observable.registerObserver(người quan sát);  
    }
```

Đây là hai phương thức QuackObservable của chúng tôi. Lưu ý rằng chúng tôi chỉ ủy quyền cho trình trợ giúp.



Chuốt bút chì của bạn

Chúng ta chưa thay đổi việc triển khai một Quackable, trình trang trí QuackCounter. Chúng ta cần biến nó thành một Observable nữa. Tại sao bạn không viết cái đó:

- ⑯ Chúng ta gần hoàn thành rồi! Chúng ta chỉ cần làm việc ở phía Observer của mẫu.

Chúng tôi đã triển khai mọi thứ chúng tôi cần cho Observables; bây giờ chúng tôi cần một số Observer. Chúng tôi sẽ bắt đầu với giao diện Observer:



Giao diện Observer chỉ có một phương thức là `update()`, được truyền vào `QuackObservable` đang kêu.

```
giao diện công khai Observer {
    public void update(QuackObservable duck);
}
```

Bây giờ chúng ta cần một Người quan sát: những nhà Quackologist kia đâu rồi?!

Chúng ta cần triển khai giao diện Observable, nếu không chúng ta sẽ không thể đăng ký với `QuackObservable`.



```
lớp công khai Quackologist triển khai Observer {

    public void update(QuackObservable duck) {
        System.out.println("Quackologist: " + vịt +
    }
}
```



"chí kêu quắc quắc thôi."

Quackologist rất đơn giản; nó chỉ có một phương thức là `update()`, in ra `Quackable` vừa phát ra tiếng kêu.

hợp chất bầy đàn cũng có thể quan sát được



Chuốt bút chì của bạn

Nếu một Quackologist muốn quan sát toàn bộ một đàn thì sao? Dù sao thì điều đó có nghĩa là gì? Hãy nghĩ về nó như thế này: nếu chúng ta quan sát một hợp chất, thì chúng ta đang quan sát mọi thứ trong hợp chất đó. Vì vậy, khi bạn đăng ký với một đàn, hợp chất đàn sẽ đảm bảo rằng bạn được đăng ký với tất cả các con của nó (xin lỗi, tất cả những con chim quacker nhỏ của nó), có thể bao gồm các đàn khác.

Hãy tiếp tục và viết mã quan sát Flock trước khi chúng ta đi xa hơn...

mẫu hợp chất

- (18) Chúng ta đã sẵn sàng để quan sát. Hãy cập nhật trình mô phỏng và thử nghiệm:

```

lớp công khai DuckSimulator {
    public static void main(String[] args) {
        Trình mô phỏng DuckSimulator = new DuckSimulator();
        Tóm tắtDuckFactory duckFactory = new CountingDuckFactory();

        simulator.simulate(duckFactory);
    }

    void mô phỏng(AbstractDuckFactory duckFactory) {
        // tạo nhà máy sản xuất vịt và vịt ở đây

        // tạo đàn ở đây

        System.out.println("\nTrình mô phỏng vịt: Với Observer");
        Quackologist quackologist = Quackologist mới();
        flockOfDucks.registerObserver(bác sĩ thú y);

        mô phỏng(bầy vịt);

        System.out.println("\nNhững con vịt kêu " + QuackCounter.getQuacks() + " lần");
    }

    void mô phỏng(Vịt có thể quack) {
        vịt.quack();
    }
}

```

Tất cả những gì chúng ta làm ở đây là tạo ra một Quackologist và giao cho anh ta nhiệm vụ quan sát đàn chim.

Lần này chúng ta sẽ mô phỏng toàn bộ đàn.

Hãy thử xem nó hoạt động thế nào nhé!

kết thúc của con vịt

Đây là phần kết lớn. Năm, không, sáu mẫu đã kết hợp lại với nhau để tạo nên Duck Simulator tuyệt vời này. Không nói thêm nữa, chúng tôi xin giới thiệu DuckSimulator!

Cửa sổ chỉnh sửa tệp Trợ giúp DucksAreEverywhere

```
% java DuckSimulator
Duck Simulator: Với Observer
Lang băm
Quackologist: Việt dâu đỏ vừa kêu quạc.
Kwak
Quackologist: Duck Call vừa kêu quạc.

Tiếng kêu cát kèt
Quackologist: Việt cao su chỉ kêu quạc thôi.

Tiếng còi
Nhà nghiên cứu về loài ngỗng: Con ngỗng già làm vịt chỉ kêu quạc.
Lang băm
Nhà nghiên cứu về loài vịt: Việt trời chỉ kêu quạc.
Lang băm
Nhà nghiên cứu về loài vịt: Việt trời chỉ kêu quạc.
Lang băm
Nhà nghiên cứu về loài vịt: Việt trời chỉ kêu quạc.
Vịt kêu 7 lần.

%
```

Sau mỗi tiếng kêu, bắt kè là tiếng kêu nào, người quan sát đều nhận được thông báo.

Và bác sĩ lang băm vẫn đếm được.

không có
Những câu hỏi ngớ ngẩn

H: Vậy đây là một mô hình hợp chất?

H: Vậy vẻ đẹp thực sự của Thiết kế Mẫu là tôi có thể lấy một vấn đề và bắt đầu áp dụng các mẫu cho đến khi tìm ra giải pháp. Đúng không?

A: Không, đây chỉ là một tập hợp các mẫu làm việc cùng nhau. Một mẫu hợp chất là một tập hợp một số mẫu được kết hợp để giải quyết một vấn đề chung. Chúng ta sắp xem xét mẫu hợp chất Model-View-Controller; đó là một tập hợp một số mẫu đã được sử dụng nhiều lần trong nhiều giải pháp thiết kế.

A: Sai rồi. Chúng tôi đã trải qua điều này bài tập với Ducks để cho bạn thấy các mẫu có thể hoạt động cùng nhau như thế nào. Bạn sẽ không bao giờ thực sự muốn tiếp cận một thiết kế như chúng tôi vừa làm. Trên thực tế, có thể có các giải pháp cho các phần của trình mô phỏng vit mà một số mẫu này là quá mức cần thiê

Đôi khi chỉ cần sử dụng các nguyên tắc thiết kế hướng đối tượng tốt là có thể tự mình giải quyết được vấn đề.

Chúng ta sẽ nói thêm về điều này trong chương tiếp theo, nhưng bạn chỉ muốn áp dụng các mẫu khi và nơi chúng có ý nghĩa. Bạn không bao giờ muốn bắt đầu với ý định sử dụng các mẫu chỉ vì mục đích đó. Bạn nên coi thiết kế của DuckSimulator là gương én và phản tảo.

Nhưng này, nó rất vui và giúp chúng ta có ý tưởng hay về cách kết hợp nhiều mẫu hình và một giải pháp.

Chúng tôi đã làm gì?

Chúng tôi bắt đầu với một loạt Quackables...

Một con ngỗng đi ngang qua và muốn hành động giống như Quackable. Vì vậy, chúng tôi đã sử dụng Adapter Pattern để điều chỉnh con ngỗng thành Quackable. Bây giờ, bạn có thể gọi quack() trên một con ngỗng được bọc trong bộ điều hợp và nó sẽ kêu!

Sau đó, Quackologists quyết định họ muốn thêm quacks. Vì vậy, chúng tôi đã sử dụng Decorator Pattern để thêm một decorator QuackCounter theo dõi số lần quack() được gọi, sau đó chuyển quacks cho Quackable đang bao bọc nó.

Nhưng Quackologists lo lắng rằng họ sẽ quên thêm trình trang trí QuackCounter. Vì vậy, chúng tôi đã sử dụng Abstract Factory Pattern để tạo ra những chú vịt cho họ. Bây giờ, bất cứ khi nào họ muốn một chú vịt, họ sẽ yêu cầu nhà máy cung cấp một chú, và nhà máy sẽ trả lại một chú vịt được trang trí. (Và đừng quên, họ cũng có thể sử dụng một nhà máy vịt khác nếu họ muốn một chú vịt không được trang trí!)

Chúng tôi gặp vấn đề về quản lý khi theo dõi tất cả những con vịt, ngỗng và những con quackable. Vì vậy, chúng tôi đã sử dụng Composite Pattern để nhóm những con quackable vào Flocks. Pattern này cũng cho phép quackologist tạo ra các sub-Flocks để quản lý các họ vịt. Chúng tôi đã sử dụng Iterator Pattern trong quá trình triển khai của mình bằng cách sử dụng iterator của java.util trong ArrayList.

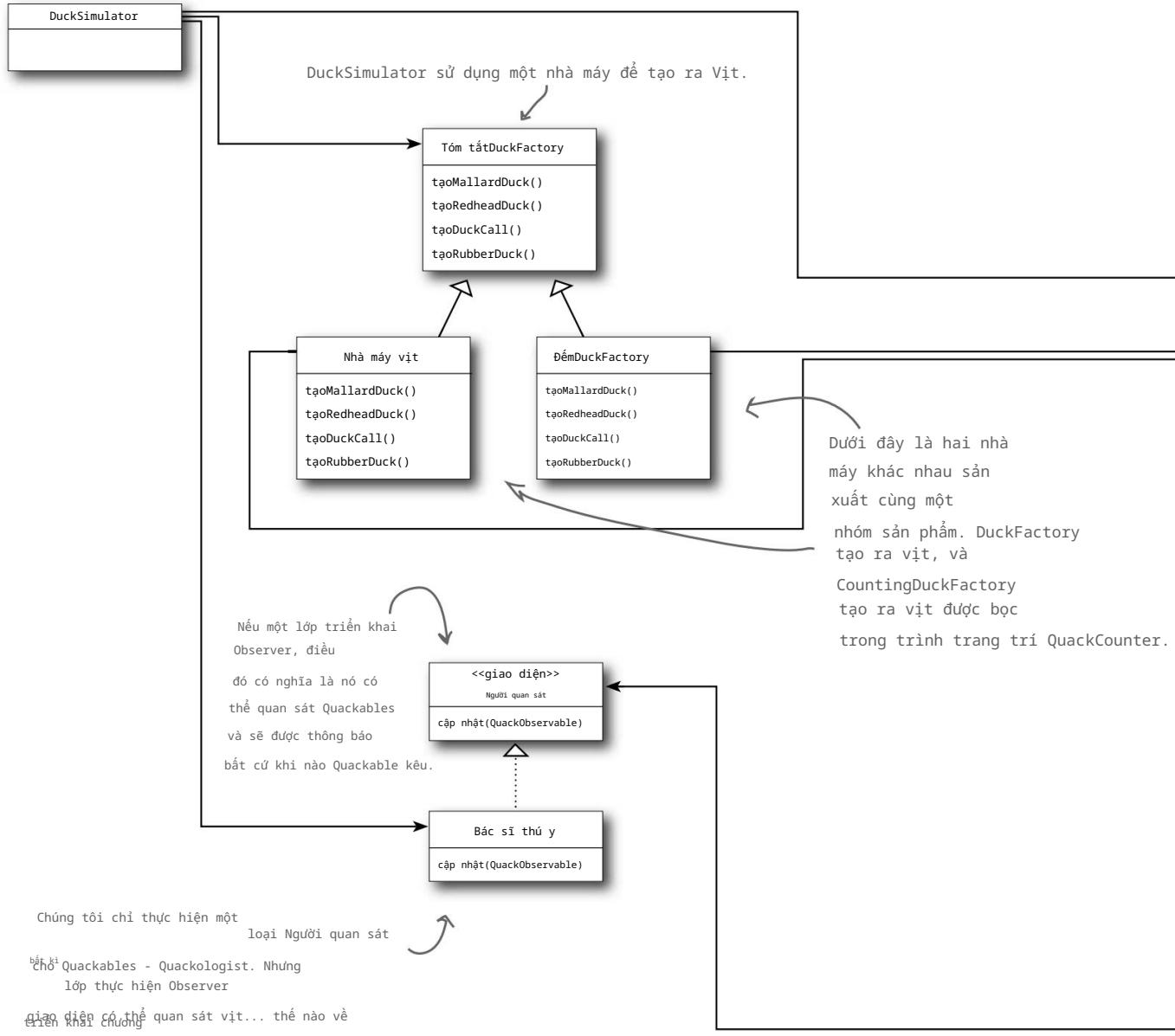
Quackologists cũng muốn được thông báo khi bất kỳ quackable nào kêu quack. Vì vậy, chúng tôi đã sử dụng Observer Pattern để cho Quackologists đăng ký là Quackable Observer. Bây giờ họ được thông báo mỗi khi bất kỳ Quackable nào kêu quack. Chúng tôi đã sử dụng iterator một lần nữa trong triển khai này. Quackologists thậm chí có thể sử dụng Observer Pattern với các hợp chất của họ.



góc nhìn của con vịt

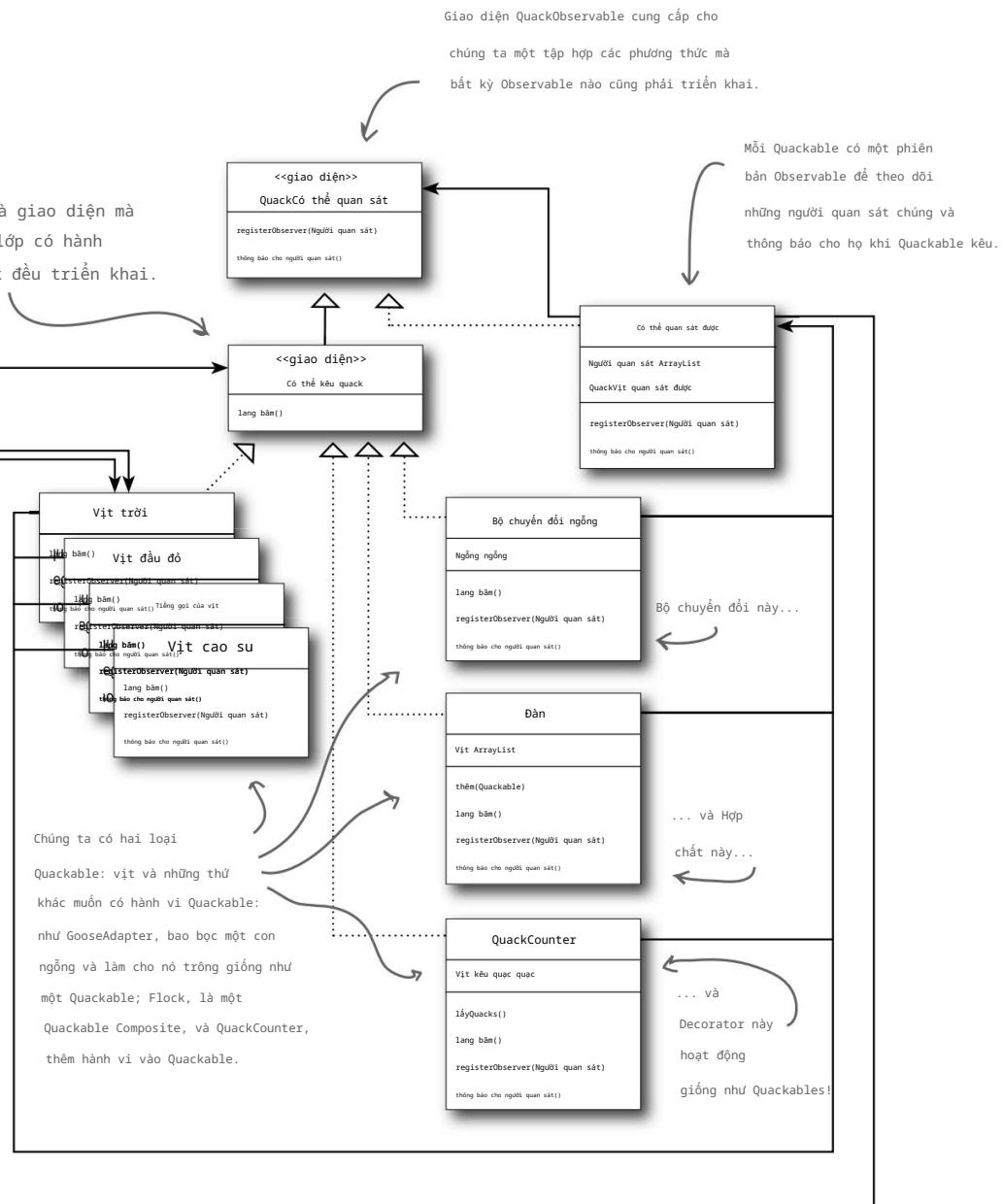
Góc ~~nhìn~~ từ trên cao của một chú chim: sơ đồ lớp

Chúng tôi đã đóng gói rất nhiều mẫu vào một mô phỏng vịt nhỏ! Đây là hình ảnh lớn về những gì chúng tôi đã làm:



mẫu hợp chất

Quackable là giao diện mà tất cả các lớp có hành vi kêu quack đều triển khai.



bài hát bộ điều khiển chế độ xem mô hình

Ông vua của các mẫu hợp chất Nếu Elvis
là một mẫu hợp chất, tên của ông ấy sẽ là Model-View-Controller, và ông
ấy sẽ hát một bài hát nhỏ như thế này...

Model, View, Controller

Lời và nhạc của James Dempsey.

MVC là mô hình phân chia mã của bạn thành các phần đoạn chức năng, giúp bạn không bị rối trí.

Để đạt được khả năng tái sử dụng, bạn phải giữ những ranh giới sạch sẽ

Mô hình ở một bên, Chế độ xem ở bên kia,
Bộ điều khiển ở giữa.

Bạn có thể mô hình hóa một bướm ga và một ống phân phổi

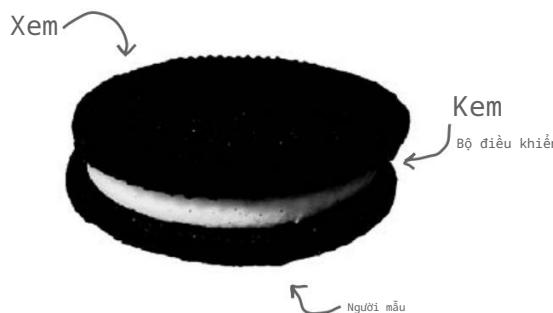
Mô hình dáng đi của một đứa trẻ hai tuổi

Mô hình một chai Chardonnay hảo hạng

Mô hình tất cả các điểm dừng thanh quản mọi người nói

Làm mẫu việc áp trúng luộc

Bạn có thể mô phỏng dáng đi lạch bạch của Hexley



Model View, nó có ba lớp giống như Oreos

Bộ điều khiển chế độ xem mô hình

Chế độ xem mô hình, Chế độ xem mô hình, Bộ điều khiển chế độ xem mô hình

Các đối tượng mô hình đại diện cho lý do tồn tại của ứng dụng của bạn Các đối tượng tùy chỉnh chứa dữ liệu, logic, v.v. Bạn tạo các lớp tùy chỉnh, trong miền vấn đề của ứng dụng, bạn có thể chọn sử dụng lại chúng với tất cả các chế độ xem nhưng các đối tượng mô hình vẫn giữ nguyên.

tái sử dụng

Chế độ xem mô hình, tất cả được hiển thị rất đẹp trong màu xanh nước biển

Bộ điều khiển chế độ xem mô hình

Có lẽ bây giờ bạn đang tự hỏi

Bạn có lẽ đang tự hỏi làm thế nào

Luồng dữ liệu giữa Model và View

Người kiểm soát phải làm trung gian

Giao trạng thái thay đổi của mỗi lớp

Để đồng bộ hóa dữ liệu của hai

mẫu hợp chất

Nó kéo và đẩy mọi giá trị đã thay đổi

Tôi đã gửi một TextField StringValue.

Model View, xin gửi lời khen ngợi chân thành đến nhóm smalltalk!

Mô hình xem

Bộ điều khiển chế độ xem mô hình

Làm sao chúng ta có thể đào sâu tất cả keo đó

Bộ điều khiển chế độ xem mô hình

Model View, phát âm là Oh Oh chứ không phải Ooo Ooo

Bộ điều khiển chế độ xem mô hình

Họ thường sử dụng mã hóa cứng có thể gây ảnh hưởng đến khả năng tái sử dụng

Câu chuyện này còn một chút gì đó chưa kể

Nhưng bây giờ bạn có thể kết nối mỗi khóa mô hình mà bạn chọn với bất kỳ thuộc tính chế độ xem nào

Thêm vài dặm nữa trên con đường này

Có vẻ như không ai nhận được nhiều vinh quang

Từ việc viết mã điều khiển

Và một khi bạn bắt đầu ràng buộc

Vâng, nhiệm vụ quan trọng của mô hình

Tôi nghĩ bạn sẽ thấy ít mã hơn trong cây nguồn của bạn

Và quang cảnh thật tuyệt đẹp

Vâng, tôi biết tôi rất vui mừng vì những thứ họ đã tự động hóa và những thứ bạn

Tôi có thể lười biếng, nhưng đôi khi nó thật điện rồ

nhận được miễn phí

Bao nhiêu mã tôi viết chỉ là keo dán

Và tôi nghĩ bạn nên lặp lại toàn bộ mã mà

Và nó sẽ không bị thám đến thế

bạn sẽ không cần khi kết nối nó vào IB.

Nhưng mã không làm được điều kỳ diệu

Nó chỉ là di chuyển các giá trị thông qua



Sử dụng Swing.

Và tôi không có ý định trở nên đặc ác

Model View, thậm chí còn xử lý nhiều lựa chọn nữa

Nhưng nó trở nên lặp đi lặp lại

Bộ điều khiển chế độ xem mô hình

Làm tắt cả những việc mà người điều khiển làm

Model View, tôi cá là tôi sẽ gửi đơn đăng ký của mình trước bạn

Và tôi ước mình có một xu

Bộ điều khiển chế độ xem mô hình

Cho mỗi lần duy nhất



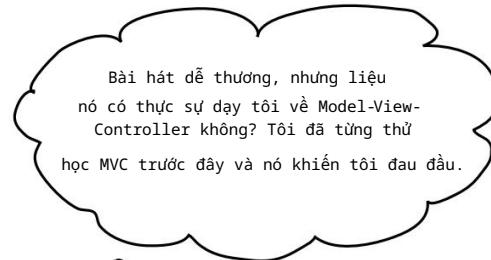
Sức mạnh của tai

Đừng chỉ đọc! Sau cùng thì đây là một cuốn sách Head First... hãy cầm iPod của bạn lên,

nhấp vào URL này: <http://www.headfirstlabs.com/books/hfdp/media.html>

Hãy ngồi xuống và lắng nghe nhé.

mvc là các mẫu được ghép lại với nhau



Không. Mẫu thiết kế là chìa
khóa để bạn bước vào MVC.

Chúng tôi chỉ muốn kích thích sự thèm ăn của bạn thôi.
Tôi nói cho bạn biết, sau khi đọc xong chương
này, hãy quay lại và nghe lại bài hát một lần
nữa - bạn sẽ thấy vui hơn nữa.

Nghe có vẻ như bạn đã từng gặp rắc rối với MVC trước
đây? Hầu hết chúng ta đều đã từng. Bạn có thể đã
nghe những nhà phát triển khác nói với bạn rằng nó
đã thay đổi cuộc sống của họ và có thể tạo ra hòa bình
thế giới. Chắc chắn đó là một mô hình hợp chất
mạnh mẽ và mặc dù chúng ta không thể khẳng định nó sẽ
tạo ra hòa bình thế giới, nhưng nó sẽ giúp bạn tiết kiệm
hàng giờ viết mã khi bạn biết điều đó.

Nhưng trước tiên bạn phải học nó, đúng
không? Vâng, lần này sẽ có sự khác biệt lớn
vì bây giờ bạn đã biết các mẫu rồi!

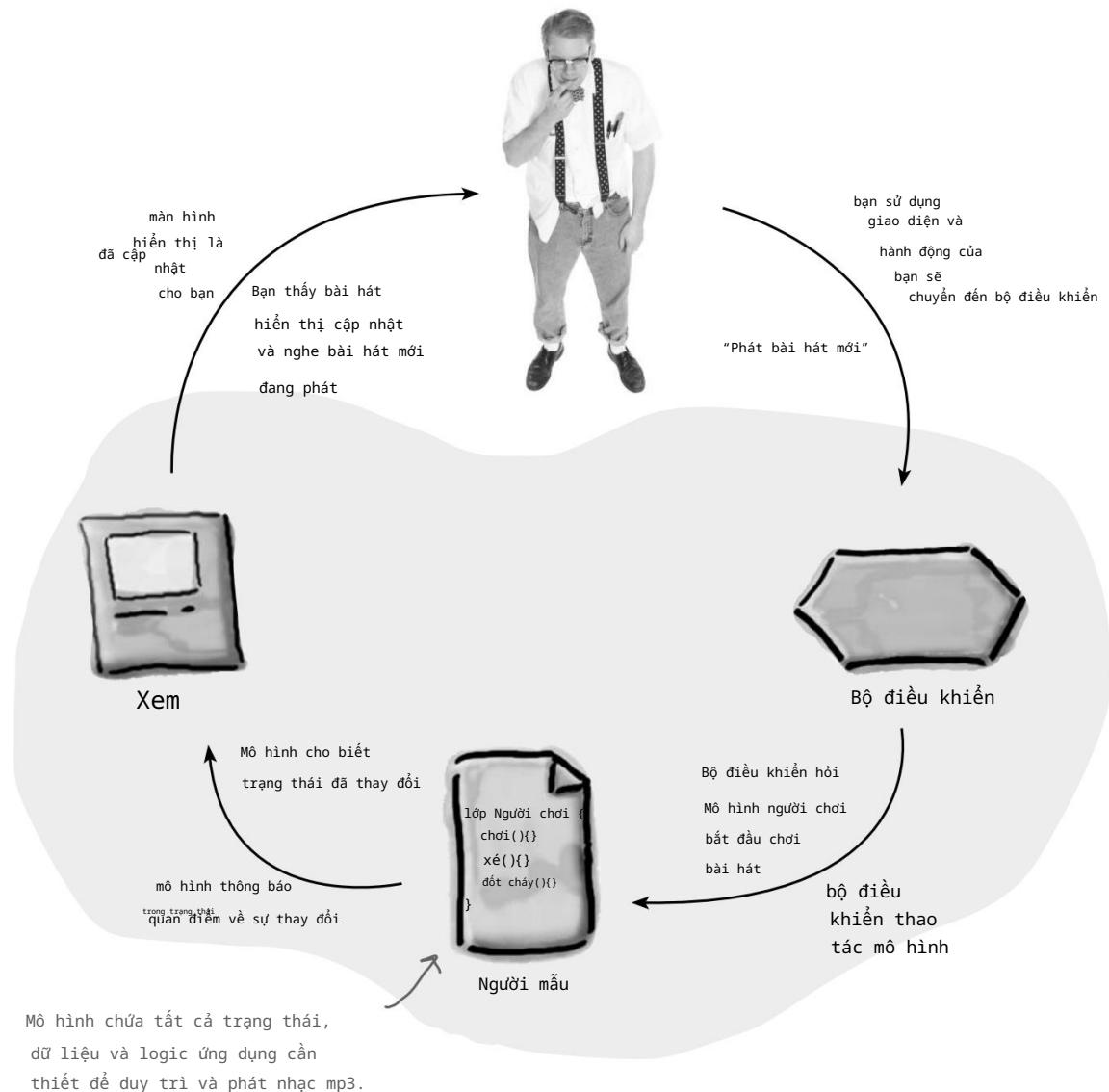
Đúng vậy, mô hình chính là chìa khóa của MVC.
Học MVC từ trên xuống dưới rất khó; không nhiều nhà phát
triển thành công. Đây là bí quyết để học MVC: nó
chỉ là một vài mẫu được ghép lại với nhau. Khi bạn
tiếp cận việc học MVC bằng cách xem xét các mẫu,
đột nhiên nó bắt đầu có ý nghĩa.

Hãy bắt đầu thôi. Lần này bạn sẽ nắm vững
MVC!

Gặp gỡ Model-View-Controller

Hãy tưởng tượng bạn đang sử dụng trình phát MP3 yêu thích của mình, như iTunes. Bạn có thể sử dụng giao diện của trình phát để thêm bài hát mới, quản lý danh sách phát và đổi tên bài hát. Trình phát sẽ đảm nhiệm việc duy trì một cơ sở dữ liệu nhỏ về tất cả các bài hát của bạn cùng với tên và dữ liệu liên quan. Trình phát cũng đảm nhiệm việc phát các bài hát và khi thực hiện, giao diện người dùng liên tục được cập nhật với tiêu đề bài hát hiện tại, thời lượng phát, v.v.

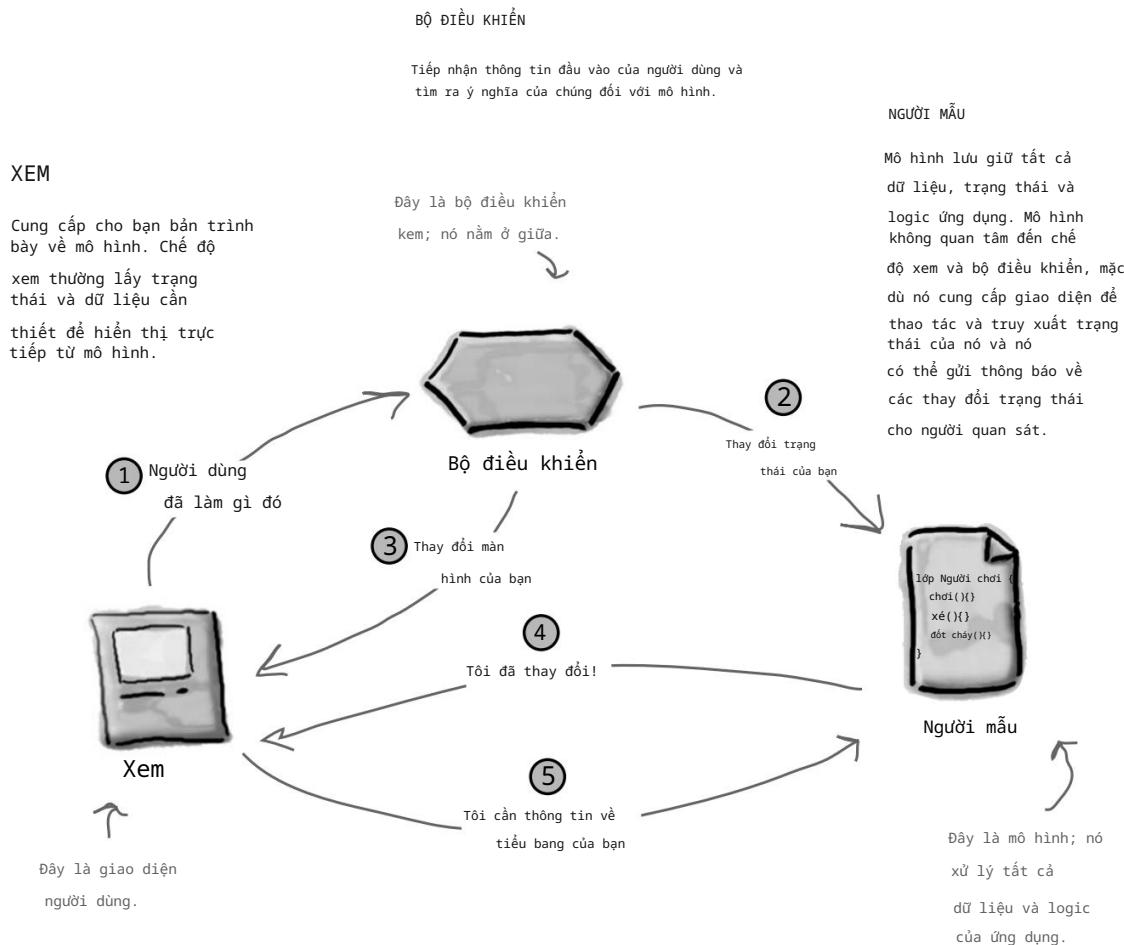
Vâng, bên dưới tắt cả là Model-View-Controller...



mvc cận cảnh

Nhìn kỹ hơn...

Mô tả về Trình phát MP3 cung cấp cho chúng ta góc nhìn tổng quan về MVC, nhưng nó thực sự không giúp bạn hiểu được bản chất của cách thức hoạt động của mẫu hợp chất, cách bạn tự xây dựng một mẫu hoặc lý do tại sao nó lại tốt như vậy. Hãy bắt đầu bằng cách tìm hiểu từng bước về mối quan hệ giữa mô hình, chế độ xem và bộ điều khiển, sau đó chúng ta sẽ xem xét lại từ góc nhìn của Mẫu thiết kế.



- ① Bạn là người dùng – bạn tương tác với chế độ xem.
View là cửa sổ của bạn tới mô hình. Khi bạn làm gì đó với view (như nhấp vào nút Play) thì view sẽ cho bộ điều khiển biết bạn đã làm gì. Nhiệm vụ của bộ điều khiển là xử lý điều đó.
- ② Bộ điều khiển yêu cầu mô hình thay đổi trạng thái của nó.
Bộ điều khiển tiếp nhận hành động của bạn và diễn giải chúng. Nếu bạn nhấp vào một nút, nhiệm vụ của bộ điều khiển là tìm ra ý nghĩa của nút đó và cách mô hình nên được điều khiển dựa trên hành động đó.
- ③ Bộ điều khiển cũng có thể yêu cầu thay đổi chế độ xem.
Khi bộ điều khiển nhận được hành động từ chế độ xem, nó có thể cần phải yêu cầu chế độ xem thay đổi theo kết quả. Ví dụ, bộ điều khiển có thể bật hoặc tắt một số nút hoặc mục menu trong giao diện.
- ④ Mô hình sẽ thông báo cho chế độ xem khi trạng thái của nó thay đổi.
Khi có điều gì đó thay đổi trong mô hình, dựa trên một số hành động bạn thực hiện (như nhấp vào nút) hoặc một số thay đổi nội bộ khác (như bài hát tiếp theo trong danh sách phát đã bắt đầu), mô hình sẽ thông báo cho chế độ xem rằng trạng thái của nó đã thay đổi.
- ⑤ Chế độ xem yêu cầu mô hình cung cấp trạng thái.
View lấy trạng thái hiển thị trực tiếp từ model. Ví dụ, khi model thông báo cho view rằng một bài hát mới đã bắt đầu phát, view sẽ yêu cầu tên bài hát từ model và hiển thị nó. View cũng có thể yêu cầu model cung cấp trạng thái khi controller yêu cầu một số thay đổi trong view.

không có Những câu hỏi ngớ ngẩn

Q: Bộ điều khiển có bao giờ trở thành

người quan sát mô hình?

A: Chắc chắn. Trong một số thiết kế, bộ điều khiển đăng ký với mô hình và được thông báo về các thay đổi. Trường hợp này có thể xảy ra khi một cái gì đó trong mô hình ảnh hưởng trực tiếp đến các điều khiển giao diện người dùng. Ví dụ, một số trạng thái nhất định trong mô hình có thể chỉ định một số mục giao diện được bật hoặc tắt. Nếu vậy, thì nhiệm vụ thực sự của bộ điều khiển là yêu cầu chế độ xem cập nhật màn hình hiển thị của nó cho phù hợp.

Q: Tất cả những gì bộ điều khiển làm là lấy thông tin người dùng

dầu vào từ chế độ xem và gửi đến mô hình, đúng không? Tại sao lại có nó nếu đó là tất cả những gì nó làm? Tại sao không chỉ có mã trong chính chế độ xem? Trong hầu hết các trường hợp, bộ điều khiển không chỉ gọi một phương thức trên mô hình sao?

A: Bộ điều khiển làm nhiều hơn

chỉ cần "gửi nó đến mô hình", bộ điều khiển có trách nhiệm diễn giải dữ liệu đầu vào và thao tác mô hình dựa trên dữ liệu đầu vào đó. Nhưng câu hỏi thực sự của bạn có lẽ là "tại sao tôi không thể làm điều đó trong mã xem?"

Bạn có thể; tuy nhiên, bạn không muốn làm như vậy vì hai lý do: Đầu tiên, bạn sẽ làm phức tạp mã xem của mình vì giờ nó có hai trách nhiệm: quản lý giao diện người dùng và xử lý logic về cách điều khiển mô hình. Thứ hai, bạn đang liên kết chặt chẽ chế độ xem của mình với mô hình. Nếu bạn muốn sử dụng lại chế độ xem với một mô hình khác, hãy quên nó đi. Bộ điều khiển tách logic điều khiển khỏi chế độ xem và tách chế độ xem khỏi mô hình. Bằng cách giữ chế độ xem và bộ điều khiển được liên kết lỏng lẻo, bạn đang xây dựng một thiết kế linh hoạt và có thể mở rộng hơn, một thiết kế có thể dễ dàng thích ứng với sự thay đổi trong tương lai.

các mẫu trong mvc

Nhìn MVC qua lăng kính màu mẫu

Chúng tôi đã nói với bạn rằng con đường tốt nhất để học MVC là xem nó như bản chất của nó: một tập hợp các mẫu hoạt động cùng nhau trong cùng một thiết kế.

Hãy bắt đầu với mô hình. Như bạn có thể đoán, mô hình sử dụng Observer để cập nhật các chế độ xem và bộ điều khiển về những thay đổi trạng thái mới nhất.

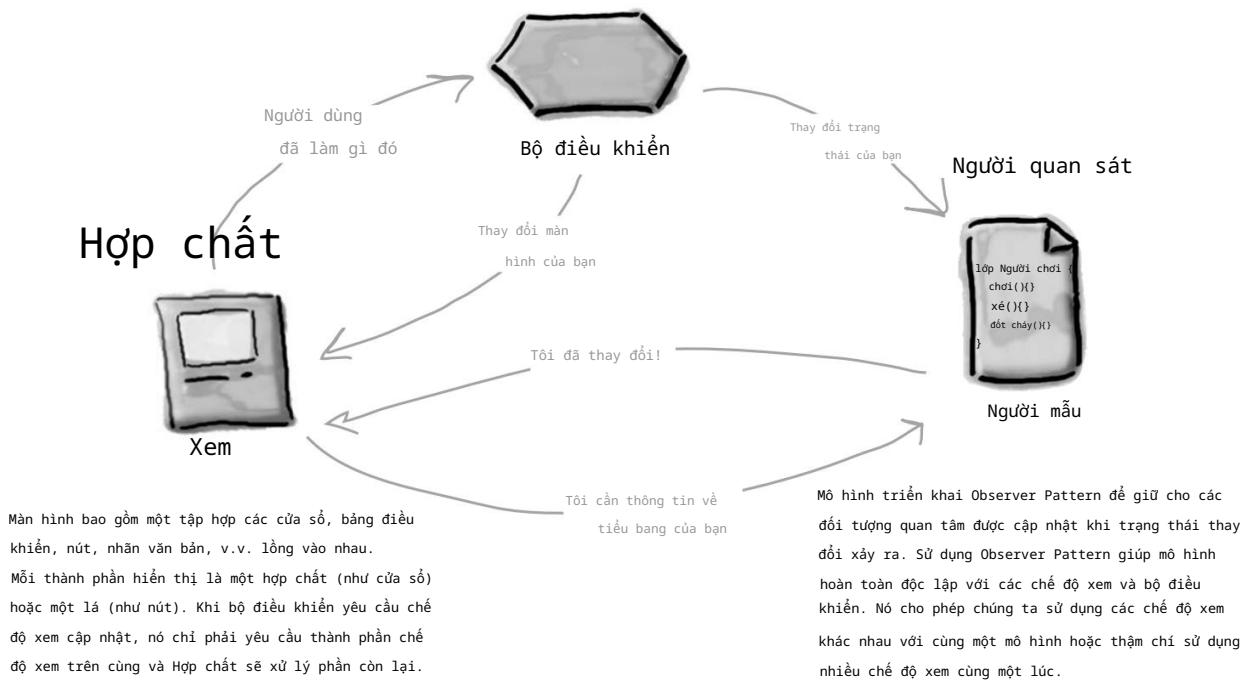
Mặt khác, view và controller triển khai Strategy Pattern. Controller là hành vi của view và có thể dễ dàng trao đổi với một controller khác nếu bạn muốn có hành vi khác. Bên thân view cũng sử dụng một pattern nội bộ để quản lý các cửa sổ, nút và các thành phần khác của màn hình: Composite Pattern.



Chúng ta hãy xem xét kỹ hơn:

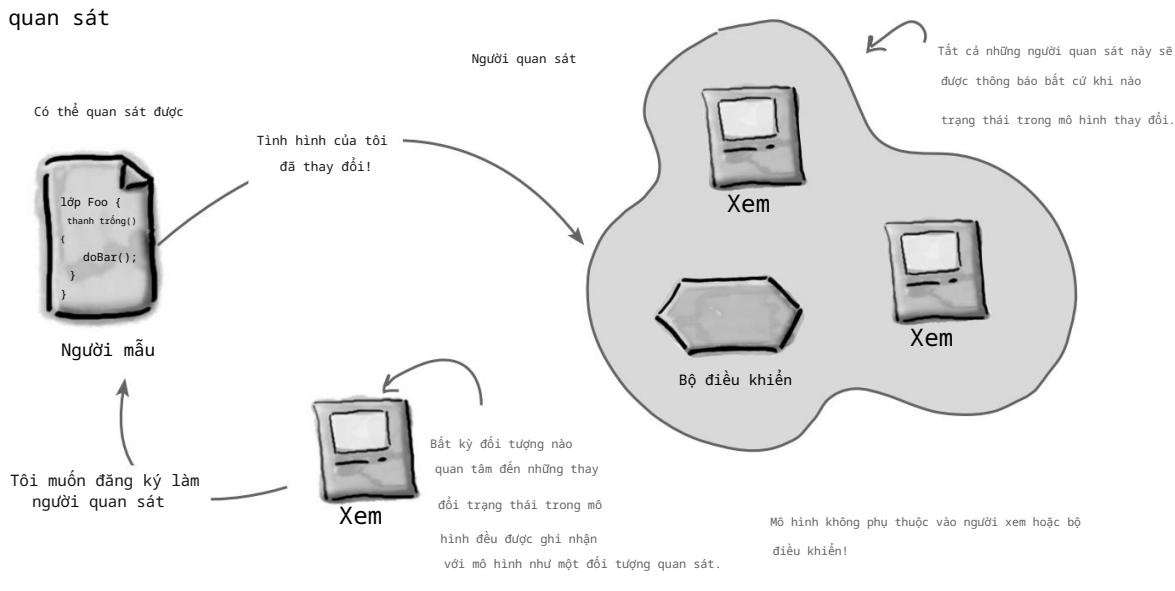
Chiến lược

View và controller triển khai Strategy Pattern cổ điển: view là một đối tượng được cấu hình với một chiến lược. Controller cung cấp chiến lược. View chỉ quan tâm đến các khía cạnh trực quan của ứng dụng và chuyển giao cho controller mọi quyết định về hành vi giao diện. Sử dụng Strategy Pattern cũng giúp view tách biệt khỏi model vì controller chịu trách nhiệm tương tác với model để thực hiện các yêu cầu của người dùng. View không biết gì về cách thực hiện điều này.

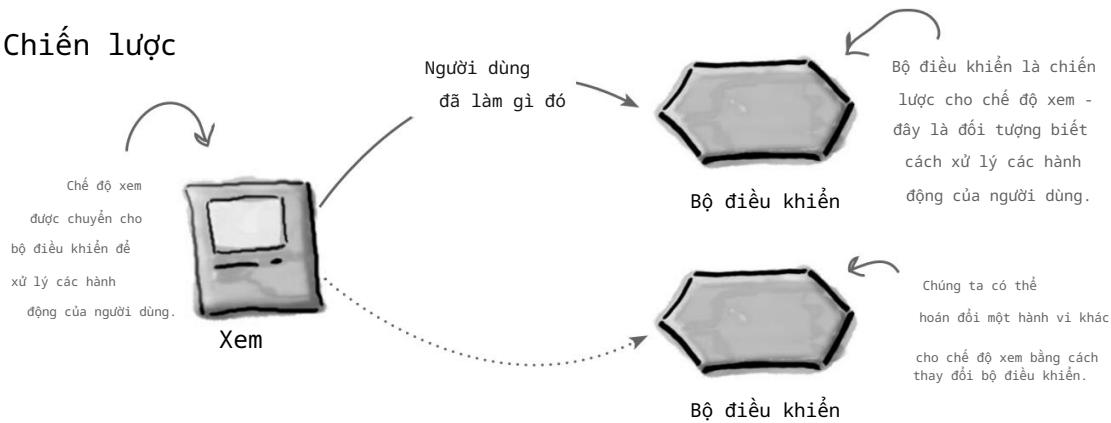


mẫu hợp chất

Người quan sát

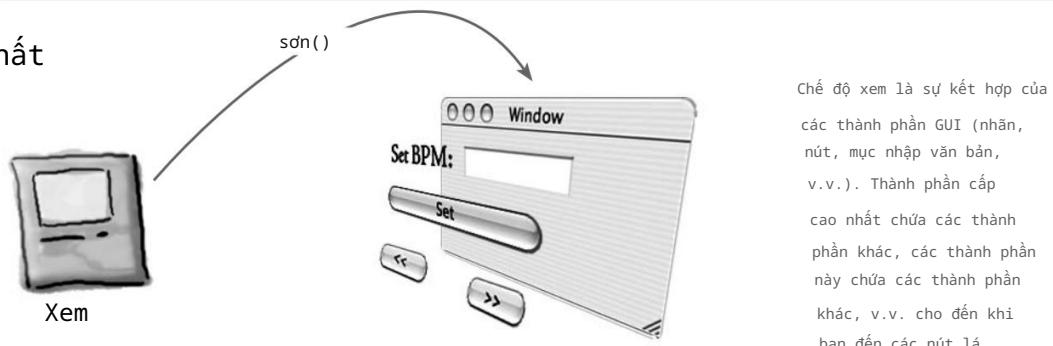


Chiến lược



Phần xem chỉ quan tâm đến việc trình bày, còn phần điều khiển quan tâm đến việc chuyển đổi dữ liệu đầu vào của người dùng thành các hành động trên mô hình.

Hợp chất



mvc và chế độ xem dj

Sử dụng MVC để điều khiển nhịp điệu...

Đã đến lúc bạn trở thành DJ. Khi bạn là một DJ, tất cả là về nhịp điệu. Bạn có thể bắt đầu bản phối của mình bằng một giai điệu chậm rãi, chậm rãi ở 95 nhịp mỗi phút (BPM) và sau đó đưa đám đông lên đến 140 BPM điên cuồng của nhạc trance techno. Bạn sẽ kết thúc buổi biểu diễn của mình bằng một bản phối ambient êm dịu ở 80 BPM.

Bạn sẽ làm điều đó như thế nào? Bạn phải kiểm soát nhịp điệu và bạn sẽ xây dựng công cụ để thực hiện điều đó.



Gặp gỡ Java DJ View

Hãy bắt đầu với chế độ xem của công cụ. Chế độ xem cho phép bạn tạo nhịp trống và điều chỉnh số nhịp trống theo phút...

Chế độ xem có hai phần, phần dùng để xem trạng thái của mô hình và phần dùng để kiểm soát mọi thứ.

Thanh nhịp đập hiển thị nhịp đập theo thời gian thực.

Màn hình hiển thị BPM hiện tại và tự động cài đặt bất cứ khi nào BPM thay đổi.

Bạn có thể nhập BPM cụ thể và nhấp vào nút Cài đặt để cài đặt số nhịp cụ thể mỗi phút hoặc bạn có thể sử dụng các nút tăng và giảm để tinh chỉnh.

Toggle	Giảm BPM	Tăng
	một nhịp mỗi phút.	BPM thêm một nhịp mỗi phút.

mẫu hợp chất

Sau đây là một vài cách khác để kiểm soát DJ View...



Bạn có thể bắt đầu chơi nhạc bằng cách chọn mục menu Bắt đầu trong menu “Điều khiển DJ”.

Lưu ý chức năng Dừng sẽ bị vô hiệu hóa cho đến khi bạn bắt đầu nhịp.

Bạn sử dụng nút Dừng để tắt chế độ tạo nhịp.

Lưu ý Bắt đầu sẽ bị vô hiệu hóa sau khi nhịp đã bắt đầu.



Mọi hành động của người dùng đều được gửi đến bộ điều khiển.

Bộ điều khiển nằm ở giữa...

Bộ điều khiển nằm giữa chế độ xem và mô hình. Nó lấy thông tin đầu vào của bạn, như chọn “Bắt đầu” từ menu DJ Control, và biến nó thành hành động trên mô hình để bắt đầu tạo nhịp.

Bộ điều khiển tiếp nhận thông tin đầu vào từ người dùng và tìm cách chuyển thông tin đó thành các yêu cầu trên mô hình.



Bộ điều khiển

Chúng ta đừng quên mô hình bên dưới nhé...

Bạn không thể nhìn thấy mô hình, nhưng bạn có thể nghe thấy nó. Mô hình nằm bên dưới mọi thứ khác, quản lý nhịp điệu và điều khiển loa bằng MIDI.

BeatModel là trái tim của ứng dụng. Nó thực hiện logic để bắt đầu và dừng nhịp, thiết lập số nhịp mỗi phút (BPM) và tạo ra âm thanh.

Bvà ^{Tôi}_{TRÊN()} ^{ngày}_{và}

đặtBPM() tắt()

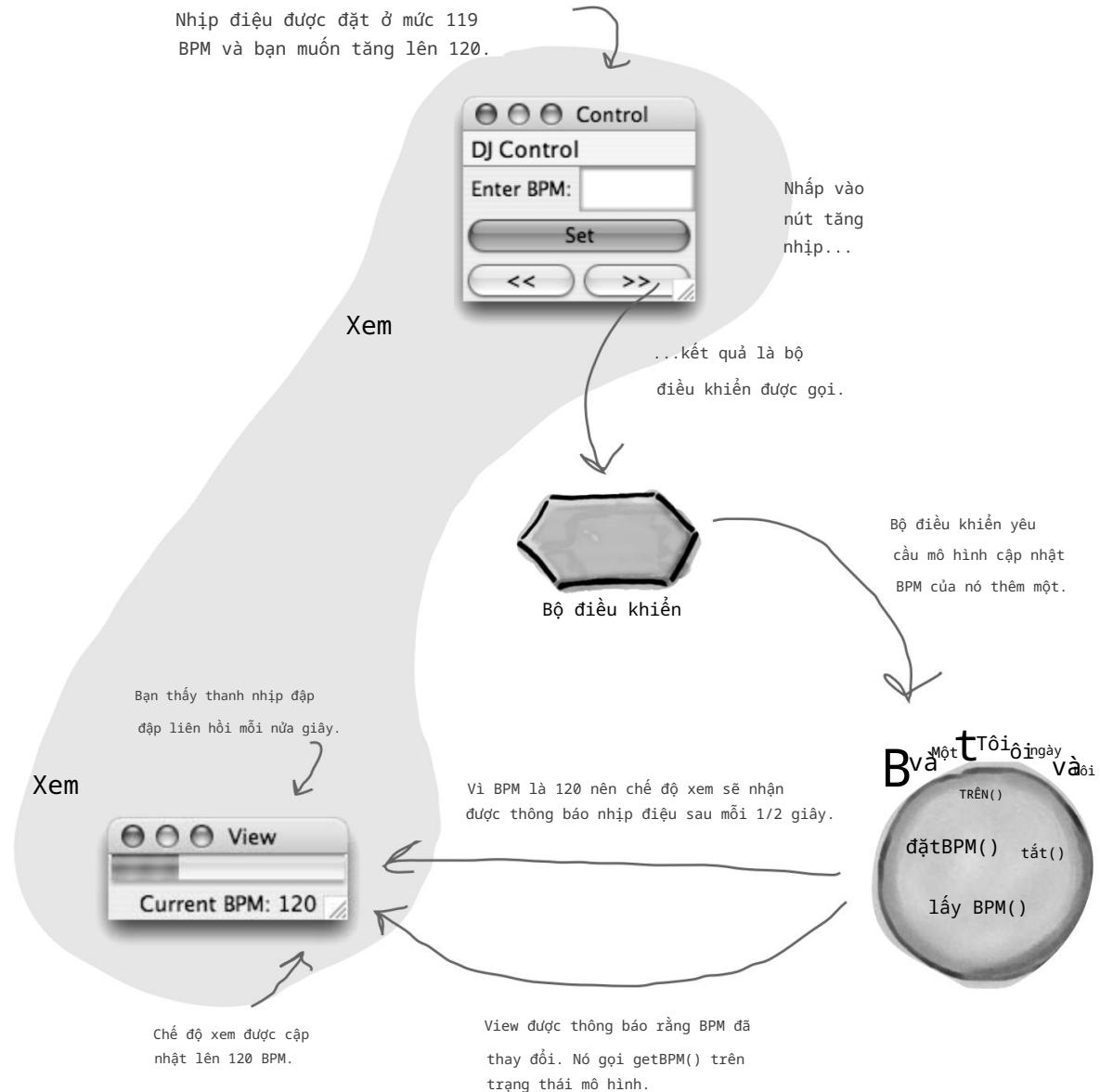
lấy BPM()

Mô hình này cũng cho phép chúng ta lấy trạng thái hiện tại thông qua phương thức getBPM().



mô hình dj , chế độ xem và bộ điều khiển

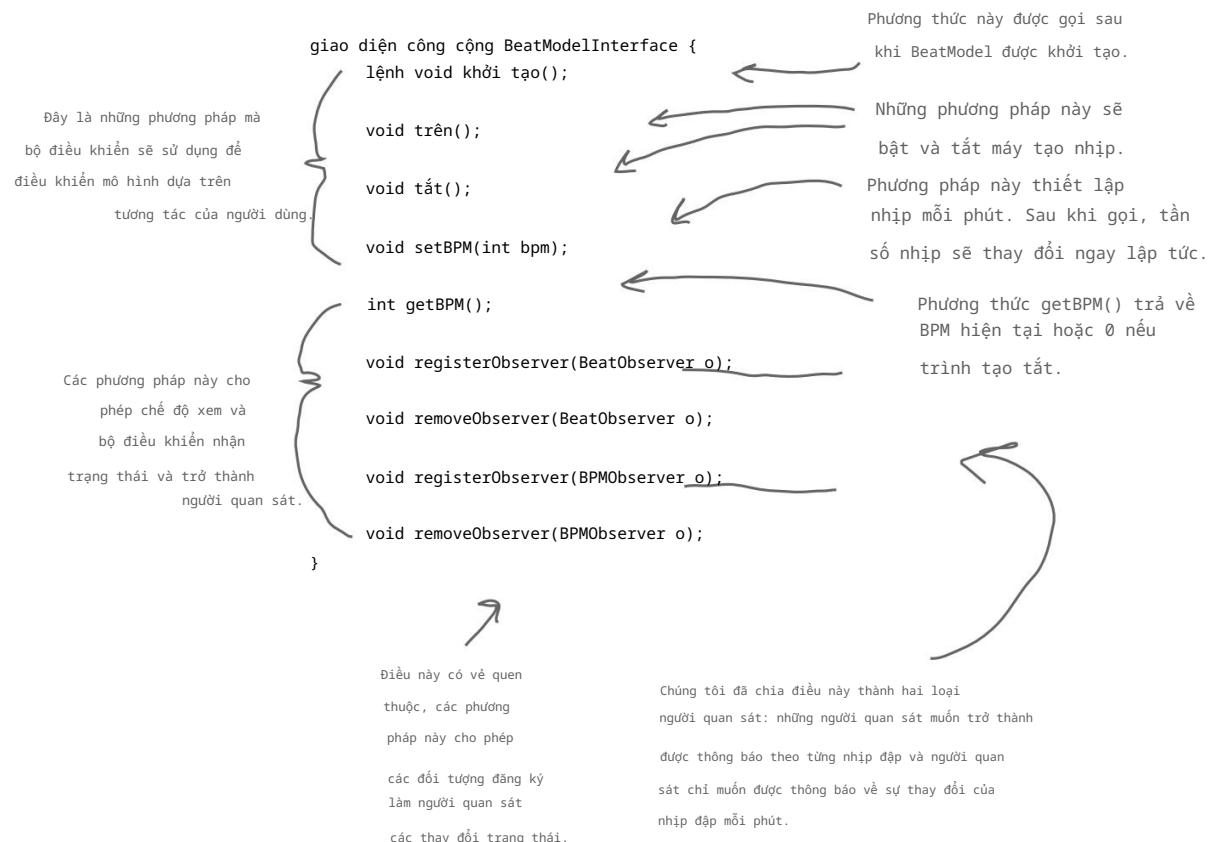
Ghép các mảnh lại với nhau



Xây dựng các mảng ghép

Được rồi, bạn biết mô hình chịu trách nhiệm duy trì tất cả dữ liệu, trạng thái và bắt kỳ logic ứng dụng nào. Vậy BeatModel có gì trong đó? Công việc chính của nó là quản lý nhịp, vì vậy nó có trạng thái duy trì nhịp hiện tại mỗi phút và rất nhiều mã tạo ra các sự kiện MIDI để tạo ra nhịp mà chúng ta nghe. Nó cũng hiển thị một giao diện cho phép bộ điều khiển thao tác nhịp và cho phép chế độ xem và bộ điều khiển lấy trạng thái của mô hình. Ngoài ra, đừng quên rằng mô hình sử dụng Observer Pattern, vì vậy chúng ta cũng cần một số phương pháp để cho phép các đối tượng đăng ký làm người quan sát và gửi thông báo.

Hãy cùng xem qua BeatModelInterface trước khi xem xét phần triển khai:



mô hình nhịp đập

Bây giờ chúng ta hãy xem xét lớp BeatModel cụ thể:

```

Chúng tôi triển khai BeatModelInterface.
Điều này là cần thiết cho mã MIDI.

lớp công khai BeatModel triển khai BeatModelInterface, MetaEventListener {
    Máy sáp xếp trình tự;
    ArrayList beatObservers = new ArrayList();
    ArrayList bpmObservers = new ArrayList();
    int bpm = 90;
    // các biến thể hiện khác ở đây

    công khai void khởi tạo() {
        thiết lậpMidi();
        xây dựngTrackAndStart();
    }

    công khai void trên() {
        sequencer. bắt đầu();
        đặtBPM(90);
    }

    công khai void tắt() {
        đặtBPM(0);
        sequencer. dừng();
    }

    công khai void setBPM(int bpm) {
        nhịp này.bpm = nhịp/phút;
        bộ sáp xếp.setTempoInBPM(getBPM());
        thông báo cho BPMObservers();
    }

    công khai int getBPM() {
        trả về bpm;
    }

    void beatEvent() {
        thông báo choBeatObservers();
    }

    // Mã để đăng ký và thông báo cho người quan sát
    // Rất nhiều mã MIDI để xử lý nhịp điệu
}

```

Điều này là cần thiết cho mã MIDI.

Bộ sáp xếp là vật thể biết cách tạo ra nhịp điệu thực (mà bạn có thể nghe thấy!).

Các ArrayList này chứa hai loại trình quan sát (trình quan sát Beat và BPM).

Biến thể hiện bpm giữ tần số nhịp - theo mặc định là 90 BPM.

Phương pháp này thiết lập trên trình sáp xếp và thiết lập các bản nhạc cho chúng ta.

Phương thức on() khởi động trình sáp xếp và đặt BPM theo mặc định: 90 BPM.

Và off() sẽ tắt nó bằng cách đặt BPM thành 0 và dừng trình sáp xếp.

Phương thức setBPM() là cách bộ điều khiển thao tác nhịp. Nó thực hiện ba điều:

- (1) Đặt biến thể hiện bpm
- (2) Yêu cầu bộ sáp xếp thay đổi BPM của nó.
- (3) Thông báo cho tất cả Người quan sát BPM rằng BPM đã thay đổi.

Phương thức getBPM() chỉ trả về biến thể hiện bpm, cho biết số nhịp hiện tại mỗi phút.

Phương thức beatEvent(), không có trong BeatModelInterface, được gọi bởi mã MIDI bất cứ khi nào một nhịp mới bắt đầu. Phương thức này thông báo cho tất cả BeatObservers rằng một nhịp mới vừa xảy ra.



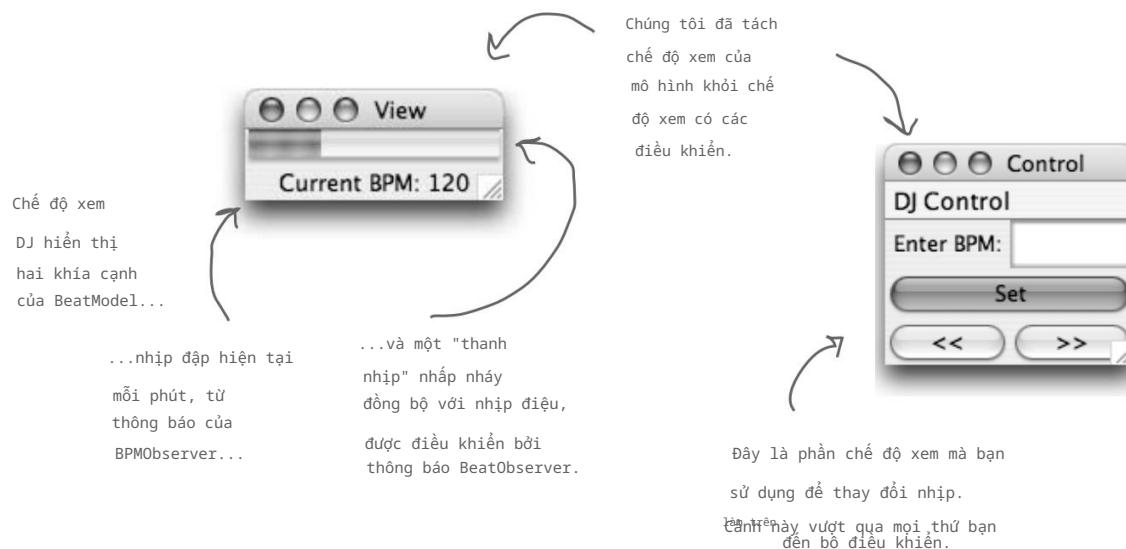
Mã săn sàng nướng

Mô hình này sử dụng hỗ trợ MIDI của Java để tạo nhịp. Bạn có thể kiểm tra triển khai đầy đủ của tất cả các lớp DJ trong các tệp nguồn Java có sẵn trên trang headfirstlabs.com hoặc xem mã ở cuối chương.

Quan điểm

Bây giờ niềm vui bắt đầu; chúng ta có thể kết nối chế độ xem và hình dung BeatModel!

Điều đầu tiên cần lưu ý về chế độ xem là chúng tôi đã triển khai nó để nó được hiển thị trong hai cửa sổ riêng biệt. Một cửa sổ chứa BPM hiện tại và xung; cửa sổ còn lại chứa các điều khiển giao diện. Tại sao? Chúng tôi muốn nhấn mạnh sự khác biệt giữa giao diện chứa chế độ xem của mô hình và phần còn lại của giao diện chứa tập hợp các điều khiển của người dùng. Chúng ta hãy xem xét kỹ hơn hai phần của chế độ xem:



não Apower

BeatModel của chúng tôi không đưa ra bất kỳ giả định nào về chế độ xem. Mô hình được triển khai bằng Observer Pattern, do đó, nó chỉ thông báo cho bất kỳ chế độ xem nào được đăng ký là người quan sát khi trạng thái của nó thay đổi. Chế độ xem sử dụng API của mô hình để truy cập vào trạng thái. Chúng tôi đã triển khai một loại chế độ xem, bạn có thể nghĩ ra các chế độ xem khác có thể sử dụng thông báo và trạng thái trong BeatModel không?

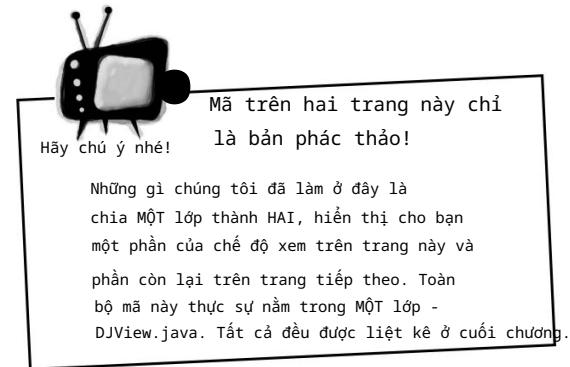
Một chương trình biểu diễn ánh sáng dựa trên nhịp điệu thời gian thực.

Chế độ xem văn bản hiển thị thể loại nhạc dựa trên BPM (nhạc ambient, nhạc downbeat, nhạc techno, v.v.).

quan điểm của dj

Triển khai View

Hai phần của chế độ xem - chế độ xem của mô hình và chế độ xem có các điều khiển giao diện người dùng - được hiển thị trong hai cửa sổ, nhưng cùng tồn tại trong một lớp Java. Trước tiên, chúng tôi sẽ chỉ cho bạn mã tạo chế độ xem của mô hình, hiển thị BPM hiện tại và thanh nhịp. Sau đó, chúng tôi sẽ quay lại trang tiếp theo và chỉ cho bạn mã tạo ra các điều khiển giao diện người dùng, hiển thị trường nhập văn bản BPM và các nút.



```
lớp công khai DJView triển khai ActionListener, BeatObserver, BPMObserver {
    Mô hình giao diện BeatModel;
    Bộ điều khiển Giao diện bộ điều khiển;
    JFrame xem Khung;
    JPanel xemBảng điều khiển;
    Thanh BeatBar;
    Nhãn JbmOutputLabel;
```

View giữ tham chiếu đến cả model và controller. Controller chỉ được sử dụng bởi giao diện điều khiển, chúng ta sẽ xem xét sau...
Ở đây, chúng ta tạo một số thành phần cho màn hình.

```
public DJView(ControllerInterface controller, BeatModelInterface model) { this.controller = controller;
    this.model = mô hình;
    model.registerObserver((BeatObserver)này);
    model.registerObserver((BPMObserver)này);
}
```

Hàm tạo sẽ lấy tham chiếu đến bộ điều khiển và mô hình, và chúng tôi lưu trữ các tham chiếu đến chúng trong các biến thể hiện.

```
công khai void createView() {
    // Tạo tất cả các thành phần Swing ở đây
}
```

Chúng tôi cũng đăng ký làm BeatObserver và BPMObserver của mô hình.

```
công khai void updateBPM() {
    int bpm = model.getBPM();
    nếu (bpm == 0) {
        bpmOutputLabel.setText("ngoại tuyến");
    } khác {
        bpmOutputLabel.setText("BPM hiện tại:
    }
}
```

Phương thức updateBPM() được gọi khi có sự thay đổi trạng thái xảy ra trong mô hình. Khi điều đó xảy ra, chúng ta cập nhật màn hình hiển thị với BPM hiện tại. Chúng ta có thể lấy giá trị này bằng cách yêu cầu trực tiếp từ mô hình.

```
công khai void updateBeat() {
    beatBar. setValue(100);
}
}
```

Tương tự như vậy, phương thức updateBeat() được gọi khi mô hình bắt đầu một nhịp mới. Khi điều đó xảy ra, chúng ta cần tạo xung cho "beat bar" của mình. Chúng ta thực hiện điều này bằng cách đặt nó ở giá trị tối đa (100) và để nó xử lý hoạt ảnh của xung.

Triển khai View, tiếp theo...

Bây giờ, chúng ta sẽ xem mã cho phần điều khiển giao diện người dùng của chế độ xem. Chế độ xem này cho phép bạn điều khiển mô hình bằng cách cho bộ điều khiển biết phải làm gì, sau đó, bộ điều khiển sẽ cho mô hình biết phải làm gì. Hãy nhớ rằng, mã này nằm trong cùng một tệp lớp với mã chế độ xem khác.

```

lớp công khai DJView triển khai ActionListener, BeatObserver, BPMObserver {
    Mô hình giao diện BeatModel;
    Bộ điều khiển Giao diện bộ điều khiển;
    Nhãn Jbm;
    Trưởng văn bản JTextField;
    JButton đặtBPMButton;
    JButton tăng BPMButton;
    JButton giảmBPMButton;
    JMenuBar thanh menu;
    Trình đơn JMenu;
    Bắt đầu MenuItem;
    Dừng MenuItem;

    công khai void createControls() {
        // Tạo tất cả các thành phần Swing ở đây
    }
    công khai void enableStopMenuItem() {
        stopMenuItem.setEnabled(true);
    }

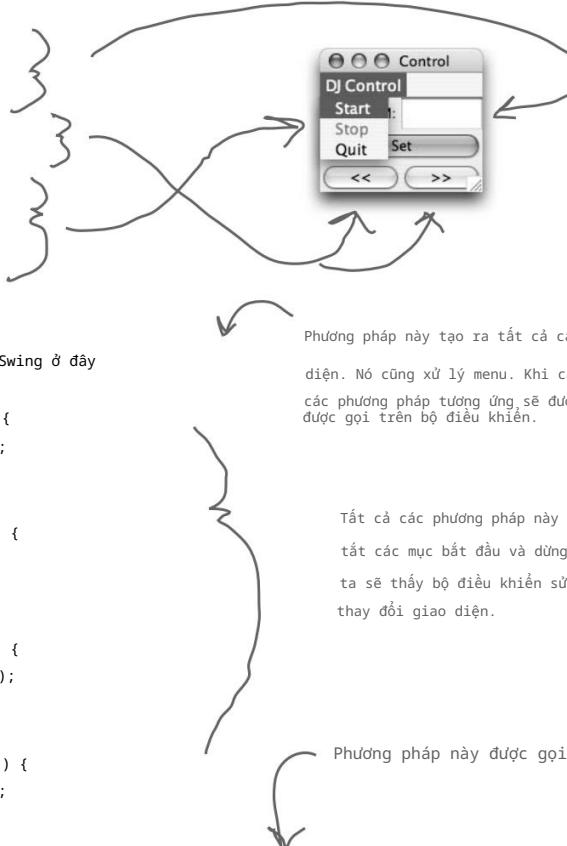
    công khai void disableStopMenuItem() {
        stopMenuItem.setEnabled(false);
    }

    công khai void enableStartMenuItem() {
        startMenuItem.setEnabled(true);
    }

    công khai void disableStartMenuItem() {
        startMenuItem.setEnabled(false);
    }

    public void actionPerformed(sự kiện.ActionEvent) {
        nếu (event.getSource() == setBPMButton) {
            int bpm = Integer.parseInt(bpmTextField.getText());
            bộ điều khiển. setBPM(bpm);
        } nếu không thì nếu (event.getSource() == increaseBPMButton) {
            bộ điều khiển. tăng BPM();
        } nếu không thì nếu (event.getSource() == giảmBPMButton) {
            bộ điều khiển. giảm BPM();
        }
    }
}

```



Phương pháp này tạo ra tất cả các điều khiển và đặt chúng vào giao diện. Nó cũng xử lý menu. Khi các mục dừng hoặc bắt đầu được chọn, các phương pháp tương ứng sẽ được gọi trên bộ điều khiển.

Tất cả các phương pháp này cho phép bật và tắt các mục bắt đầu và dừng trong menu. Chúng ta sẽ thấy bộ điều khiển sử dụng các mục này để thay đổi giao diện.

Phương pháp này được gọi khi nhấp vào một nút.

Nếu nhấp vào nút Set
thì nút này sẽ được
chuyển đến bộ điều khiển
cùng với nhịp bpm mới.

Tương tự như vậy, nếu
nhấp vào nút tăng hoặc
giảm, thông tin này sẽ
được truyền đến bộ điều khiển.

bộ điều khiển dj

Bây giờ đến bộ điều khiển

Đã đến lúc viết phần còn thiếu: bộ điều khiển. Hãy nhớ rằng bộ điều khiển là chiến lược mà chúng ta đưa vào chế độ xem để cung cấp cho nó một số tính năng thông minh.

Vì chúng ta đang triển khai Strategy Pattern, chúng ta cần bắt đầu bằng một giao diện cho bất kỳ Strategy nào có thể được cắm vào DJ View. Chúng ta sẽ gọi nó là ControllerInterface.

```
giao diện công khai ControllerInterface {
    lệnh bắt đầu();
    lệnh dừng();
    hàm void tăng BPM();
    hàm giảmBPM();
    void setBPM(int bpm);
}
```

Sau đây là tất cả các phương thức mà chế độ xem có thể gọi trên bộ điều khiển.

Những thứ này trông quen thuộc sau khi xem giao diện của mô hình. Bạn có thể dừng và bắt đầu tạo nhịp và thay đổi BPM. Giao diện này "phong phú" hơn giao diện BeatModel vì bạn có thể điều chỉnh BPM bằng cách tăng và giảm.



Thiết kế câu đố

Bạn đã thấy rằng view và controller cùng sử dụng Strategy Pattern. Bạn có thể vẽ sơ đồ lớp của hai thành phần này để biểu diễn cho pattern này không?

Và đây là cách triển khai bộ điều khiển:

```

lớp công khai BeatController thực hiện ControllerInterface {
    Mô hình giao diện BeatModel;
    Chế độ xem DJView;

    công khai BeatController(BeatModelInterface mô hình) {
        this.model = mô hình;
        view = new DJView(cái này, mô hình);
        view.createView();
        view.createControls();
        xem. vô hiệu hóaStopMenuItem();
        view.enableStartMenuItem();
        model.khởi tạo();
    }

    công khai void bắt đầu() {
        mô hình.on();
        view. vô hiệu hóaStartMenuItem();
        xem. enableStopMenuItem();
    }

    công khai void dừng() {
        mô hình. tắt();
        xem. vô hiệu hóaStopMenuItem();
        view.enableStartMenuItem();
    }

    công khai void increaseBPM() {
        int bpm = model.getBPM();
        mô hình.setBPM(bpm + 1);
    }

    công khai void giảmBPM() {
        int bpm = model.getBPM();
        mô hình.setBPM(bpm - 1);
    }

    công khai void setBPM(int bpm) {
        mô hình. setBPM(bpm);
    }
}

```

Bộ điều khiển triển khai ControllerInterface.

Bộ điều khiển là phần mềm nằm ở giữa cookie MVC Oreo, do đó, nó là đối tượng giữ lại chế độ xem và mô hình và kết nối tất cả lại với nhau.

Bộ điều khiển được truyền cho mô hình trong hàm tạo và sau đó tạo ra chế độ xem.

Khi bạn chọn Bắt đầu từ menu giao diện người dùng, bộ điều khiển sẽ bật mô hình và sau đó thay đổi giao diện người dùng để mục menu Bắt đầu bị vô hiệu hóa và mục menu Dừng được bật.

Tương tự như vậy, khi bạn chọn Dừng từ menu, bộ điều khiển sẽ tắt mô hình và thay đổi giao diện người dùng để mục menu dừng bị vô hiệu hóa và mục menu bắt đầu được bật.

Nếu nhấp vào nút tăng, bộ điều khiển sẽ lấy BPM hiện tại từ mô hình, thêm một BPM, sau đó đặt BPM mới.

Tương tự như vậy, chỉ khác là chúng ta trừ một vào BPM hiện tại.

Cuối cùng, nếu giao diện người dùng được sử dụng để thiết lập BPM tùy ý, bộ điều khiển sẽ hướng dẫn mô hình thiết lập BPM của nó.

LƯU Ý: bộ điều khiển đang đưa ra quyết định thông minh cho chế độ xem.

Chế độ xem chỉ biết cách bật và tắt các mục menu; không biết trường hợp nào cần tắt chúng.

kết hợp tất cả lại với nhau

Tổng hợp tất cả lại...

Chúng ta đã có mọi thứ cần thiết: mô hình, chế độ xem và bộ điều khiển.

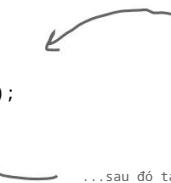
Bây giờ là lúc đưa tất cả chúng vào MVC! Chúng ta sẽ xem và nghe chúng hoạt động tốt như thế nào khi kết hợp với nhau.

Tất cả những gì chúng ta cần là một đoạn mã nhỏ để bắt đầu; không mất nhiều thời gian đâu:

```
lớp công khai DJTestDrive {
    public static void main (String[] args) {
        Mô hình giao diện BeatModel = new BeatModel();
        ControllerInterface controller = new BeatController(model);
    }
}
```



Đầu tiên hãy tạo một mô hình...



...sau đó tạo một bộ điều khiển và
truyền cho nó mô hình. Hãy nhớ rằng, bộ
điều khiển tạo ra chế độ xem, vì vậy
chúng ta không cần phải làm điều đó.

Và bây giờ là lúc chạy thử nghiệm...

```
Cửa sổ chỉnh sửa tệp Trợ giúp LetTheBassKick
% java DJTestDrive
%
```



Chạy cái này...

...và bạn sẽ thấy điều này.



Những việc cần làm

- 1 Bắt đầu tạo nhịp bằng mục menu Bắt đầu; lưu ý bộ điều khiển sẽ vô hiệu hóa mục này sau đó.
- 2 Sử dụng mục nhập văn bản cùng với các nút tăng và giảm để thay đổi BPM. Lưu ý cách hiển thị chế độ xem phản ánh các thay đổi mặc dù không có liên kết logic với các bộ điều khiển.
- 3 Lưu ý thanh nhịp luôn theo kịp nhịp vì nó là đơn vị quan sát mô hình.
- 4 Bật bài hát yêu thích của bạn và xem bạn có thể bắt nhịp theo bài hát đó bằng cách sử dụng nút tăng và giảm nhịp hay không.
- 5 Dừng máy phát điện. Lưu ý cách bộ điều khiển vô hiệu hóa mục menu Dừng và bật mục menu Bắt đầu.

mẫu hợp chất

Khám phá chiến lược

Hãy cùng tìm hiểu thêm về Strategy Pattern để hiểu rõ hơn về cách sử dụng nó trong MVC. Chúng ta sẽ thấy một pattern thân thiện khác cũng xuất hiện - một pattern mà bạn thường thấy trong bộ ba MVC: Adapter Pattern.

Hãy nghĩ một giây về chức năng của DJ View: nó hiển thị nhịp độ và mạch độ. Nghe có vẻ giống thứ gì khác không? Thê còn nhịp tim thì sao? Tình cờ chúng ta có một lớp học về máy theo dõi nhịp tim; đây là sơ đồ lớp học:



Chúng tôi có phương pháp để lấy được nhịp tim hiện tại.
 Và may mắn thay, các nhà phát triển của nó biết về giao diện Beat và BPM Observer!

não Apower

Chắc chắn sẽ rất tuyệt nếu chúng ta có thể sử dụng lại chế độ xem hiện tại của mình với HeartModel, nhưng chúng ta cần một bộ điều khiển hoạt động với mô hình này. Ngoài ra, giao diện của HeartModel không khớp với những gì chế độ xem mong đợi vì nó có phương thức getHeartRate() thay vì getBPM(). Bạn sẽ thiết kế một tập hợp các lớp như thế nào để cho phép chế độ xem được sử dụng lại với mô hình mới?

MVC và bộ điều hợp

Điều chỉnh mô hình

Để bắt đầu, chúng ta sẽ cần điều chỉnh HeartModel thành BeatModel. Nếu không, view sẽ không thể hoạt động với model, vì view chỉ biết cách getBPM(), và phương thức tương đương của heart model là getHeartRate(). Chúng ta sẽ làm điều này như thế nào? Tất nhiên là chúng ta sẽ sử dụng Adapter Pattern! Hóa ra đây là một kỹ thuật phổ biến khi làm việc với MVC: sử dụng một adapter để điều chỉnh model để hoạt động với các controller và view hiện có.

Sau đây là mã để điều chỉnh HeartModel thành BeatModel:

```
lớp công khai HeartAdapter triển khai BeatModelInterface {
    Giao diện HeartModel trái tim;

    công khai HeartAdapter(HeartModelInterface trái tim) {
        this.heart = trái tim;
    }
    công khai void khởi tạo() {}

    công khai void trên() {}

    công khai void off() {}

    công khai int getBPM() {
        trả về heart.getHeartRate();
    }

    công khai void setBPM(int bpm) {}

    công khai void registerObserver(BeatObserver o) {
        heart.registerObserver(o);
    }

    công khai void removeObserver(BeatObserver o) {
        heart.removeObserver(o);
    }

    public void registerObserver(BPMObserver o) {
        heart.registerObserver(o);
    }

    công khai void removeObserver(BPMObserver o) {
        heart.removeObserver(o);
    }
}
```

Chúng ta cần triển khai giao diện mục tiêu, trong trường hợp này là BeatModelInterface.

Tại đây, chúng ta lưu trữ tham chiếu tới mô hình trái tim.

Chúng ta không biết những thứ này sẽ làm gì với trái tim, nhưng nghe có vẻ đáng sợ. Vậy nên chúng ta sẽ để chúng ở trạng thái "không hoạt động".

Khi getBPM() được gọi, chúng ta sẽ dịch nó thành lệnh gọi getHeartRate() trên mô hình tim.

Chúng tôi không muốn làm điều này trên trái tim!
Một lần nữa, chúng ta hãy để nó ở trạng thái "không hoạt động".

Sau đây là các phương pháp quan sát của chúng tôi.
Chúng tôi chỉ giao chúng cho mô hình trái tim được gói lại.

mẫu hợp chất

Bây giờ chúng ta đã sẵn sàng cho HeartController

Với HeartAdapter trong tay, chúng ta đã sẵn sàng tạo một bộ điều khiển và chạy chế độ xem với HeartModel. Hãy nói về việc tái sử dụng!

```
lớp công khai HeartController thực hiện ControllerInterface {
    Mô hình giao diện HeartModel;
    Chế độ xem DJView;

    public HeartController(HeartModelInterface model) {
        this.model = mô hình;
        view = new DJView(cái này, new HeartAdapter(mô hình));
        view.createView();
        view.createControls();
        xem. vô hiệu hóaStopMenuItem();
        view. vô hiệu hóaStartMenuItem();
    }

    công khai void start() {}

    công khai void dừng() {}

    công khai void increaseBPM() {}

    công khai void giảmBPM() {}

    công khai void setBPM(int bpm) {}
}
```

HeartController triển khai ControllerInterface, giống như BeatController đã làm.

Giống như trước, bộ điều khiển tạo ra chế độ xem và kết nối mọi thứ lại với nhau.

Có một thay đổi: chúng ta được truyền một HeartModel, không phải BeatModel...

...và chúng ta cần bọc mô hình đó bằng một bộ điều hợp trước khi đưa nó vào chế độ xem.

Cuối cùng, HeartController vô hiệu hóa các mục menu vì chúng không cần thiết.

Không có nhiều việc để làm ở đây; xét cho cùng, chúng ta không thể thực sự kiểm soát được trái tim như cách chúng ta có thể đánh bại máy móc.

Và thế là xong! Bây giờ là lúc thử nghiệm mǎ...

```
lớp công khai HeartTestDrive {
    public static void main (String[] args) {
        HeartModel heartModel = new HeartModel();
        Mô hình ControllerInterface = new HeartController(heartModel);
    }
}
```

Tất cả những gì chúng ta cần làm là tạo bộ điều khiển và truyền cho nó một màn hình theo dõi nhịp tim.

kiểm tra mô hình trái tim

Và bây giờ là lúc chạy thử nghiệm...



Những việc cần làm

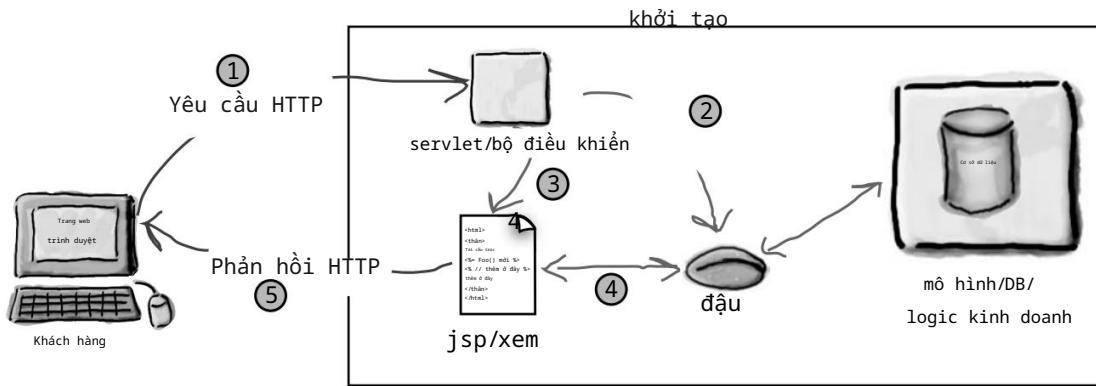
- 1 Lưu ý rằng màn hình hoạt động tốt với hình trái tim!
Thanh nhịp trông giống như một xung. Vì HeartModel cũng hỗ trợ BPM và Beat Observers nên chúng ta có thể nhận được các bản cập nhật nhịp giống như với các nhịp DJ.
- 2 Vì nhịp tim có sự thay đổi tự nhiên nên hãy chú ý màn hình sẽ được cập nhật theo nhịp mới mỗi phút.
- 3 Mỗi khi chúng ta nhận được bản cập nhật BPM, bộ điều hợp sẽ thực hiện nhiệm vụ biên dịch các lệnh gọi getBPM() thành các lệnh gọi getHeartRate().
- 4 Các mục menu Bắt đầu và Dừng không được bật vì bộ điều khiển đã tắt chúng.
- 5 Các nút khác vẫn hoạt động nhưng không có tác dụng vì bộ điều khiển không thực hiện bất kỳ thao tác nào cho chúng.
Có thể thay đổi chế độ xem để hỗ trợ việc vô hiệu hóa các mục này.

Nhịp tim khỏe mạnh.

MVC và Web

Không lâu sau khi Web được tạo ra, các nhà phát triển bắt đầu điều chỉnh MVC để phù hợp với mô hình trình duyệt/máy chủ. Sự điều chỉnh phổ biến được gọi đơn giản là "Mô hình 2" và sử dụng sự kết hợp của công nghệ servlet và JSP để đạt được sự tách biệt giữa mô hình, chế độ xem và bộ điều khiển giống như chúng ta thấy trong GUI thông thường.

Hãy cùng xem Mô hình 2 hoạt động như thế nào:



- ① Bạn thực hiện một yêu cầu HTTP và yêu cầu này được servlet tiếp nhận.

Sử dụng trình duyệt web, bạn thực hiện yêu cầu HTTP. Điều này thường liên quan đến việc gửi một số dữ liệu mẫu, như tên người dùng và mật khẩu của bạn. Một servlet nhận dữ liệu mẫu này và phân tích cú pháp.

- ② Servlet đóng vai trò là bộ điều khiển.

Servlet đóng vai trò là bộ điều khiển và xử lý yêu cầu của bạn, nhiều khả năng là thực hiện yêu cầu trên mô hình (thường là cơ sở dữ liệu). Kết quả của việc xử lý yêu cầu thường được đóng gói dưới dạng JavaBean.

- ③ Bộ điều khiển chuyển tiếp quyền điều khiển đến chế độ xem.

View được biểu diễn bởi JSP. Nhiệm vụ duy nhất của JSP là tạo trang biểu diễn view của model (mà nó lấy được thông qua JavaBean) cùng với bất kỳ điều khiển nào cần thiết cho các hành động tiếp theo.

- ⑤ Chế độ xem trả về một trang cho trình duyệt thông qua HTTP.

Một trang được trả về trình duyệt, nơi nó được hiển thị dưới dạng chế độ xem. Người dùng gửi thêm các yêu cầu, được xử lý theo cùng một cách.

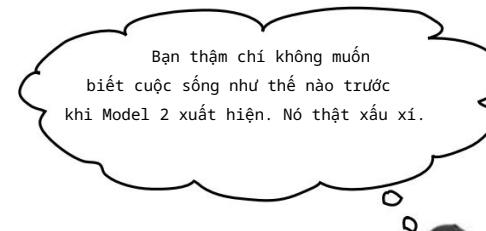
mô hình 2

Model 2 không chỉ có thiết kế
đẹp mắt.

Lợi ích của việc tách biệt view, model và controller khá rõ ràng với bạn hiện nay. Nhưng bạn cần biết "phần còn lại của câu chuyện" với Model 2 - rằng nó đã cứu nhiều cửa hàng web khỏi tình trạng hỗn loạn.

Làm sao? Vâng, Mô hình 2 không chỉ cung cấp sự tách biệt các thành phần về mặt thiết kế, mà còn cung cấp sự tách biệt về trách nhiệm sản xuất. Hãy đối mặt với nó, ngày xưa, bất kỳ ai có quyền truy cập vào JSP của bạn đều có thể vào và viết bất kỳ mã Java nào họ muốn, đúng không? Và điều đó bao gồm rất nhiều người không biết tệp jar từ một lọ bơ đậu phộng. Thực tế là hầu hết các nhà sản xuất web đều biết về nội dung và HTML, không phải phần mềm.

May mắn thay, Model 2 đã đến giải cứu. Với Model 2, chúng ta có thể giao công việc phát triển cho những người biết về Servlet và để các nhà sản xuất web tự do phát triển các JSP theo phong cách Model 2 đơn giản, trong đó tất cả những gì họ có thể truy cập là HTML và JavaBean đơn giản.



cựu nhân viên DOT COM

Mô hình 2: Làm DJ từ điện thoại di động

Bạn không nghĩ tài sè có gắng bỏ qua mà không chuyển BeatModel tuyệt vời đó sang Web chứ? Hãy nghĩ xem, bạn có thể điều khiển toàn bộ phiên DJ của mình thông qua một trang web trên điện thoại di động. Vì vậy, bây giờ bạn có thể ra khỏi gian hàng DJ đó và hòa mình vào đám đông. Bạn còn chờ gì nữa? Hãy viết mã đó!



Kế hoạch

① Sửa chữa mô hình.

Thực ra, chúng ta không cần phải sửa mô hình, nó vẫn ổn như vậy thôi!

② Tạo bộ điều khiển servlet

Chúng ta cần một servlet đơn giản có thể nhận các yêu cầu HTTP của chúng ta và thực hiện một vài thao tác trên mô hình. Tất cả những gì nó cần làm là dừng, bắt đầu và thay đổi nhịp mỗi phút.

③ Tạo chế độ xem HTML.

Chúng ta sẽ tạo một view đơn giản với JSP. Nó sẽ nhận một JavaBean từ controller để cho biết mọi thứ cần hiển thị. Sau đó, JSP sẽ tạo ra một giao diện HTML.



Những điều thú vị

Thiết lập môi trường Servlet của bạn

Việc chỉ cho bạn cách thiết lập môi trường servlet có vẻ hơi lạc đà đối với một cuốn sách về Mẫu thiết kế, ít nhất là nếu bạn không muốn cuốn sách đó nặng hơn bạn!

Mở trình duyệt web của bạn và truy cập thẳng vào <http://jakarta.apache.org/tomcat/> để tìm Tomcat Servlet Container của Dự án Apache Jakarta. Bạn sẽ tìm thấy mọi thứ bạn cần ở đó để bắt đầu và chạy.

Bạn cũng có thể muốn xem qua cuốn Head First Servlets & JSP của Bryan Basham, Kathy Sierra và Bert Bates.



mô hình 2 bộ điều khiển servlet

Bước một: mô hình

Hãy nhớ rằng trong MVC, mô hình không biết gì về các view hoặc controller. Nói cách khác, nó hoàn toàn tách biệt. Tất cả những gì nó biết là nó có thể có các observer cần thông báo. Đó là vẻ đẹp của Observer Pattern. Nó cũng cung cấp một giao diện mà các view và controller có thể sử dụng để lấy và thiết lập trạng thái của nó.

Bây giờ tất cả những gì chúng ta cần làm là điều chỉnh nó để hoạt động trong môi trường web, nhưng vì nó không phụ thuộc vào bất kỳ lớp bên ngoài nào nên thực sự không có việc gì phải làm. Chúng ta có thể sử dụng BeatModel của mình mà không cần thay đổi. Vì vậy, hãy làm việc hiệu quả và chuyển sang bước hai!

Bước hai: servlet điều khiển

Hãy nhớ rằng, servlet sẽ hoạt động như bộ điều khiển của chúng ta; nó sẽ nhận dữ liệu đầu vào từ trình duyệt web trong yêu cầu HTTP và chuyển đổi thành các hành động có thể áp dụng cho mô hình.

Sau đó, theo cách hoạt động của Web, chúng ta cần trả về một view cho trình duyệt. Để làm điều này, chúng ta sẽ chuyển quyền điều khiển cho view, có dạng JSP. Chúng ta sẽ thực hiện điều đó ở bước ba.

Sau đây là phác thảo về servlet; ở trang tiếp theo, chúng ta sẽ xem xét toàn bộ quá trình triển khai.

```

lớp công khai DJView mở rộng HttpServlet {
    public void init() ném ServletException {
        BeatModel beatModel = new BeatModel();
        beatModel.initialize();
        getServletContext().setAttribute("beatModel", beatModel);
    }
    // phương thức doPost ở đây
    public void doGet(HttpServletRequest yêu cầu,
                      Phản hồi HttpServletResponse)
        ném IOException, ServletException
    {
        // thực hiện ở đây
    }
}

```

Chúng tôi mở rộng lớp HttpServlet để có thể thực hiện những việc tương tự như servlet, như nhận các yêu cầu HTTP.

Đây là phương thức init; phương thức này được gọi khi servlet được tạo lần đầu tiên.

Đầu tiên chúng ta tạo một đối tượng BeatModel...

...và đặt tham chiếu đến nó trong ngữ cảnh của servlet để có thể truy cập dễ dàng.

Đây là phương thức doGet(). Đây là nơi công việc thực sự diễn ra. Chúng tôi có phần triển khai của nó ở trang tiếp theo.

Sau đây là cách triển khai phương thức doGet() từ trang trước:

```

public void doGet(HttpServletRequest yêu cầu,
                    HttpServletResponse phản hồi)
    ném IOException, ServletException
{
    Mô hình nhịp đậm Mô hình nhịp đậm =
        (BeatModel) getServletContext().getAttribute("beatModel");

    Chuỗi bpm = request.getParameter("bpm");
    nếu (bpm == null) {
        bpm = beatModel.getBPM() + "";
    }

    Chuỗi set = request.getParameter("set");
    nếu (đặt != null) {
        int bpmNumber = 90;
        bpmNumber = Integer.parseInt(bpm);
        beatModel.setBPM(bpmNumber);
    }

    Giảm chuỗi = request.getParameter("giảm");
    nếu (giảm != null) {
        beatModel.setBPM(beatModel.getBPM() - 1);
    }

    Tăng chuỗi = request.getParameter("tăng");
    nếu (tăng != null) {
        beatModel.setBPM(beatModel.getBPM() + 1);
    }

    Chuỗi trên = request.getParameter("on");
    nếu (trên != null) {
        beatModel.bắt đầu();
    }

    Chuỗi tắt = request.getParameter("tắt");
    nếu (tắt != null) {
        beatModel.dừng();
    }

    yêu cầu.setAttribute("beatModel", beatModel);

    RequestDispatcher người điều phối =
        yêu cầu.getRequestDispatcher("/jsp/DJView.jsp");
    dispatcher.forward(yêu cầu, phản hồi);
}

```

Đầu tiên, chúng ta lấy mô hình từ ngữ cảnh servlet. Chúng ta không thể thao tác mô hình mà không có tham chiếu đến nó.

Tiếp theo chúng ta lấy tất cả các lệnh/tham số HTTP...

Nếu chúng ta nhận được lệnh set, thì chúng ta sẽ lấy giá trị của set đó và thông báo cho mô hình.

Để tăng hoặc giảm, chúng ta lấy BPM hiện tại từ mô hình và điều chỉnh tăng hoặc giảm một đơn vị.

Nếu chúng ta nhận được lệnh bật/tắt, chúng ta sẽ yêu cầu mô hình bắt đầu hoặc dừng lại.

Cuối cùng, công việc của chúng ta với tư cách là một bộ điều khiển đã hoàn thành. Tắt cả những gì chúng ta cần làm là yêu cầu view tiếp quản và tạo một view HTML.

Theo định nghĩa Model 2, chúng ta truyền cho JSP một bean có trạng thái model trong đó. Trong trường hợp này, chúng ta truyền cho nó model thực tế, vì nó là một bean.

mô hình 2 xem

Bây giờ chúng ta cần một góc nhìn...

Tất cả những gì chúng ta cần là một chế độ xem và chúng ta đã có trình tạo nhịp điệu dựa trên trình duyệt sẵn sàng hoạt động!

Trong Model 2, view chỉ là JSP. Tất cả những gì JSP biết là bean mà nó nhận được từ controller. Trong trường hợp của chúng ta, bean đó chỉ là model và JSP sẽ chỉ sử dụng thuộc tính BPM của nó để trích xuất nhịp đậm hiện tại mỗi phút. Với dữ liệu đó trong tay, nó tạo ra view và cả các điều khiển giao diện người dùng.

```
<jsp:useBean id="beatModel" scope="request" class="headfirst.combined.djview.BeatModel" />
```

```
<html>
    <đầu>
        <title>Xem DJ</title>
    </đầu>
    <thân>
        <h1>Xem DJ</h1>
        Nhịp mỗi phút = <jsp:getProperty name="beatModel" property="BPM" />
        <br />
        <giờ>
        <br />
```

```
<form method="post" action="/djview/servlet/DJView">
    BPM: <input type="text" name="bpm"
        value=<jsp:getProperty name="beatModel" property="BPM" />">
    &nbsp;

    <input type="button" name="set" value="set"><br />
    <input type="button" name="giảm" value="<<" />
    <input type="button" name="tăng" value=">>" /><br />
    <input type="button" name="on" value="on" />
    <input type="button" name="tắt" value="tắt"><br />
    </biểu mẫu>

</thân>
</html>
```

Và đây là phần cuối
của HTML.

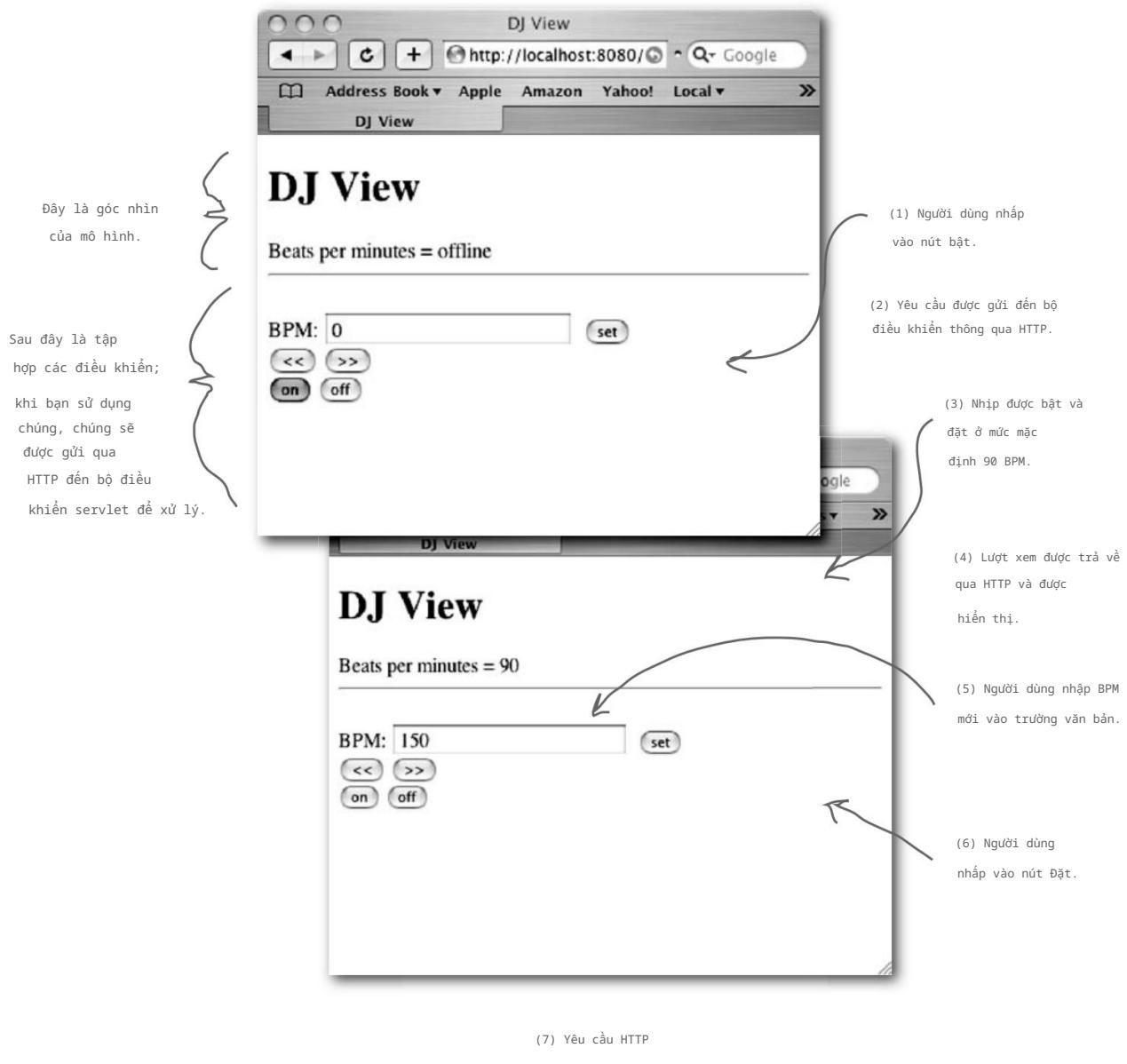
LƯU Ý rằng giống như MVC, trong Mô hình 2, chế độ xem
không thay đổi mô hình (đó là công việc của bộ điều khiển);

tất cả những gì nó làm là sử dụng trạng thái của mô hình!

mẫu hợp chất

Đưa Mô hình 2 vào thử nghiệm...

Đã đến lúc khởi động trình duyệt web, nhấn DJView Servlet và chạy thử hệ thống...

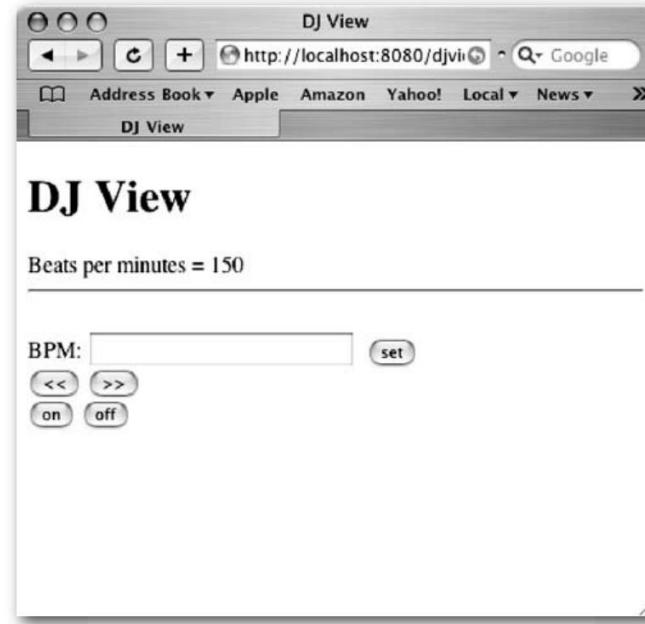


bạn đang ở đây 4 555

những việc cần làm với mô hình 2

(8) Bộ điều khiển
thay đổi mô hình thành
150 nhịp/phút

(9) Xem trả về HTML
phản ánh mô hình hiện
tại.



Những việc cần làm

- 1 Đầu tiên, hãy truy cập vào trang web; bạn sẽ thấy nhịp mỗi phút ở mức 0. Tiếp tục và nhấp vào nút "bật".
- 2 Nay giờ bạn sẽ thấy nhịp mỗi phút ở cài đặt mặc định: 90 BPM. Bạn cũng sẽ nghe thấy nhịp trên máy mà máy chủ đang chạy.
- 3 Nhập một nhịp cụ thể, ví dụ, 120, và nhấp vào nút "set". Trang sẽ làm mới với nhịp mỗi phút là 120 (và bạn sẽ nghe thấy nhịp tăng lên).
- 4 Nay giờ hãy sử dụng các nút tăng/giảm để điều chỉnh nhịp lên hoặc xuống.
- 5 Hãy nghĩ về cách thức hoạt động của từng bước trong hệ thống. Giao diện HTML đưa ra yêu cầu đến servlet (bộ điều khiển); servlet phân tích cú pháp đầu vào của người dùng và sau đó đưa ra yêu cầu đến mô hình. Sau đó, servlet chuyển quyền điều khiển đến JSP (chế độ xem), tạo ra chế độ xem HTML được trả về và hiển thị.

mẫu hợp chất

Mẫu thiết kế và Mô hình 2

Sau khi triển khai DJ Control cho Web bằng Model 2, bạn có thể tự hỏi các mẫu đã đi đâu. Chúng ta có một chế độ xem được tạo bằng HTML từ JSP nhưng chế độ xem không còn là trình lắng nghe của mô hình nữa. Chúng ta có một bộ điều khiển là một servlet nhận các yêu cầu HTTP, nhưng chúng ta vẫn đang sử dụng Strategy Pattern chứ? Còn Composite thì sao? Chúng ta có một chế độ xem được tạo từ HTML và hiển thị trong trình duyệt web. Đó vẫn là Composite Pattern chứ?

Mô hình 2 là sự thích ứng của MVC với Web

Mặc dù Model 2 không giống hệt MVC "sách giáo khoa", tắt cả các phần vẫn ở đó; chúng chỉ được điều chỉnh để phản ánh những đặc điểm riêng của mô hình trình duyệt web. Hãy cùng xem xét lại...

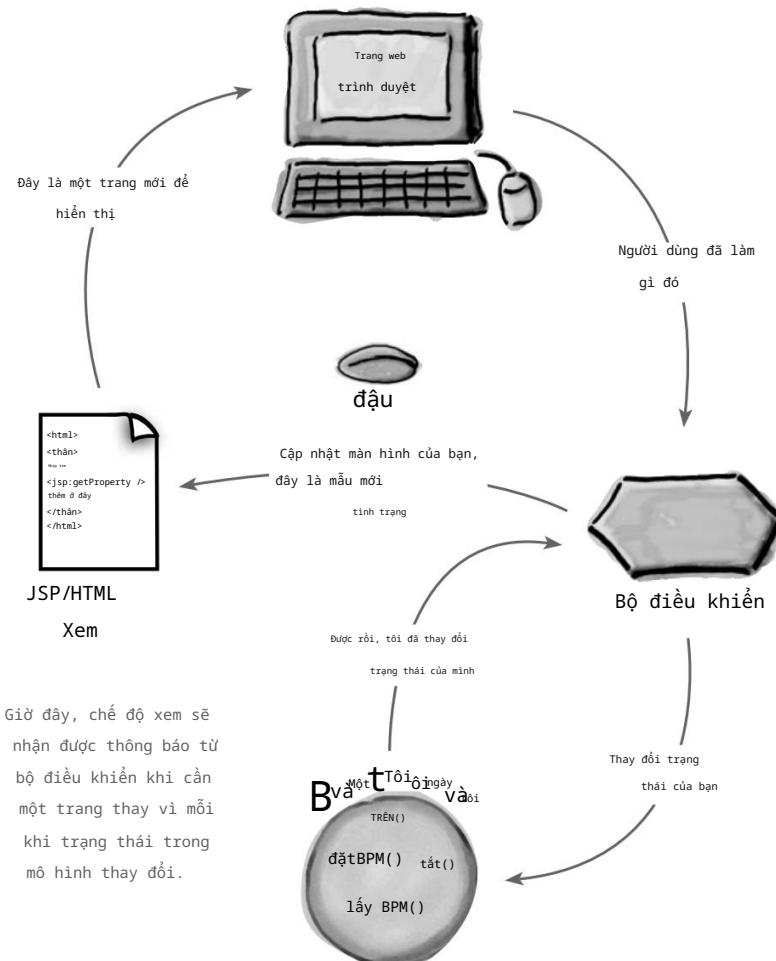
Người quan sát

Theo nghĩa cổ điển, chế độ xem không còn là người quan sát mô hình nữa; nghĩa là, nó không đăng ký với mô hình để nhận thông báo thay đổi trạng thái.

Tuy nhiên, view nhận được thông báo tương đương gián tiếp từ controller khi model đã thay đổi. Controller thậm chí còn truyền cho view một bean cho phép view lấy trạng thái của model.

Nếu bạn nghĩ về mô hình trình duyệt, chế độ xem chỉ cần cập nhật thông tin trạng thái khi phản hồi HTTP được trả về trình duyệt; thông báo vào bất kỳ thời điểm nào

khác sẽ vô nghĩa. Chỉ khi một trang được tạo và trả về thì việc tạo chế độ xem và kết hợp trạng thái của mô hình mới có ý nghĩa.



mẫu 2

Chiến lược

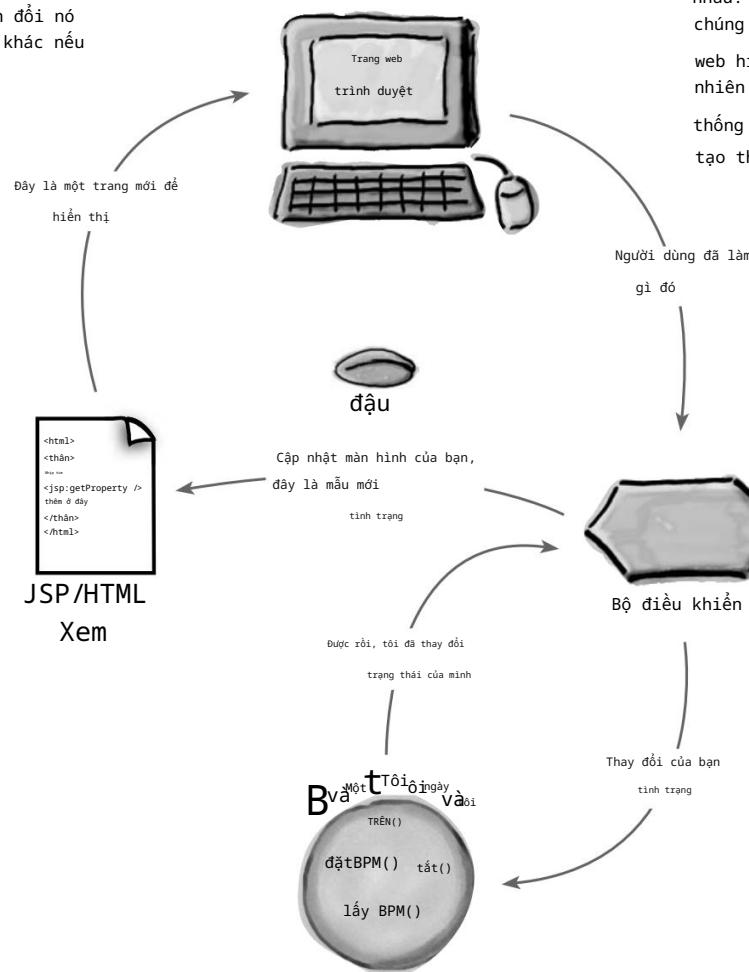
Trong Mô hình 2, đối tượng

Strategy vẫn là servlet điều

khiển; tuy nhiên, nó không

được tạo trực tiếp với chế độ
xem theo cách cổ điển.

Có thể nói, đây là một đối tượng
thực hiện hành vi cho chế độ xem
và chúng ta có thể hoán đổi nó
bằng một bộ điều khiển khác nếu
muốn có hành vi khác.



Hợp chất

Giống như GUI Swing của chúng tôi, chế độ xem cuối cùng
được tạo thành từ một tập hợp
các thành phần đồ họa lồng
nhau. Trong trường hợp này,
chúng được trình duyệt

web hiển thị từ mô tả HTML, tuy
nhiên bên dưới có một hệ
thống đối tượng rất có thể
tạo thành một hợp chất.

Bộ điều khiển vẫn cung
 cấp hành vi xem,
 ngay cả khi nó không
 được cấu thành với chế
 độ xem bằng cách sử dụng
 cấu thành đối tượng.

không có Những câu hỏi ngắn

Q: Có vẻ như bạn thực sự khéo tay

vỗ tay chào thực tế là Composite Pattern thực sự nằm trong MVC. Nó có thực sự ở đó không?

A: Vâng, Virginia, thực sự có một

Composite Pattern trong MVC. Nhưng thực ra, đây là một câu hỏi rất hay. Ngày nay, các gói GUI, như Swing, đã trở nên phức tạp đến mức chúng ta hầu như không nhận thấy cấu trúc bên trong và việc sử dụng composite trong việc xây dựng và cập nhật hiển thị.

Thậm chí còn khó nhận ra hơn khi chúng ta có trình duyệt web có thể sử dụng ngôn ngữ đánh dấu và chuyển đổi nó thành giao diện người dùng.

Trở lại thời điểm MVC mới được phát hiện, việc tạo GUI đòi hỏi nhiều sự can thiệp thủ công hơn và mô hình này rõ ràng là một phần của MVC.

Q: Bộ điều khiển có bao giờ

triển khai bất kỳ logic ứng dụng nào?

A: Không, bộ điều khiển thực hiện

hành vi cho quan điểm. Đó là sự thông minh để các hành động từ chế độ xem để hành động trên mô hình. Mô hình thực hiện các hành động đó và triển khai logic ứng dụng để quyết định phải làm gì để phản hồi lại các hành động đó. Bộ điều khiển có thể phải làm một chút công việc để xác định các lệnh gọi phương thức nào sẽ thực hiện trên mô hình, nhưng điều đó không được coi là "logic ứng dụng". Logic ứng dụng là mã quản lý và thao tác dữ liệu của bạn và nó nằm trong mô hình của bạn.

H: Tôi luôn tìm thấy từ

"Mô hình" thật khó để hiểu.

Bây giờ tôi hiểu rằng đó là phần cốt lõi của ứng dụng, nhưng tại sao lại sử dụng một từ mơ hồ, khó hiểu như vậy?

mô tả khía cạnh này của MVC?

A: Khi MVC được đặt tên, họ cần

một từ bắt đầu bằng chữ "M" hoặc bằng chữ khác thì họ không thể gọi nó là MVC.

Nhưng nghiêm túc mà nói, chúng tôi đồng ý với bạn, mọi người đều gán đầu và tự hỏi mô hình là gì.

Nhưng rồi mọi người nhận ra rằng họ cũng không thể nghĩ ra từ nào hay hơn.

H: Bạn đã nói rất nhiều về nhà nước của mô hình. Điều này có nghĩa là nó có Mẫu trạng thái trong đó không?

A: Không, chúng tôi muốn nói đến ý tưởng chung của trạng thái. Nhưng chắc chắn một số mô hình sử dụng Mẫu trạng thái để quản lý trạng thái nội bộ của chúng.

H: Tôi đã thấy mô tả về MVC

nói bộ điều khiển được mô tả như một "người trung gian" giữa quan điểm và mô hình. Bộ điều khiển có triển khai Mô hình trung gian không?

A: Chúng tôi chưa dề cập đến Người hòa giải

Pattern (mặc dù bạn sẽ tìm thấy bản tóm tắt về pattern trong phần phụ lục), vì vậy chúng ta sẽ không đi sâu vào quá nhiều chi tiết ở đây, nhưng mục đích của bộ trung gian là đóng gói cách các đối tượng tương tác và thúc đẩy sự kết hợp lồng lèo bằng cách ngăn hai đối tượng tham chiếu đến nhau một cách rõ ràng. Vì vậy, ở một mức độ nào đó, bộ điều khiển có thể được coi là một bộ trung gian, vì chế độ xem không bao giờ đặt trạng thái trực tiếp trên mô hình, mà luôn đi qua bộ điều khiển. Tuy nhiên, hãy nhớ rằng chế độ xem có tham chiếu đến mô hình để truy cập trạng thái của nó. Nếu bộ điều khiển thực sự là một bộ trung gian, chế độ xem sẽ phải đi qua bộ điều khiển để có được trạng thái của mô hình.

H: Liệu quan điểm luôn phải hỏi?

mô hình cho trạng thái của nó? Chúng ta không thể sử dụng mô hình đầy và gửi trạng thái của mô hình cùng với thông báo cập nhật sao?

A: Vâng, mô hình chắc chắn có thể gửi

trạng thái của nó với thông báo, và trên thực tế, nếu bạn nhìn lại chế độ xem JSP/HTML, thì đó chính xác là những gì chúng ta đang làm. Chúng ta đang gửi toàn bộ mô hình trong một bean, chế độ xem sử dụng bean này để truy cập trạng thái cần thiết bằng cách sử dụng các thuộc tính bean. Chúng ta có thể làm điều gì đó tương tự với BeatModel bằng cách chỉ gửi trạng thái mà chế độ xem quan tâm. Tuy nhiên, nếu bạn nhớ chương Observer Pattern, bạn cũng sẽ nhớ rằng có một vài nhược điểm đối với điều này. Nếu bạn không quay lại và xem lại lần thứ hai,

H: Nếu tôi có nhiều hơn một chế độ xem, tôi có phải luôn cần nhiều hơn một bộ điều khiển?

A: Thông thường, bạn cần một bộ điều khiển cho mỗi chế độ xem khi chạy; tuy nhiên, cùng một lớp bộ điều khiển có thể dễ dàng quản lý nhiều chế độ xem.

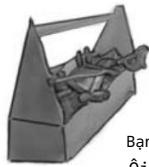
Q: Quan điểm không được cho là

thao tác mô hình, tuy nhiên tôi nhận thấy trong quá trình triển khai của bạn rằng chế độ xem có toàn quyền truy cập vào các phương thức thay đổi trạng thái của mô hình. Điều này có nguy hiểm không?

A: Bạn nói đúng; chúng tôi đã đưa ra quan điểm

quyền truy cập đầy đủ vào tập hợp các phương pháp của mô hình. Chúng tôi đã làm điều này để giữ mọi thứ đơn giản, nhưng có thể có những trường hợp bạn muốn cấp quyền truy cập chế độ xem chỉ cho một phần API của mô hình. Có một mẫu thiết kế tuyệt vời cho phép bạn điều chỉnh giao diện để chỉ cung cấp một tập hợp con. Bạn có thể nghĩ ra không?

hộp công cụ thiết kế



Công cụ cho hộp công cụ thiết kế của bạn

Bạn có thể gây ấn tượng với bất kỳ ai bằng bộ công cụ thiết kế của mình.
Ôi, hãy nhìn tất cả những nguyên tắc, mô hình đó và bây giờ là
mô hình hợp chất!

Nguyên tắc 00

Bao gồm những gì thay đổi.

Ưu tiên thành phần hơn là thừa kế.

Chương trình hướng tới giao diện, không phải triển khai.

Có gắng thiết kế các đối tượng tương tác một cách lồng lèo.

Các lớp học nên mở để mở rộng nhưng đóng để sửa đổi.

Phụ thuộc vào sự trùu tượng. Không phụ thuộc vào các lớp cũ thê.

Chỉ nói chuyện với bạn bè.

Đừng gọi cho chúng tôi, chúng tôi sẽ gọi cho bạn.

Một lớp học chỉ nên có một lý do để thay đổi.

Cặp bồ đề 00

Abstrac^t
Hình tượng
Đóng gói
Đối Mình
Điều Sắp

Mẫu hợp chất

Chúng ta có một cái mới
thể loại! MVC và
Model 2 là các mô
mẫu hợp chất.

Chúng tôi có một cái mới
thể loại! MVC và
Model 2 là các mô
mẫu hợp chất.

Mẫu hợp chất

một Mẫu hợp chất kết hợp hai hoặc
nhiều mẫu thành một giải pháp giải
quyết một vấn đề chung hoặc thường gặp.

Mẫu 00

Sau đây là một số ví dụ về cách sử dụng:

- Tiếng Anh: Cho phép một ứng dụng thay đổi trạng thái của nó mà không cần thay đổi ứng dụng.
- Điều khiển từ xa: Cho phép một ứng dụng thay đổi trạng thái của nó mà không cần thay đổi ứng dụng.
- Khái niệm: Cho phép một ứng dụng thay đổi trạng thái của nó mà không cần thay đổi ứng dụng.
- Yêu cầu hàng đợi hoặc yêu cầu bài nhảy kỹ, và hỗ trợ điều khiển hoàn việc khởi tạo thêm.
- Mô hình cấp độ không gian: Cho phép một ứng dụng thay đổi trạng thái của nó mà không cần thay đổi ứng dụng.



ĐIỂM ĐẦU TIÊN

B Bộ điều khiển chế độ xem mô hình

Mô hình MVC là một mô hình hợp thành bao gồm các mô hình Observer, Strategy và Composite.

B Mô hình sử dụng

Mẫu quan sát viên để có thể cập nhật thông tin cho người quan sát nhưng vẫn tách biệt khỏi họ.

B Bộ điều khiển là chiến lược

cho chế độ xem. Chế độ xem có thể sử dụng các triển khai khác nhau của bộ điều khiển để có được hành vi khác nhau.

B Chế độ xem sử dụng Mẫu tổng hợp để triển khai giao diện người dùng, thường bao gồm các thành phần lồng nhau như bảng điều khiển, khung và nút.

Các mẫu này hoạt động cùng nhau để tách biệt ba thành phần trong mô hình MVC, giúp thiết kế rõ ràng và linh hoạt.

B Mẫu Bộ điều hợp có thể là

được sử dụng để điều chỉnh một mô hình mới cho chế độ xem và bộ điều khiển hiện có.

B Mô hình 2 là sự điều chỉnh của MVC cho các ứng dụng web.

B Trong Mô hình 2, bộ điều khiển là

được triển khai dưới dạng servlet và JSP & HTML triển khai chế độ xem.



Giải pháp bài tập



Chuốt bút chì của bạn

QuackCounter cũng là Quackable. Khi chúng ta thay đổi Quackable để mở rộng QuackObservable, chúng ta phải thay đổi mọi lớp triển khai Quackable, bao gồm cả QuackCounter:

QuackCounter là Quackable, vì vậy bây giờ nó cũng là QuackObservable.

```

lớp công khai QuackCounter thực hiện Quackable {
    vịt kêu quack quack;
    int tinh sốQuacks;

    công khai QuackCounter(Quackable duck) {
        this.duck = vịt;
    }

    công khai void quack() {
        vịt.quack();
        số lần Quacks++;
    }

    công khai tinh int getQuacks() {
        trả về numberofQuacks;
    }

    public void registerObserver(Người quan sát quan sát) {
        duck.registerObserver(nghười quan sát);
    }

    công khai void thông báo cho người quan sát () {
        duck.notifyObservers();
    }
}

```

Đây là con vịt mà QuackCounter đang trang trí. Con vịt này thực sự cần phải xử lý các phương pháp có thể quan sát được.

Toàn bộ mã này giống với phiên bản trước của QuackCounter.

Sau đây là hai phương thức QuackObservable.

Lưu ý rằng chúng ta chỉ cần chuyển giao cả hai lệnh gọi đến con vịt mà chúng ta đang trang trí.

dung dịch mài sắc



Chuốt bút chì của bạn

Nếu Quackologist của chúng ta muốn quan sát toàn bộ một đàn thì sao? Dù sao thì điều đó có nghĩa là gì? Hãy nghĩ về nó như thế này: nếu chúng ta quan sát một tổ hợp, thì chúng ta đang quan sát mọi thứ trong tổ hợp đó. Vì vậy, khi bạn đăng ký với một đàn, tổ hợp đàn sẽ đảm bảo rằng bạn được đăng ký với tất cả các con của nó, có thể bao gồm các đàn khác.

```

lớp công khai Flock thực hiện Quackable {
    ArrayList vịt = new ArrayList();

    public void add(Vịt kêu cót két) {
        ducks.add(vịt);
    }

    công khai void quack() {
        Trình lặp iterator = ducks.iterator();
        trong khi (iterator.hasNext()) {
            Vịt kêu = (Quackable)iterator.next();
            vịt.quạc();
        }
    }

    public void registerObserver(Người quan sát quan sát) {
        Trình lặp iterator = ducks.iterator();
        trong khi (iterator.hasNext()) {
            Vịt kêu = (Quackable)iterator.next();
            duck.registerObserver(người quan sát);
        }
    }

    công khai void thông báo cho người quan sát () { }
}

```

Flock là Quackable, vì vậy bây giờ nó cũng là QuackObservable.

Đây là những Quackables trong đàn.

Khi bạn đăng ký làm Người quan sát với Đàn, thực ra bạn được đăng ký với mọi thứ TRONG Đàn, tức là mọi Quackable, dù là một con vịt hay một Đàn khác.

Chúng tôi lặp lại tất cả Quackables trong Flock và chuyển giao cuộc gọi cho từng Quackable. Nếu Quackable là một Flock khác, nó sẽ làm như vậy.

Mỗi Quackable đều có thông báo riêng nên Flock không cần phải lo lắng về điều đó. Điều này xảy ra khi Flock chuyển giao quack() cho mỗi Quackable trong Flock.

mẫu hợp chất



Chuốt bút chì của bạn

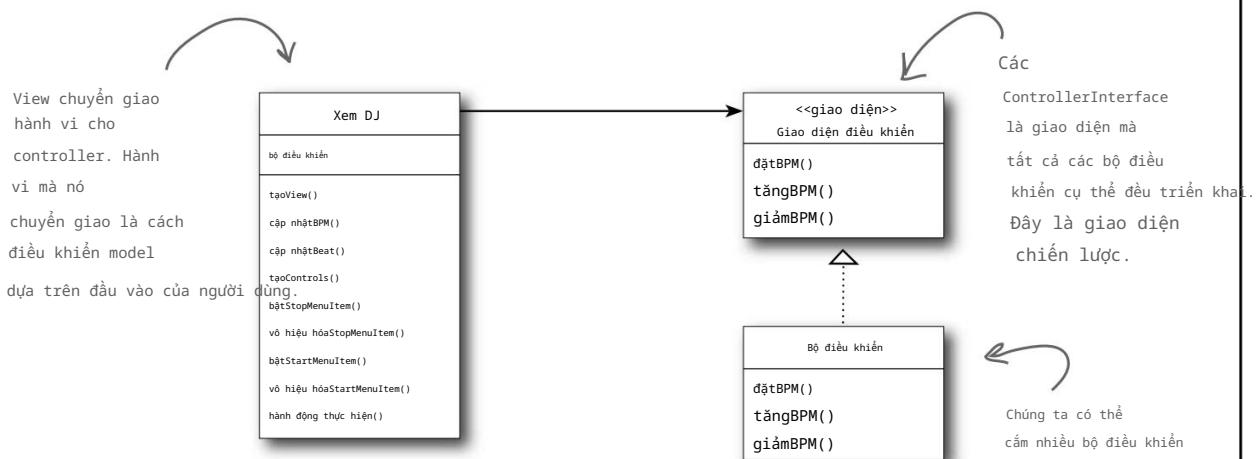
Chúng tôi vẫn đang trực tiếp khởi tạo Geese bằng cách dựa vào các lớp cụ thể. Bạn có thể viết Abstract Factory cho Geese không? Nó sẽ xử lý việc tạo ra "goose ducks" như thế nào?

Bạn có thể thêm phương thức `createGooseDuck()` vào Duck Factory hiện có. Hoặc, bạn có thể tạo một Factory hoàn toàn riêng biệt để tạo ra các họ Ngỗng.



Lớp thiết kế

Bạn đã thấy View và Controller cùng sử dụng Strategy Pattern. Bạn có thể vẽ sơ đồ lớp của hai thành phần này để thể hiện mẫu này không?



bạn đang ở đây 4 563

mã ready-bake : ứng dụng dj



Mã sẵn sàng nướng

Đây là bản triển khai đầy đủ của DJView. Nó hiển thị tất cả mã MIDI để tạo âm thanh và tắt cả các thành phần Swing để tạo chế độ xem. Bạn cũng có thể tải xuống mã này tại

<http://www.headfirstlabs.com>. Chúc bạn vui vẻ!

```
gói headfi rst.combined.djview;
```

```
lớp công khai DJTestDrive {
    public static void main (String[] args) {
        Mô hình giao diện BeatModel = new BeatModel();
        ControllerInterface controller = new BeatController(model);
    }
}
```

Mô hình nhịp độ

```
gói headfi rst.combined.djview;
```

```
giao diện công khai BeatModelInterface { void khởi
    tạo();

    void trên();

    void tắt();

    void setBPM(int bpm);

    int getBPM();

    void registerObserver(BeatObserver o);

    void removeObserver(BeatObserver o);

    void registerObserver(BPMObserver o);

    void removeObserver(BPMObserver o);
}
```

```
gói headfirst.combined.djview;

import javax.sound.midi.*; import java.util.*;
public class BeatModel implements
BeatModelInterface, MetaEventListener { Sequencer sequencer; ArrayList beatObservers = new ArrayList(); int bpm = 90; //
các biến thể hiện khác tại đây Sequence sequence; Track track;

công khai void khởi tạo() { setUpMidi();
    buildTrackAndStart();

}

public void on()
{ sequencer.start();
    setBPM(90);
}

public void off() { setBPM(0);
    sequencer.stop();

}

public void setBPM(int bpm) { this.bpm =
    bpm;
    sequencer.setTempoInBPM(getBPM());
    notifyBPMObservers();
}

công khai int getBPM() { trả về
    bpm;
}

void beatEvent() { thông
    báo choBeatObservers();
}

công khai void registerObserver(BeatObserver o) {
    beatObservers.add(o);
}

công khai void thông báoBeatObservers() {
    đối với (int i = 0; i < beatObservers.size(); i++) {
```

mã sẵn sàng để nướng : mô hình



Mã sẵn sàng nướng

```

    Người quan sát BeatObserver = (BeatObserver)beatObservers.get(i); người quan
    sát.updateBeat();
}

}

public void registerObserver(BPMObserver o) {
    bpmObservers.add(o);
}

công khai void notifyBPMObservers() {
    đối với (int i = 0; i < bpmObservers.size(); i++) { Người quan sát
        BPMObserver = (BPMObserver)bpmObservers.get(i); người quan sát.updateBPM();

    }
}

công khai void removeObserver(BeatObserver o) {
    int i = beatObservers.indexOf(o); nếu (i >= 0)

    { beatObservers.remove(i);
    }
}

công khai void removeObserver(BPMObserver o) {
    int i = bpmObservers.indexOf(o); nếu (i >= 0)
    { bpmObservers.remove(i);

    }
}

public void meta(MetaMessage message) { if (message.getType()
== 47) { beatEvent(); sequencer.start();
setBPM(getBPM());

    }
}

công khai void setUpMidi() { thử
    { trình
        sáp xếp = MidiSystem.getSequencer();
    }
}

```

```

sequencer.open();
sequencer.addMetaEventListener(cái này); sequence =
new Sequence(Sequence.PPQ,4); track = sequence.createTrack();
sequencer.setTempoInBPM(getBPM());

} catch(Ngoại lệ e) {
    e.printStackTrace();
}
}

công khai void buildTrackAndStart() {
    int[] trackList = {35, 0, 46, 0};

    chuỗi.deleteTrack(null); track =
    chuỗi.createTrack();

    makeTracks(trackList);
    track.add(makeEvent(192,9,1,0,4)); thử

    { sequencer.setSequence(sequence);
    } catch(Ngoại lệ e) {
        e.printStackTrace();
    }
}

công khai void makeTracks(int[] danh sách) {

    đối với (int i = 0; i < độ dài danh sách; i++) {
        int key = danh sách[i];

        nếu (key != 0)
            { track.add(makeEvent(144,9,key, 100, i));
            track.add(makeEvent(128,9,key, 100, i+1));
            }
    }
}

công khai MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
    Sự kiện MidiEvent = null; thử

    { ShortMessage a = new ShortMessage(); a.setMessage(comd,
        chan, one, two); sự kiện = new MidiEvent(a, tick);

    } catch(Ngoại lệ e) {
        e.printStackTrace();
    }
    trả về sự kiện;
}
}

```

máy sưởi sưởi để nướng : xem

Quan điểm

```
gói headfi rst.combined.djview;
```

```
giao diện công khai BeatObserver { void
    updateBeat();
}
```

```
gói headfi rst.combined.djview;
```

```
giao diện công khai BPMObserver { void
    updateBPM();
}
```

```
gói headfi rst.combined.djview;
```

```
nhập java.awt.*; nhập
```

```
java.awt.event.*; nhập javax.swing.*;
```

```
lớp công khai DJView triển khai
```

```
ActionListener, BeatObserver, BPMObserver {
```

```
Mô hình giao diện BeatModel;
```

```
Bộ điều khiểnGiao diện bộ điều khiển;
```

```
JFrame xem Khung;
```

```
JPanel xemBảng điều khiển;
```

```
Thanh BeatBar;
```

```
Nhãn JbmOutputLabel;
```

```
JFrame điều khiển Khung;
```

```
Bảng điều khiển JPanel;
```

```
Nhãn Jbm;
```

```
Trường văn bản JTextField;
```

```
JButton đặtBPMButton;
```

```
JButton tăng BPMButton;
```

```
JButton giảmBPMButton;
```

```
JMenuBar thanh menu;
```

```
Trình đơn JMenu;
```

```
Bắt đầu MenuItem;
```

```
đóng MenuItem;
```

```
public DJView(ControllerInterface controller, BeatModelInterface model) { this.controller = controller; this.model
    = model; model.registerObserver((BeatObserver)this);
```

```
    model.registerObserver((BPMObserver)this);
```

```
}
```

```
công khai void createView() {
```

Máy sưởi sưởi nướng



mẫu hợp chất

```

// Tạo tất cả các thành phần Swing tại đây viewPanel =
new JPanel(new GridLayout(1, 2)); viewFrame = new JFrame("View");
viewFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
viewFrame.setSize(new Dimension(100, 80)); bpmOutputLabel = new JLabel("offline",
SwingConstants.CENTER); beatBar = new BeatBar(); beatBar.setValue(0);
JPanel bpmPanel = new JPanel(new GridLayout(2, 1)); bpmPanel.add(beatBar); bpmPanel.add(bpmOutputLabel);
viewPanel.add(bpmPanel);

viewFrame.getContentPane().add(viewPanel, BorderLayout.CENTER); viewFrame.pack();
viewFrame.setVisible(true);

}

public void createControls() { // Tạo tất cả các
    thành phần Swing tại đây
    JFrame.setDefaultLookAndFeelDecorated(true); controlFrame = new
    JFrame("Control");
    controlFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); controlFrame.setSize(new Dimension(100,
    80));

    controlPanel = JPanel mới(GridLayout mới(1, 2));

    menuBar = new JMenuBar(); menu =
    new JMenu("DJ Control"); startMenuItem =
    new JMenuItem("Start"); menu.add(startMenuItem);
    startMenuItem.addActionListener(new
    ActionListener() { public void actionPerformed(ActionEvent event) {

        controller.start();
    }
});

stopMenuItem = new JMenuItem("Dừng"); menu.add(stopMenuItem);

stopMenuItem.addActionListener(ActionListener() mới { public void actionPerformed(sự
    kiện ActionEvent) { controller.stop(); //bpmOutputLabel.setText("ngoại tuyến");

    }
});

JMenuItem thoát = new JMenuItem("Thoát");
exit.addActionListener(ActionListener() mới {
    public void actionPerformed(sự kiện ActionEvent) { System.exit(0);

    }
});
}

```

bạn đang ở đây 4 569

mã săn sàng để nướng : xem



Mã săn sàng nướng

```
menu.add(thoát);
menuBar.add(menu);
controlFrame.setJMenuBar(menuBar);

bpmTextField = new JTextField(2); bpmLabel =
new JLabel("Nhập BPM:", SwingConstants.RIGHT); setBPMButton = new JButton("Set");
setBPMButton.setSize(new Dimension(10,40));
increaseBPMButton = new JButton(">>"); decreaseBPMButton =
new JButton("<<"); setBPMButton.addActionListener(này);
increaseBPMButton.addActionListener(này);
decreaseBPMButton.addActionListener(này);

Nút JPanelPanel = JPanel mới(GridLayout mới(1, 2));

buttonPanel.add(giảmBPMButton);
buttonPanel.add(tăngBPMButton);

JPanel enterPanel = new JPanel(new GridLayout(1, 2));
enterPanel.add(bpmLabel);
enterPanel.add(bpmTextField); JPanel
insideControlPanel = new JPanel(new GridLayout(3, 1)); insideControlPanel.add(enterPanel);
insideControlPanel.add(setBPMButton);
insideControlPanel.add(buttonPanel);
controlPanel.add(insideControlPanel);

bpmLabel.setBorder(BorderFactory.createEmptyBorder(5,5,5,5));
bpmOutputLabel.setBorder(BorderFactory.createEmptyBorder(5,5,5,5));

controlFrame.getRootPane().setDefaultButton(setBPMButton);
controlFrame.getContentPane().add(controlPanel, BorderLayout.CENTER);

controlFrame.pack();
controlFrame.setVisible(true);
}

public void enableStopMenuItem()
{ stopMenuItem.setEnabled(true);
}

công khai void disableStopMenuItem()
{ stopMenuItem.setEnabled(false);
}
```

```

}

public void enableStartMenuItem()
    { startMenuItem.setEnabled(dúng);
}

công khai void disableStartMenuItem()
    { startMenuItem.setEnabled(sai);
}

public void actionPerformed(sự kiện ActionEvent) { nếu (event.getSource()
== setBPMButton) { int bpm = Integer.parseInt(bpmTextField.getText());
    bộ điều khiển.setBPM(bpm);

} nếu không thì nếu (event.getSource() == increaseBPMButton) {
    bộ điều khiển. tăng BPM();
} nếu không thì nếu (event.getSource() == giảmBPMButton) {
    bộ điều khiển. giảm BPM();
}
}

công khai void updateBPM() {
    int bpm = model.getBPM(); if (bpm == 0)

    { bpmOutputLabel.setText("ngoại tuyến"); } else

    { bpmOutputLabel.setText("BPM hiện tại:
" + model.getBPM());
}
}

công khai void updateBeat()
    { beatBar.setValue(100);
}
}
}

```

Bộ điều khiển

```

gói headfirst.combined.djview;

Giao diện công khai ControllerInterface { void start();
void stop(); void
increaseBPM();
void decreaseBPM(); void
setBPM(int bpm);

}

```

mã sắn sàng để nướng : bộ điều khiển

Mã sắn sàng nướng



```

gói headfi rst.combined.djview;

lớp công khai BeatController thực hiện ControllerInterface {
    Mô hình giao diện BeatModel;
    Chế độ xem DJView;

    công khai BeatController(BeatModelInterface mô hình) {
        this.model = model; view
        = new DJView(this, model); view.createView();
        view.createControls();
        view.disableStopMenuItem();
        view.enableStartMenuItem();
        model.initialize();

    }

    công khai void start()
    { model.on();
      view.disableStartMenuItem();
      view.enableStopMenuItem();
    }

    công khai void stop() { model.
        tắt(); view. vô
        hiệu hóaStopMenuItem(); view.
        bậtStartMenuItem();
    }

    công khai void increaseBPM() {
        int bpm = model. getBPM(); model.
        setBPM(bpm + 1);
    }

    công khai void giảmBPM() {
        int bpm = model. getBPM(); model.
        setBPM(bpm - 1);
    }

    công khai void setBPM(int bpm) { model.
        setBPM(bpm);
    }
}

```

Mô hình trái tim

```
gói headfirst.combined.djview;

lớp công khai HeartTestDrive { công khai
    tĩnh void main (String[] args) {
        HeartModel heartModel = new HeartModel();
        Mô hình ControllerInterface = new HeartController(heartModel)
    }
}
```

```
gói headfirst.combined.djview; giao diện công khai  
HeartModelInterface {  
    int getHeartRate(); void  
    registerObserver(BeatObserver o); void  
    removeObserver(BeatObserver o); void  
    registerObserver(BPMObserver o); void  
    removeObserver(BPMObserver o);  
}
```

gói headfirst.combined.djview; nhập java.util.*

```
lớp công khai HeartModel triển khai HeartModelInterface, Runnable {  
    ArrayList beatObservers = new ArrayList(); ArrayList bpmObservers =  
    new ArrayList(); int time = 1000; int bpm = 90; Random random = new
```

```
Random(System.currentTimeMillis()); Luồng luồng;
```

```
public HeartModel() { thread =  
    new Thread(this); thread.start(); }
```

}

```
công khai void run() { int  
    lastrate = -1;
```

```
đổi với (;;)  
    { int thay đổi = random.nextInt(10);  
        (random.nextInt(2) == 0) {  
            thay đổi = 0 - thay đổi;
```

```
>     } int tyle = 60000/(thời gian + thay đổi); nếu (tyle < 120 && tyle > 50) { thời gian += thay đổi;
```

mã ready-bak : mô hình nhịp tim

```

        notifyBeatObservers(); nếu (tỷ lệ !=  

        tỷ lệ cuối cùng) { tỷ lệ cuối cùng =  

        tỷ lệ; notifyBPMObservers();  

    }  

  
    } thử  

    { Thread.sleep(thời gian); }  

    bắt (Ngoại lệ e) {}  

}  

  
} public int getHeartRate() { trả về 60000/  

    lần;  

}  

  
công khai void registerObserver(BeatObserver o) {  

    beatObservers.add(o);  

}  

  
công khai void removeObserver(BeatObserver o) {  

    int i = beatObservers.indexOf(o); nếu (i >= 0)  

  
    { beatObservers.remove(i);  

    }  

}  

  
công khai void thông báoBeatObservers() {  

    đối với (int i = 0; i < beatObservers.size(); i++) {  

        Người quan sát BeatObserver = (BeatObserver)beatObservers.get(i); người quan  

        sát.updateBeat();  

    }  

}  

  
public void registerObserver(BPMObserver o) {  

    bpmObservers.add(o);  

}  

  
công khai void removeObserver(BPMObserver o) {  

    int i = bpmObservers.indexOf(o); nếu (i >= 0)  

    { bpmObservers.remove(i);  

    }  

}  

  
công khai void notifyBPMObservers() {  

    đối với (int i = 0; i < bpmObservers.size(); i++) { Người quan sát  

        BPMObserver = (BPMObserver)bpmObservers.get(i); người quan sát.updateBPM();  

    }  

}
}
```

Mã sẵn sàng nướng



Bộ chuyển đổi trái tim

```
gói headfirst.combined.djview; lớp công khai
HeartAdapter triển khai BeatModelInterface {
    Giao diện HeartModel trái tim;

    công khai HeartAdapter(HeartModelInterface trái tim) {
        this.heart = trái tim;
    }

    công khai void khởi tạo() {}

    công khai void trên() {}

    công khai void off() {}

    công khai int getBPM() { trả về
        heart.getHeartRate();
    }

    công khai void setBPM(int bpm) {}

    công khai void registerObserver(BeatObserver o)
        { heart.registerObserver(o);
    }

    công khai void removeObserver(BeatObserver o) { heart.removeObserver(o);

    }

    public void registerObserver(BPMObserver o) { heart.registerObserver(o);

    }

    công khai void removeObserver(BPMObserver o)
        { heart.removeObserver(o);
    }
}
```

mã ready-bak : bộ điều khiển nhịp tim

Bộ điều khiển

Máy sưởi nướng



```

gói headfi rst.combined.djview;

lớp công khai HeartController thực hiện ControllerInterface {
    Mô hình giao diện HeartModel;
    Chế độ xem DJView;

    public HeartController(HeartModelInterface model) {
        this.model = mô hình;
        view = new DJView(cái này, new HeartAdapter(mô hình));
        view.createView();
        view.createControls();
        view.disableStopMenuItem();
        view.disableStartMenuItem();
    }

    công khai void start() {}

    công khai void dừng() {}

    công khai void increaseBPM() {}

    công khai void giảmBPM() {}

    công khai void setBPM(int bpm) {}
}

```

13 Cuộc sống tốt hơn với các mẫu

h Các mẫu trong hg g Thực tế Thế giới



Ahhh, giờ thì bạn đã sẵn sàng cho một thế giới mới tươi sáng tràn ngập các Mẫu thiết kế. Nhưng trước khi bạn mở tất cả những cánh cửa cơ hội mới đó, chúng ta cần đ𝐞 cập đến một vài chi tiết mà bạn sẽ gặp phải ngoài thế giới thực - đúng vậy, mọi thứ trở nên phức tạp hơn một chút so với ở Objectville. Hãy cùng theo dõi, chúng tôi có một hướng dẫn hay để giúp bạn vượt qua giai đoạn chuyển đổi ở trang tiếp theo...

những gì bạn sẽ học được từ hướng dẫn



Hướng dẫn của Objectville về

cuộc sống tốt đẹp hơn với các mẫu thiết kế

Vui lòng chấp nhận hướng dẫn hữu ích của chúng tôi với các mẹo và thủ thuật để sống chung
thế giới. Trong hướng dẫn này, bạn sẽ:

với các mẫu trong thế giới thực . Tìm hiểu những quan niệm sai lầm phổ biến về
"Mẫu thiết kế."

định nghĩa của một mẫu . Khám phá các Danh mục Mẫu thiết kế tiện dụng và lý do tại
lấy một cái.

sao bạn nhất định phải sử dụng . Tránh sự bối rối khi sử dụng Mẫu thiết kế không đúng lúc.

b Học cách giữ nguyên các mẫu trong phân loại theo đúng vị trí của chúng.

b Hãy đảm bảo rằng việc khám phá ra các mô hình không chỉ dành cho các bậc thầy; hãy đọc bài
Hướng dẫn và trở thành người viết mẫu.

viết nhanh của chúng tôi b Hãy có mặt ở đó khi danh tính thực sự của Băng nhóm bốn người bí ẩn được tiết lộ.

b Theo kịp hàng xóm - sách bàn cà phê bất kỳ mẫu người dùng
phải sở hữu.

b Học cách rèn luyện tâm trí như một thiền sư.

b Thu hút bạn bè và gây ảnh hưởng đến các nhà phát triển bằng cách cải thiện các mẫu của bạn
từ vựng.

Mẫu thiết kế được xác định

Chúng tôi cá là bạn đã có một ý tưởng khá hay về mẫu sau khi đọc cuốn sách này. Nhưng chúng tôi chưa bao giờ thực sự đưa ra định nghĩa cho Mẫu thiết kế. Vâng, bạn có thể hơi ngạc nhiên về định nghĩa được sử dụng phổ biến:

Một mẫu là giải pháp cho một vấn đề trong một bối cảnh.

Đó không phải là định nghĩa tiết lộ nhất phải không? Nhưng đừng lo lắng, chúng ta sẽ đi qua từng phần, bối cảnh, vấn đề và giải pháp sau:

Bối cảnh là tình huống mà mô hình áp dụng. Đây phải là tình huống lặp lại.

Vấn đề đề cập đến mục tiêu mà bạn đang cố gắng đạt được trong bối cảnh này, nhưng nó cũng đề cập đến bất kỳ hạn chế nào xảy ra trong bối cảnh đó.

Giải pháp chính là thứ bạn đang tìm kiếm: một thiết kế chung mà bất kỳ ai cũng có thể áp dụng để giải quyết mục tiêu và tập hợp các ràng buộc.

Ví dụ: Bạn có một bộ sưu tập các đối tượng.

Bạn cần phải bước qua các đối tượng mà không làm lộ phần triển khai của bộ sưu tập.

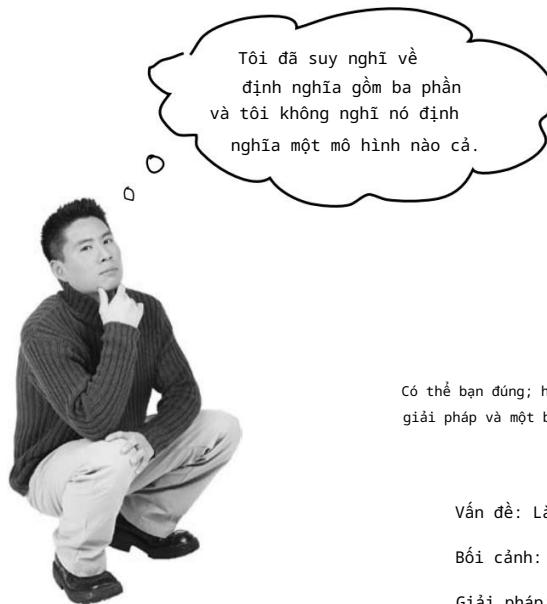
Đóng gói vòng lặp vào một lớp riêng biệt.

Đây là một trong những định nghĩa cần thời gian để thẩm nhuần, nhưng hãy thực hiện từng bước một. Sau đây là một mẹo ghi nhớ nhỏ mà bạn có thể lặp lại với chính mình để ghi nhớ:

"Nếu bạn thấy mình đang ở trong bối cảnh có một vấn đề có mục tiêu bị ảnh hưởng bởi một loạt các ràng buộc, thì bạn có thể áp dụng một thiết kế giải quyết được mục tiêu và các ràng buộc đó, đồng thời đưa ra giải pháp."

Bây giờ, có vẻ như rất nhiều công sức chỉ để tìm ra Design Pattern là gì. Rốt cuộc, bạn đã biết rằng Design Pattern cung cấp cho bạn giải pháp cho một vấn đề thiết kế thường gặp. Tất cả những thủ tục này mang lại cho bạn điều gì? Vâng, bạn sẽ thấy rằng bằng cách có một cách chính thức để mô tả các mẫu, chúng ta có thể tạo ra một danh mục các mẫu, có đủ mọi loại lợi ích.

mẫu thiết kế được xác định



Tôi đã suy nghĩ về
định nghĩa gồm ba phần
và tôi không nghĩ nó định
nghĩa một mô hình nào cả.

Có thể bạn đúng; hãy cùng suy nghĩ về điều này một chút... Chúng ta cần một vấn đề, một
giải pháp và một bối cảnh:

Vấn đề: Làm sao để tôi có thể đi làm đúng giờ?

Bối cảnh: Tôi đã quên chìa khóa trong xe.

Giải pháp: Đập vỡ cửa sổ, lên xe, nổ máy và lái xe
đi làm.

Chúng ta có tất cả các thành phần của định nghĩa: chúng ta có một vấn
đề, bao gồm mục tiêu đi làm, và các hạn chế về thời gian, khoảng cách và có
thể là một số yếu tố khác. Chúng ta cũng có một bối cảnh mà chìa khóa xe
không thể tiếp cận được. Và chúng ta có một giải pháp giúp chúng ta tiếp cận
được chìa khóa và giải quyết cả các hạn chế về thời gian và khoảng cách.

Bây giờ chúng ta phải có một khuôn mẫu chứ! Đúng không?

não Apower

Chúng tôi đã tuân theo định nghĩa của Design Pattern và định nghĩa một vấn đề, một bối cảnh và một giải pháp (có hiệu quả!). Đây
có phải là một pattern không? Nếu không, nó đã thất bại như thế nào? Chúng ta có thể thất bại theo cùng một cách khi định nghĩa một
Design Pattern OO không?

sống tốt hơn với các mẫu

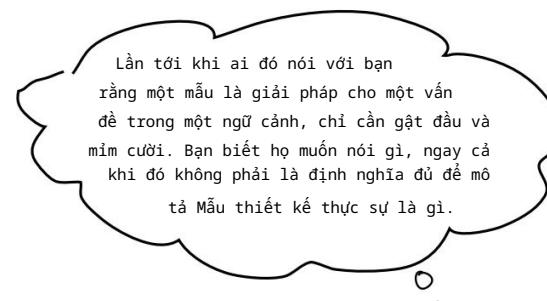
Nhìn kỹ hơn vào Định nghĩa mẫu thiết kế

Ví dụ của chúng tôi có vẻ phù hợp với định nghĩa Mẫu thiết kế, nhưng nó không phải là mẫu thực sự. Tại sao? Để bắt đầu, chúng ta biết rằng một mô hình cần áp dụng cho một vấn đề thường xuyên xảy ra. Trong khi một người đăng trí có thể thường xuyên khóa chìa khóa trong xe, thì việc đập vỡ cửa sổ xe không đủ điều kiện là một giải pháp có thể áp dụng nhiều lần (hoặc ít nhất là không có khả năng nếu chúng ta cân bằng mục tiêu với một ràng buộc khác: chi phí).

Nó cũng thất bại ở một số điểm khác: đầu tiên, không dễ để lấy mô tả này, đưa cho ai đó và yêu cầu họ áp dụng vào vấn đề riêng của họ.

Thứ hai, chúng ta đã vi phạm một khía cạnh quan trọng nhưng đơn giản của một mẫu: chúng ta thậm chí còn chưa đặt tên cho nó! Nếu không có tên, mẫu sẽ không trở thành một phần của vốn từ vựng có thể chia sẻ với các nhà phát triển khác.

May mắn thay, các mô hình không được mô tả và ghi chép lại như một vấn đề, bối cảnh và giải pháp đơn giản; chúng ta có nhiều cách tốt hơn để mô tả các mô hình và thu thập chúng lại thành danh mục mô hình.



Q: Tôi có thể thấy mẫu không?
những mô tả được nêu dưới dạng một vấn đề, một bối cảnh và một giải pháp?

A: Mô tả mẫu mà bạn sẽ thường thấy trong danh mục mẫu, thường tiết lộ nhiều hơn thế. Chúng ta sẽ xem xét chi tiết danh mục mẫu trong một phút nữa; chúng mô tả nhiều hơn về mục đích và động lực của mẫu và nơi nó có thể áp dụng, cùng với thiết kế giải pháp và hậu quả (tốt và xấu) của việc sử dụng nó.

Q: Có được phép thay đổi một chút không?
cấu trúc mẫu để phù hợp với thiết kế của tôi?
Hay tôi sẽ phải tuân theo định nghĩa nghiêm ngặt?

A: Tất nhiên là bạn có thể thay đổi nó. Giống như nguyên tắc thiết kế, các mẫu không phải là luật lệ hay quy tắc; chúng là những hướng dẫn mà bạn có thể thay đổi để phù hợp với nhu cầu của mình. Như bạn đã thấy, rất nhiều ví dụ thực tế không phù hợp với các thiết kế mẫu cổ điển.

Tuy nhiên, khi bạn điều chỉnh các mẫu, việc ghi lại sự khác biệt giữa mẫu của bạn và thiết kế cổ điển sẽ không bao giờ là thừa - theo cách đó, các nhà phát triển khác có thể nhanh chóng nhận ra các mẫu bạn đang sử dụng và bất kỳ sự khác biệt nào giữa mẫu của bạn và mẫu cổ điển.

Q: Tôi có thể lấy mẫu ở đâu?
danh mục?

A: Đầu tiên và chắc chắn nhất danh mục mẫu là Design Patterns: Elements of Reusable Object-Oriented Software, của Gamma, Helm, Johnson & Vlissides (Addison Wesley). Danh mục này trình bày 23 mẫu cơ bản. Chúng ta sẽ nói thêm một chút về cuốn sách này trong vài trang nữa. Nhiều danh mục mẫu khác cũng đang bắt đầu được xuất bản trong nhiều lĩnh vực khác nhau như phần mềm doanh nghiệp, hệ thống đồng thời và hệ thống kinh doanh.

lực lượng mục tiêu ràng buộc



Những điều thú vị

Mong rằng sức mạnh sẽ ở bên bạn

Định nghĩa về Mẫu thiết
kế cho chúng ta biết rằng
vấn đề bao gồm một mục tiêu
và một tập hợp các ràng buộc.
Các chuyên gia về mô hình có một
thuật ngữ cho những thứ này: họ
gọi chúng là lực. Tại sao? Vâng,
chúng tôi chắc chắn họ có lý do riêng,
nhưng nếu bạn nhớ bộ phim, lực này "định
hình và kiểm soát Vũ trụ".
Tương tự như vậy, các lực trong định nghĩa
mẫu hình sẽ định hình và kiểm soát giải pháp.
Chỉ khi nào giải pháp cân bằng được cả hai mặt
của lực (mặt sáng: mục tiêu của bạn và mặt tối:
những hạn chế) thì chúng ta mới có được một mô hình hữu ích.
Thuật ngữ "lực" này có thể khá khó hiểu khi bạn lần đầu
nhìn thấy nó trong các cuộc thảo luận về mô hình, nhưng
hãy nhớ rằng có hai mặt của lực (mục tiêu và ràng buộc) và
chúng cần được cân bằng hoặc giải quyết để tạo ra một giải
pháp mô hình. Đừng để thuật ngữ này cản trở bạn và mong lực sẽ
ở bên bạn!



Frank: Hãy cho chúng tôi biết, Jim. Tôi vừa mới học được các mẫu bằng cách đọc một vài bài viết ở đây và ở đó.

Jim: Chắc chắn rồi, mỗi danh mục mẫu sẽ lấy một tập hợp các mẫu và mô tả chi tiết từng mẫu cùng với mối quan hệ của nó với các mẫu khác.

Joe: Ý anh là có nhiều hơn một danh mục mẫu phải không?

Jim: Tất nhiên rồi; có các danh mục cho các Mẫu thiết kế cơ bản và cũng có các danh mục về các mẫu dành riêng cho từng miền, như mẫu EJB.

Frank: Bạn đang xem danh mục nào?

Jim: Đây là danh mục GoF cổ điển; nó bao gồm 23 Mẫu thiết kế cơ bản.

Frank: GoF à?

Jim: Đúng rồi, đó là viết tắt của Gang of Four. Gang of Four là những người đã lập ra danh mục mẫu đầu tiên.

Joe: Có gì trong danh mục?

Jim: Có một tập hợp các mẫu liên quan. Đối với mỗi mẫu có một mô tả theo một mẫu và nêu ra nhiều chi tiết của mẫu. Ví dụ, mỗi mẫu có một tên.

sử dụng danh mục mẫu

Frank: Wow, thật là chân động - một cái tên! Hãy tưởng tượng xem.

Jim: Đợi đã Frank; thực ra, cái tên thực sự quan trọng. Khi chúng ta có tên cho một mẫu, nó cho chúng ta cách để nói về mẫu đó; bạn biết đấy, toàn bộ thứ từ vựng chung đó.

Frank: Được rồi, được rồi. Tôi chỉ đùa thôi. Thôi nào, còn gì nữa?

Jim: Vâng, như tôi đã nói, mọi mẫu đều tuân theo một mẫu. Đối với mỗi mẫu, chúng tôi có một tên và một vài phần cho chúng tôi biết thêm về mẫu. Ví dụ, có một phần Ý định mô tả mẫu là gì, giống như một định nghĩa. Sau đó, có các phần Động lực và Khả năng áp dụng mô tả khi nào và ở đâu mẫu có thể được sử dụng.

Joe: Thế còn băn thiết kế thì sao?

Jim: Có một số phần mô tả thiết kế lớp cùng với tất cả các lớp tạo nên nó và vai trò của chúng.

Ngoài ra còn có một phần mô tả cách triển khai mẫu và thường là mã mẫu để chỉ cho bạn cách thực hiện.

Frank: Nghe có vẻ như họ đã nghĩ đến mọi thứ rồi.

Jim: Còn nữa. Ngoài ra còn có các ví dụ về cách sử dụng mẫu trong các hệ thống thực tế cũng như một trong những phần tôi cho là hữu ích nhất: mẫu liên quan đến các mẫu khác như thế nào.

Frank: Ô, ý anh là họ nói với anh những điều như nhà nước và chiến lược khác nhau thế nào phải không?

Jim: Chính xác!

Joe: Vậy Jim, thực ra anh sử dụng danh mục như thế nào? Khi gặp vấn đề, anh có tìm kiếm giải pháp trong danh mục không?

Jim: Tôi có gắng làm quen với tất cả các mẫu và mối quan hệ của chúng trước. Sau đó, khi tôi cần một mẫu, tôi có một số ý tưởng về mẫu đó. Tôi quay lại và xem các phần Động lực và Khả năng áp dụng để đảm bảo rằng tôi đã hiểu đúng. Ngoài ra còn có một phần thực sự quan trọng khác: Hậu quả. Tôi xem lại phần đó để đảm bảo rằng sẽ không có tác động không mong muốn nào đến thiết kế của tôi.

Frank: Điều đó có lý. Vậy khi bạn biết mẫu là đúng, bạn sẽ tiếp cận việc đưa nó vào thiết kế và triển khai nó như thế nào?

Jim: Đó là lúc sơ đồ lớp xuất hiện. Đầu tiên, tôi đọc qua phần Cấu trúc để xem lại sơ đồ và sau đó là phần Người tham gia để đảm bảo tôi hiểu vai trò của từng lớp. Từ đó, tôi đưa nó vào thiết kế của mình, thực hiện bất kỳ thay đổi nào tôi cần để làm cho nó phù hợp. Sau đó, tôi xem lại phần Triển khai và Mã mẫu để đảm bảo tôi biết về bất kỳ kỹ thuật triển khai tốt hoặc vẫn đề nào tôi có thể gặp phải.

Joe: Tôi thấy rằng một danh mục thực sự có thể giúp tôi tăng tốc việc sử dụng các mẫu mã!

Frank: Hoàn toàn đúng. Jim, anh có thể hướng dẫn chúng tôi mô tả mẫu được không?

sống tốt hơn với các mẫu

Tất cả các mẫu trong danh mục đều bắt đầu bằng một cái tên. Tên là một phần quan trọng của một mẫu - nếu không có một cái tên hay, mẫu không thể trở thành một phần của vốn từ vựng mà bạn chia sẻ với các nhà phát triển khác.

Động lực cung cấp cho bạn một kịch bản cụ thể mô tả vấn đề và cách giải quyết vấn đề.

Khả năng áp dụng mô tả các tình huống mà mẫu có thể được áp dụng.

Những người tham gia là các lớp và đối tượng trong thiết kế. Phần này mô tả trách nhiệm và vai trò của họ trong mẫu.

Hậu quả mô tả những tác động mà việc sử dụng mô hình này có thể mang lại: tốt và xấu.

Phản triển khai cung cấp các kỹ thuật bạn cần sử dụng khi triển khai mô hình này và những vấn đề bạn nên chú ý.

Những cách sử dụng đã biết mô tả các ví dụ về mô hình này được tìm thấy trong các hệ thống thực tế.

DÖN Tạo đối tượng

Ý định

Et aliquat, velesto ent llore feuis acillao rperci tet, quat nonsequam il ea at nim nos do enim qui ex faci tet, sepius diom utat, volere magnisi. Und modicore dicitur laevet augim iril el

admodum et modicore dicitur laevet augim iril el

Os nesciencia et laesio do con al cipitum correcitum sequit dolorem logit met vel

incidente dipes fengit nam elumen nra blace sequit vel etut magna aquit.

Aliquis non vel esset se minuscus do delictis ad aspirit, sic circlit ipsum dolorum dignissim

super sequit et am quate magis illam xixit ad magna feu, facint delit ut

Động lực

Et aliquat, velesto ent llore feuis acillao rperci tet, quat nonsequam il ea at nim nos do enim qui ex faci tet, sepius diom utat, volere magnisi. Und modicore dicitur laevet augim iril el

admodum et modicore dicitur laevet augim iril el

Os nesciencia et laesio do con al cipitum correcitum sequit dolorem logit met vel

incidente dipes fengit nam elumen nra blace sequit vel etut magna aquit.

Aliquis non vel esset se minuscus do delictis ad aspirit, sic circlit ipsum dolorum dignissim

super sequit et am quate magis illam xixit ad magna feu, facint delit ut

Khả năng áp dụng

Bon neden nulputre iuris exercec consilium wissifectem ad magna aliqui dñi, cumillante dolore magna feuis nos alit ad magna quarte modolare vent lut lupat prat. Dui blace min ea feupit ing

dolore magna feuis nos alit ad magna quarte modolare vent lut lupat prat. Dui blace min ea feupit

ing ent llore magnis enat wissifecte et, succilla ad minicini blam dolorse rcllit

irit, conse dolore dolore et, verci enis ent ip elesequeil ut ad esectem ing ea con eros autem

Kết cấu

Đối tượng
không rõ ràng
đã xác định
chưa xác định

Người tham gia

Bon neden nulputre iuris exercec consilium wissifectem ad magna aliqui dñi, cumillante dolore magna feuis nos alit ad magna quarte modolare vent lut lupat prat. Dui blace min ea feupit

ing ent llore magnis enat wissifecte et, succilla ad minicini blam dolorse rcllit

irit, conse dolore dolore et, verci enis ent ip elesequeil ut ad esectem ing ea con eros autem

Sự hợp tác

8. I dipes dñi huy daesitum magnis enat wissifecte et, succilla ad minicini blam dolorse rcllit

Hậu quả

Bon neden nulputre iuris exercec consilium wissifectem ad magna aliqui dñi, cumillante:

1. Dolore dolore et, verci enis ent ip elesequeil ut ad esectem ing ea con eros autem diam nornolu tptissi imodigibl er.

2. Modolare thong he lu lupat prat. Dui blace min ea feupit ing ent llore magnis

enat, verci enis ent ip elesequeil ut ad esectem ing ea con eros autem diam nornolu tptissi imodigibl er.

3. Dolore dolore et, verci enis ent ip elesequeil ut ad esectem ing ea con eros autem

diam nornolu tptissi imodigibl er.

4. Modolare thong he lu lupat prat. Dui blace min ea feupit ing ent llore magnis

enat wissifecte et, succilla ad minicini blam dolorse rcllit irit, conse dolore dolore

et, verci enis ent ip elesequeil ut ad esectem ing ea con eros autem

Mã mẫu/Thực hiện

Dolore dolore iuris exercec consilium wissifectem ad magna aliqui dñi, cumillante dolore magna feuis nos alit ad magna quarte modolare vent lut lupat prat. Dui blace min ea feupit

ing ent llore magnis enat wissifecte et, succilla ad minicini blam dolorse rcllit

irit, conse dolore dolore et, verci enis ent ip elesequeil ut ad esectem ing ea con eros autem

diam nornolu tptissi imodigibl er.

```
IdempotencyCheck ( )
Singleton tên riêng tư uniqueInstance;
// các biến thô như tên biến ở đây
Singleton riêng tư ()
```

```
công khai tên đồng bộ Singleton getInstance () { # # uniqueInstance
```

```
    => null;
    uniqueInstance = new Singleton();
    trả về uniqueInstance;
```

```
    // các phương pháp hữu ích khác ở đây
```

Công dụng đã biết

Dolore dolore iuris exercec consilium wissifectem ad magna aliqui dñi, cumillante dolore magna feuis nos alit ad magna quarte modolare vent lut lupat prat. Dui blace min ea feupit

ing ent llore magnis enat wissifecte et, succilla ad minicini blam dolorse rcllit

irit, conse dolore dolore et, verci enis ent ip elesequeil ut ad esectem ing ea con eros autem

diam nornolu tptissi imodigibl er.

Các mẫu liên quan

Cinequpit et ad esectem ing ea con eros autem diam nornolu tptissi imodigibl er; alit ad magna

magna quarte modolare vent lut lupat prat. Dui blace min ea feupit ing ent llore magnis

enat wissifecte et, succilla ad minicini blam dolorse rcllit irit, conse dolore dolore et, verci

enis ent ip elesequeil ut ad esectem ing ea con eros autem diam nornolu tptissi

bạn đang ở đây 4 585

Đây là phân loại hoặc thể loại của mẫu. Chúng ta sẽ nói về chúng trong một vài trang.

Mục đích mô tả những gì mẫu thực hiện trong một câu lệnh ngắn. Bạn cũng có thể coi đây là định nghĩa của mẫu (giống như chúng ta đã sử dụng trong cuốn sách này).

Câu trúc này cung cấp sơ đồ minh họa mối quan hệ giữa các lớp tham gia vào mô hình.

Sự hợp tác

Sự hợp tác cho chúng ta biết những người tham gia làm việc cùng nhau như thế nào trong mô hình.

Nã mẫu cung cấp các đoạn mã có thể giúp ích cho việc của bạn triển khai.

Các mẫu liên quan mô tả mối quan hệ giữa mẫu này và các mẫu khác.

khám phá các mẫu của riêng bạn

không có Những câu hỏi ngớ ngẩn

Q: Bạn có thể tự tạo Thiết kế của riêng mình không?
Các mẫu? Hay đó là điều bạn phải là "chuyên
gia về mẫu" mới làm được?

A: Đầu tiên, hãy nhớ rằng các mô hình được phát hiện,
không được tạo ra. Vì vậy, bất kỳ ai cũng có thể khám phá một Mẫu
thiết kế và sau đó viết mô tả của nó; tuy nhiên, điều này không dễ
dàng và không xảy ra nhanh chóng, cũng không thường xuyên. Trở thành
một "người viết mẫu" đòi hỏi sự cam kết.

Trước tiên, bạn nên nghĩ về lý do tại sao bạn muốn làm như
vậy - phần lớn mọi người không tạo ra các mẫu; họ chỉ sử dụng
chúng. Tuy nhiên, bạn có thể làm việc trong một lĩnh
vực chuyên môn mà bạn nghĩ rằng các mẫu mới sẽ
hữu ích, hoặc bạn có thể tìm ra giải pháp cho
vấn đề mà bạn cho là thường gặp, hoặc bạn chỉ muốn
tham gia vào cộng đồng mẫu và đóng góp vào khối
lượng công việc ngày càng tăng.

H: Tôi muốn chơi, tôi phải bắt đầu như thế nào?

A: Giống như bất kỳ ngành học nào, bạn càng biết nhiều thi
tốt hơn. Nghiên cứu các mẫu hiện có, chúng làm gì và
chúng liên quan như thế nào đến các mẫu khác là rất
quan trọng. Nó không chỉ giúp bạn quen thuộc với cách các
mẫu được tạo ra mà còn ngăn bạn phát minh lại bánh xe.
Từ đó, bạn sẽ muốn bắt đầu viết các mẫu của mình trên giấy
để có thể truyền đạt chúng cho các nhà phát triển khác;
chúng ta sẽ nói thêm về cách truyền đạt các mẫu của bạn
sau một chút. Nếu bạn thực sự quan tâm, bạn sẽ muốn đọc
phần sau các câu hỏi và trả lời này.

H: Làm sao tôi biết được mình thực sự có một khuôn mẫu?

A: Đó là một câu hỏi rất hay: bạn không có
cho đến khi những người khác đã sử dụng và thấy nó hiệu quả.
Nhìn chung, bạn sẽ không có một mẫu nào cho đến khi nó vượt
qua "Quy tắc ba". Quy tắc này nêu rằng một mẫu chỉ có thể
được gọi là mẫu nếu nó đã được áp dụng trong một giải pháp thực
tế ít nhất ba lần.

Vậy bạn có muốn trở thành ngôi sao trong
lĩnh vực thiết kế mẫu không?

Bây giờ hãy lắng nghe những gì tôi nói.

Hãy sở hữu cho mình một
danh mục mẫu,

Sau đó hãy dành thời gian và
học thật kỹ.

Và khi bạn đã mô tả đúng,

Và ba nhà phát triển đồng ý mà
không cần tranh cãi,

Khi đó bạn sẽ biết đó là
một mô hình đúng đắn.



Theo giai điệu của bài hát "So you
wanna be a Rock'n Roll Star."

sống tốt hơn với các mẫu

Vì vậy, bạn muốn trở thành một nhà văn Thiết kế mẫu

Hãy làm bài tập về nhà. Bạn cần phải thành thạo các mẫu hiện có trước khi có thể tạo ra một mẫu mới. Hầu hết các mẫu có vẻ mới, thực chất chỉ là các biến thể của các mẫu hiện có. Bằng cách nghiên cứu các mẫu, bạn sẽ nhận ra chúng tốt hơn và học cách liên hệ chúng với các mẫu khác.

Hãy dành thời gian để suy ngẫm, đánh giá. Kinh nghiệm của bạn - những vấn đề bạn đã gặp phải và các giải pháp bạn đã sử dụng - là nơi ý tưởng cho các mẫu được sinh ra. Vì vậy, hãy dành thời gian để suy ngẫm về những kinh nghiệm của bạn và sàng lọc chúng để tìm ra những thiết kế mới lạ có thể tái diễn. Hãy nhớ rằng hầu hết các thiết kế đều là biến thể của các mẫu hiện có chứ không phải mẫu mới. Và khi bạn tìm thấy thử trông giống như một mẫu mới, khả năng áp dụng của nó có thể quá hẹp để đủ điều kiện là một mẫu thực sự.

Hãy ghi ý tưởng của bạn ra giấy theo cách mà người khác có thể hiểu được. Việc tìm ra các mẫu mới không có nhiều tác dụng nếu người khác không thể sử dụng phát hiện của bạn; bạn cần ghi lại các ứng viên mẫu của mình để người khác có thể đọc, hiểu và áp dụng chúng vào giải pháp của riêng họ và sau đó cung cấp cho bạn phản hồi. May mắn thay, bạn không cần phải tự sáng tạo ra phương pháp ghi lại các mẫu của mình. Như bạn đã thấy với mẫu GoF, rất nhiều suy nghĩ đã được đưa ra về cách mô tả các mẫu và đặc điểm của chúng.

Hãy để những người khác thử các mẫu của bạn; sau đó tinh chỉnh và tinh chỉnh thêm nữa. Đừng mong đợi bạn sẽ làm đúng mẫu ngay từ lần đầu tiên. Hãy nghĩ về mẫu của bạn như một tác phẩm đang trong quá trình hoàn thiện và sẽ được cải thiện theo thời gian. Hãy để những nhà phát triển khác xem xét mẫu ứng viên của bạn, thử nghiệm và phản hồi cho bạn. Kết hợp phản hồi đó vào mô tả của bạn và thử lại. Mô tả của bạn sẽ không bao giờ hoàn hảo, nhưng đến một lúc nào đó, nó sẽ đủ vững chắc để những nhà phát triển khác có thể đọc và hiểu được.

Đừng quên quy tắc ba. Hãy nhớ rằng, trừ khi mô hình của bạn đã được áp dụng thành công trong ba giải pháp thực tế, nó không thể đủ điều kiện là một mô hình. Đó là một lý do chính đáng khác để đưa mô hình của bạn vào tay những người khác để họ có thể thử, đưa ra phản hồi và cho phép bạn hội tụ vào một mô hình hoạt động.

Sử dụng một trong các mẫu mẫu hiện có để xác định mẫu của bạn. Rất nhiều suy nghĩ đã được đưa vào các mẫu này và những người dùng mẫu khác sẽ nhận ra định dạng.



Ai làm gì?



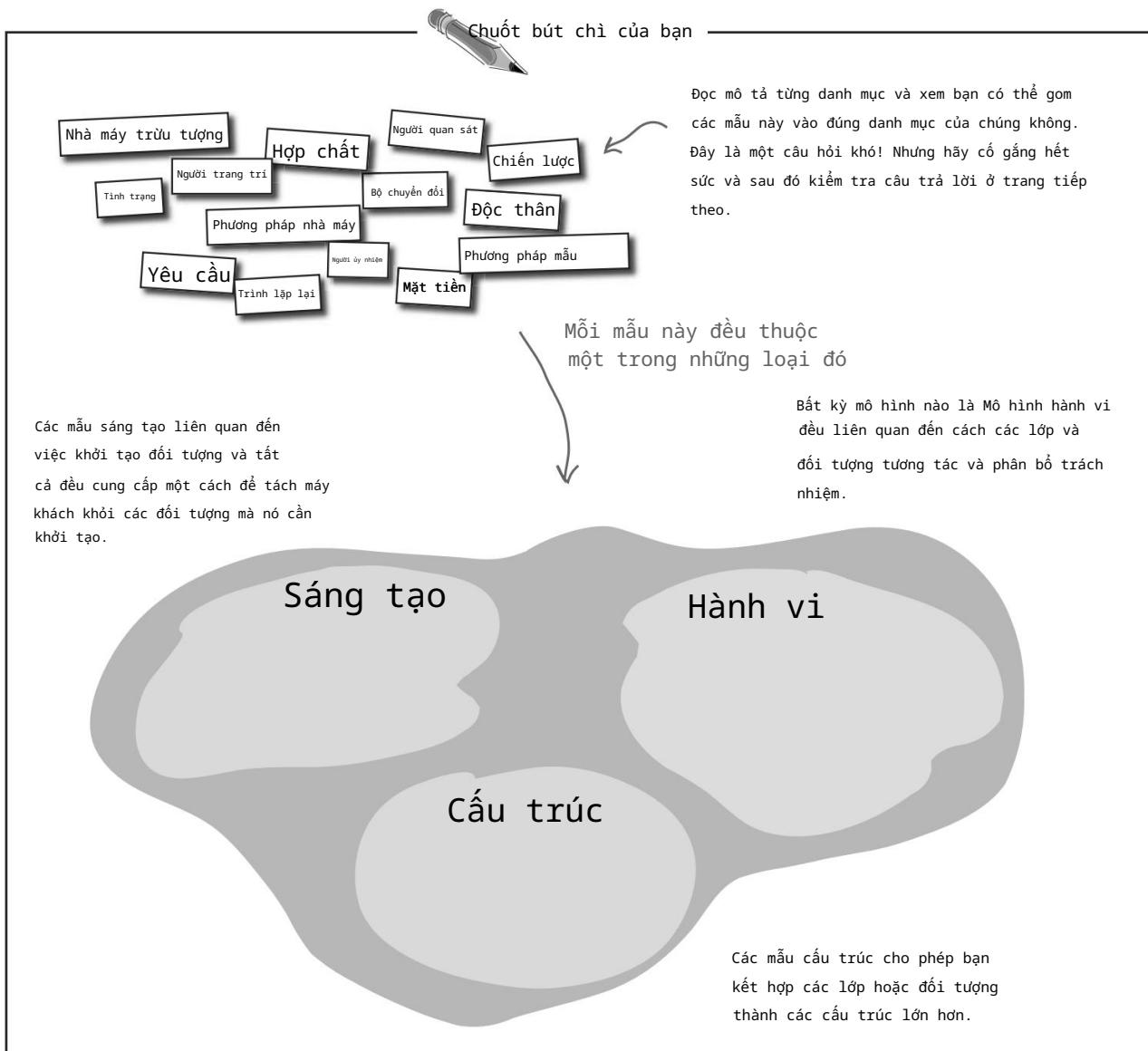
Ghép mỗi mẫu với mô tả của nó:

Mẫu	Sự miêu tả
Người trang trí	Bao bọc một đối tượng và cung cấp một giao diện khác cho đối tượng đó.
Tình trạng	Các lớp con quyết định cách triển khai các bước trong một thuật toán.
Trình lắp lại	Các lớp con quyết định lớp cụ thể nào sẽ được tạo nên.
Mặt tiền	Đảm bảo chỉ tạo ra một đối tượng duy nhất.
Chiến lược	Bao gồm các hành vi có thể hoán đổi cho nhau và sử dụng sự phân quyền để quyết định sử dụng hành vi nào.
Người ủy nhiệm	Khách hàng xử lý các tập hợp đối tượng và các đối tượng riêng lẻ một cách thống nhất.
Phương pháp nhà máy	Bao gồm các hành vi dựa trên trạng thái và sử dụng sự phân quyền để chuyển đổi giữa các hành vi.
Bộ chuyển đổi	Cung cấp một cách để duyệt qua một tập hợp các đối tượng mà không cần tiết lộ phần triển khai của nó.
Người quan sát	Đơn giản hóa giao diện của một tập hợp các lớp.
Phương pháp mẫu	Bao bọc một đối tượng để cung cấp hành vi mới.
Hợp chất	Cho phép khách hàng tạo ra các họ đối tượng mà không cần chỉ định các lớp cụ thể của chúng.
Độc thân	Cho phép các đối tượng được thông báo khi trạng thái thay đổi.
Nhà máy trừu tượng	Bao bọc một đối tượng để kiểm soát quyền truy cập vào đối tượng đó.
Yêu cầu	Đóng gói yêu cầu dưới dạng một đối tượng.

Tổ chức các mẫu thiết kế

Khi số lượng các Mẫu thiết kế được phát hiện tăng lên, việc phân chia chúng thành các phân loại là điều hợp lý để chúng ta có thể sắp xếp chúng, thu hẹp phạm vi tìm kiếm của mình xuống một tập hợp con của tất cả các Mẫu thiết kế và thực hiện so sánh trong một nhóm các mẫu.

Trong hầu hết các danh mục, bạn sẽ tìm thấy các mẫu được nhóm thành một trong số ít các lược đồ phân loại. Lược đồ nổi tiếng nhất được sử dụng bởi danh mục mẫu đầu tiên và phân chia các mẫu thành ba loại riêng biệt dựa trên mục đích của chúng: Sáng tạo, Hành vi và Cấu trúc.



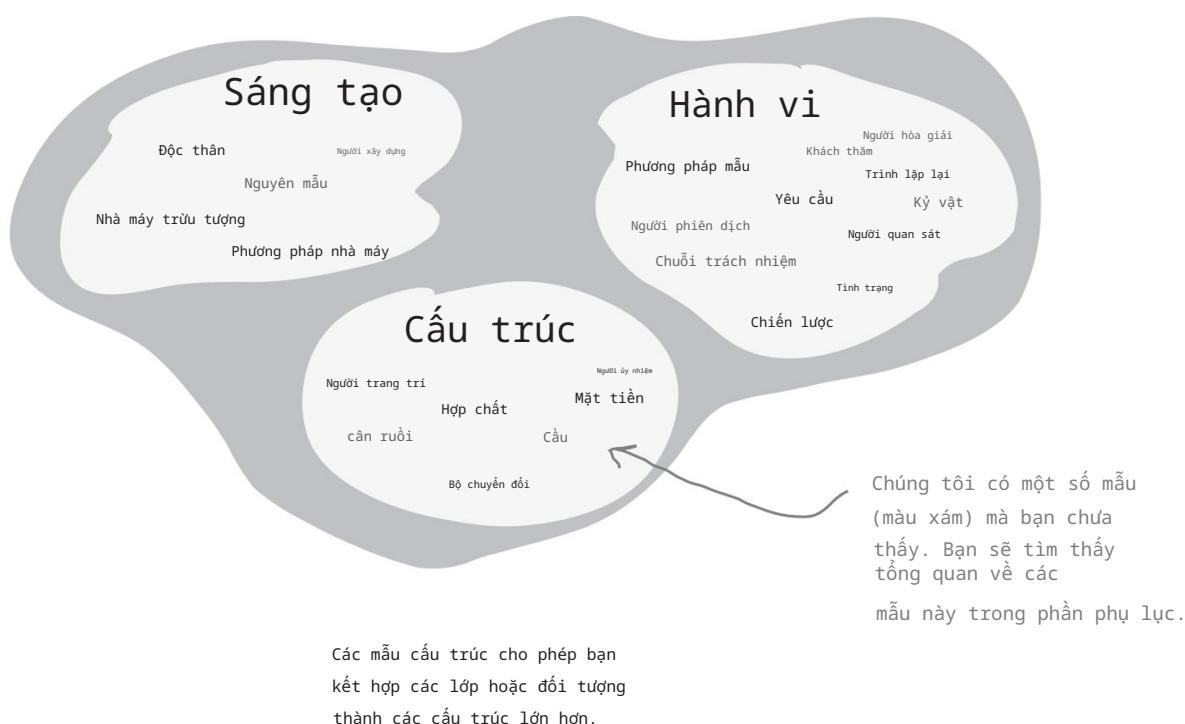
danh mục mẫu

Giải pháp: Các loại mẫu

Sau đây là nhóm các mẫu thành các danh mục. Có thể bạn thấy bài tập này khó, vì nhiều mẫu có vẻ như có thể phù hợp với nhiều hơn một danh mục. Đừng lo, mọi người đều gặp khó khăn trong việc tìm ra danh mục phù hợp cho các mẫu.

Các mẫu sáng tạo liên quan đến việc khởi tạo đối tượng và tắt cả đều cung cấp một cách để tách máy khách khỏi các đối tượng mà nó cần khởi tạo.

Bất kỳ mô hình nào là Mô hình hành vi đều liên quan đến cách các lớp và đối tượng tương tác và phân bổ trách nhiệm.



sống tốt hơn với các mẫu

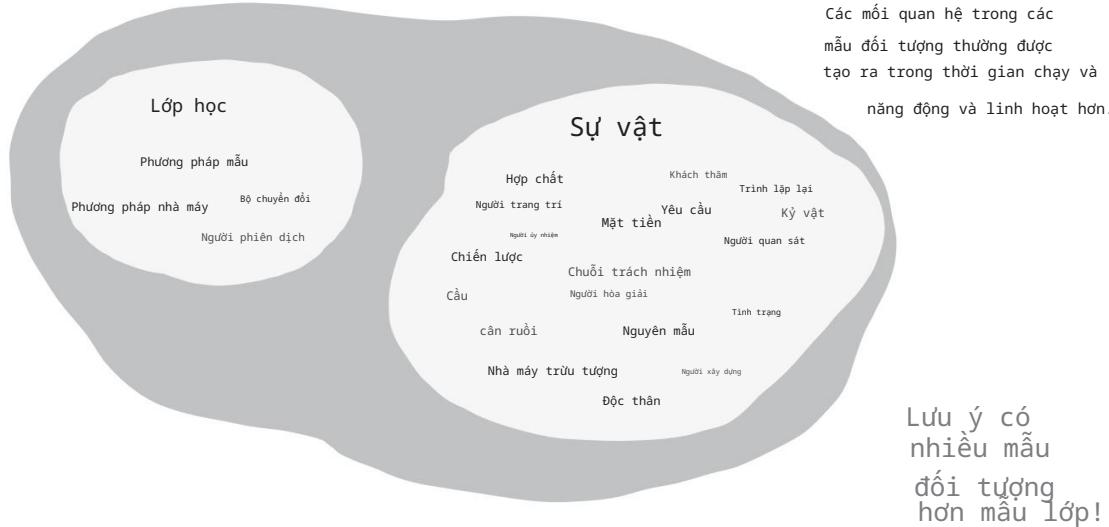
Các mẫu thường được phân loại theo thuộc tính thứ hai: mẫu đó có liên quan đến lớp hay đối tượng hay không:

Các mẫu lớp mô tả cách các mối quan hệ giữa các lớp được xác định thông qua kế thừa. Các mối quan hệ trong các mẫu lớp được thiết lập tại thời điểm biên dịch.

Các mẫu đối tượng mô tả mối quan hệ giữa các đối tượng và chủ yếu được xác định theo thành phần.

Các mối quan hệ trong các mẫu đối tượng thường được tạo ra trong thời gian chạy và

năng động và linh hoạt hơn.



Q: Đây có phải là phân loại duy nhất không?
kế hoạch?

A: Không, các chương trình khác đã được thực hiện được đề xuất. Một số chương trình khác bắt đầu với ba danh mục và sau đó thêm các danh mục phụ, như "Mẫu tách rời". Bạn sẽ muốn làm quen với các sơ đồ tổ chức mẫu phổ biến nhất, nhưng cũng có thể thoải mái tự tạo sơ đồ của riêng mình nếu điều đó giúp bạn hiểu rõ hơn về các mẫu.

Q: Việc tổ chức các mẫu thành

Các danh mục có thực sự giúp bạn nhớ chúng không?

không có Những câu hỏi ngớ ngẩn

A: Nó chắc chắn cung cấp cho bạn một khuôn khổ để so sánh. Nhưng nhiều người bị nhầm lẫn bởi các phạm trù sáng tạo, cấu trúc và hành vi; thường thì một mô hình có vẻ phù hợp với nhiều hơn một phạm trù. Điều quan trọng nhất là phải biết các mô hình và mối quan hệ giữa chúng. Khi các phạm trù giúp ích, hãy sử dụng chúng!

Q: Tại sao mẫu Decorator lại có trong phạm trù cấu trúc? Tôi nghĩ đó là một kiểu hành vi; sau cùng thì nó bổ sung thêm hành vi!

A: Có, rất nhiều nhà phát triển nói như vậy!

Sau đây là suy nghĩ đằng sau phân loại Gang of Four: các mẫu cấu trúc mô tả cách các lớp và đối tượng được kết hợp để tạo ra cấu trúc mới hoặc chức năng mới.

Mẫu trang trí cho phép bạn tạo các đối tượng bằng cách gói một đối tượng bằng một đối tượng khác để cung cấp chức năng mới.

Vì vậy, trọng tâm là cách bạn tạo ra các đối tượng một cách động để đạt được chức năng, thay vì giao tiếp và kết nối giữa các đối tượng, đó là mục đích của các mẫu hành vi.

Nhưng hãy nhớ rằng, mục đích của các mẫu này là khác nhau và đó thường là chìa khóa để hiểu một mẫu thuộc về loại nào.

danh mục mẫu



Thầy và trò...

Sư phụ: Châu chấu ơi, trông con có vẻ lo lắng.

Học viên: Vâng, em vừa mới học về phân loại
mẫu và em còn bối rối.

Sư phụ: Châu chấu, tiếp tục đi...

Học sinh: Sau khi tìm hiểu nhiều về các mô hình, tôi vừa được
cho biết rằng mỗi mô hình phù hợp với một trong ba phân loại: cầu
trúc, hành vi hoặc sáng tạo. Tại sao chúng ta cần những phân loại này?

Thầy: Châu chấu, bắt cứ khi nào chúng ta có một bộ sưu tập lớn
bất kỳ thứ gì, chúng ta thường tìm các danh mục để xếp những thứ đó
vào. Điều này giúp chúng ta nghĩ về các mục ở mức trừu tượng hơn.

Học trò: Thầy ơi, thầy có thể cho em một ví dụ được không?

Sư phụ: Tất nhiên rồi. Hãy lấy ô tô làm ví dụ; có rất nhiều mẫu ô
tô khác nhau và chúng ta tự nhiên phân loại chúng thành các
loại như xe phổ thông, xe thể thao, xe SUV, xe tải và xe sang
trọng.

Sư phụ: Châu chấu, con có vẻ ngạc nhiên, điều này không hợp lý
sao?

Học viên: Thưa thầy, điều đó rất có lý, nhưng em ngạc nhiên
khi thấy thầy lại biết nhiều về ô tô đến vậy!

Sư phụ: Châu chấu, ta không thể liên hệ mọi thứ với hoa sen hay
bát cớm. Vậy giờ, ta có thể tiếp tục không?

Học sinh: Vâng, vâng, em xin lỗi, xin anh hãy tiếp tục.

Thầy: Khi bạn đã phân loại hoặc xếp loại, bạn có thể dễ dàng
nói về các nhóm khác nhau: "Nếu bạn lái xe trên núi từ Thung
lũng Silicon đến Santa Cruz, một chiếc xe thể thao có khả năng xử
 lý tốt là lựa chọn tốt nhất." Hoặc, "Với tình hình dầu mỏ ngày càng
tồi tệ, bạn thực sự muốn mua một chiếc xe tiết kiệm nhiên liệu,
chúng tiết kiệm nhiên liệu hơn."

Học sinh: Vì vậy, bằng cách có các danh mục, chúng ta có thể nói
về một tập hợp các mẫu như một nhóm. Chúng ta có thể biết mình
cần một mẫu sáng tạo, mà không biết chính xác là mẫu nào, nhưng
chúng ta vẫn có thể nói về các mẫu sáng tạo.

Thầy: Đúng vậy, và nó cũng cung cấp cho chúng ta cách so sánh một
thành viên với phần còn lại của danh mục, ví dụ, "Mini thực sự là
chiếc xe nhỏ gọn thời trang nhất hiện nay" hoặc để thu hẹp phạm vi
tim kiếm của chúng ta, "Tôi cần một chiếc xe tiết kiệm nhiên liệu".

sống tốt hơn với các mẫu

Học viên: Em hiểu rồi, vậy em có thể nói rằng mẫu Adapter là mẫu cấu trúc tốt nhất để thay đổi giao diện của đối tượng.

Master: Vâng. Chúng ta cũng có thể sử dụng các danh mục cho một mục đích nữa: để ra mắt vào một lãnh thổ mới; ví dụ, "chúng tôi thực sự muốn cung cấp một chiếc xe thể thao với hiệu suất của Ferrari với giá Miata."

Học sinh: Nghe giống như một cái bẫy chết người vậy.

Sư phụ: Xin lỗi, ta không nghe rõ Châu Châu ạ.

Học sinh: Ủ, em nói "Em hiểu rồi."

Học sinh: Vậy thì các phạm trù cho chúng ta cách để suy nghĩ về cách các nhóm mẫu liên quan và cách các mẫu trong một nhóm liên quan đến nhau. Chúng cũng cho chúng ta cách để suy rộng ra các mẫu mới. Nhưng tại sao lại có ba phạm trù chứ không phải bốn hay năm?

Sư phụ: À, giống như những vì sao trên bầu trời đêm, có nhiều loại tùy theo bạn muốn xem. Ba là một con số tiện lợi và là con số mà nhiều người đã quyết định tạo nên một nhóm mẫu đẹp. Nhưng những người khác lại để xuất bốn, năm hoặc nhiều hơn.



bạn đang ở đây 4 593

suy nghĩ theo khuôn mẫu

Suy nghĩ theo mẫu

Bối cảnh, ràng buộc, lực lượng, danh mục, phân loại... trời ạ, điều này bắt đầu nghe có vẻ học thuật ghê gớm. Được rồi, tất cả những thứ đó đều quan trọng và kiến thức là sức mạnh. Nhưng, hãy đổi mặt với nó, nếu bạn hiểu những thứ học thuật và không có kinh nghiệm và thực hành sử dụng các khuôn mẫu, thì nó sẽ không tạo ra nhiều khác biệt trong cuộc sống của bạn.

Đây là hướng dẫn nhanh giúp bạn bắt đầu suy nghĩ theo các mẫu. Chúng tôi muốn nói gì khi nói như vậy? Chúng tôi muốn nói đến khả năng nhìn vào một thiết kế và thấy được các mẫu phù hợp tự nhiên ở đâu và không phù hợp ở đâu.



Bộ não của bạn về các mẫu

Giữ cho nó đơn giản (KISS)

Trước hết, khi bạn thiết kế, hãy giải quyết mọi thứ theo cách đơn giản nhất có thể. Mục tiêu của bạn phải là sự đơn giản, chứ không phải là "làm thế nào tôi có thể áp dụng một mẫu cho vấn đề này". Đừng cảm thấy mình không phải là một nhà phát triển tinh vi nếu bạn không sử dụng một mẫu để giải quyết vấn đề. Các nhà phát triển khác sẽ đánh giá cao và ngưỡng mộ sự đơn giản trong thiết kế của bạn. Tuy nhiên, đôi khi cách tốt nhất để giữ cho thiết kế của bạn đơn giản và linh hoạt là sử dụng một mẫu.

Các mẫu thiết kế không phải là giải pháp thần kỳ; thực tế chúng thậm chí không phải là một viên đạn!

Các mẫu, như bạn biết, là giải pháp chung cho các vấn đề thường gặp. Các mẫu cũng có lợi thế là được nhiều nhà phát triển kiểm tra kỹ lưỡng. Vì vậy, khi bạn thấy cần một mẫu, bạn có thể yên tâm vì biết rằng nhiều nhà phát triển đã từng trải qua và giải quyết vấn đề bằng các kỹ thuật tương tự.

Tuy nhiên, các mẫu không phải là viên đạn thần kỳ. Bạn không thể cắm một mẫu vào, biên dịch rồi ăn trưa sớm. Để sử dụng các mẫu, bạn cũng cần phải suy nghĩ về hậu quả đối với phần còn lại của thiết kế.

Bạn biết bạn cần một khuôn mẫu khi...

À... câu hỏi quan trọng nhất: khi nào bạn sử dụng một mẫu? Khi bạn tiếp cận thiết kế của mình, hãy giới thiệu một mẫu khi bạn chắc chắn rằng nó giải quyết được vấn đề trong thiết kế của bạn. Nếu một giải pháp đơn giản hơn có thể hiệu quả, hãy cân nhắc điều đó trước khi bạn cam kết sử dụng một mẫu.

Biết khi nào một mẫu áp dụng là lúc kinh nghiệm và kiến thức của bạn phát huy tác dụng. Khi bạn chắc chắn rằng một giải pháp đơn giản sẽ không đáp ứng được nhu cầu của mình, bạn nên xem xét vấn đề cùng với tập hợp các ràng buộc mà giải pháp sẽ cần phải hoạt động – những điều này sẽ giúp bạn khớp vấn đề của mình với một mẫu. Nếu bạn có kiến thức tốt về các mẫu, bạn có thể biết một mẫu phù hợp. Nếu không, hãy khảo sát các mẫu có vẻ như có thể giải quyết được vấn đề. Các phần về mục đích và khả năng áp dụng của danh mục mẫu đặc biệt hữu ích cho việc này. Khi bạn đã tìm thấy một mẫu có vẻ phù hợp, hãy đảm bảo rằng nó có một tập hợp các hậu quả mà bạn có thể chấp nhận và nghiên cứu tác động của nó đối với phần còn lại của thiết kế. Nếu mọi thứ có vẻ ổn, hãy thực hiện!

Có một tinh huống mà bạn sẽ muốn sử dụng một mẫu ngay cả khi một giải pháp đơn giản hơn có thể hiệu quả: khi bạn mong đợi các khía cạnh của hệ thống của mình thay đổi. Như chúng ta đã thấy, việc xác định các lĩnh vực thay đổi trong thiết kế của bạn thường là một dấu hiệu tốt cho thấy cần có một mẫu. Chỉ cần đảm bảo rằng bạn đang thêm các mẫu để giải quyết những thay đổi thực tế có khả năng xảy ra, chứ không phải thay đổi giả định điều đó có thể xảy ra.

Thời gian thiết kế không phải là thời gian duy nhất bạn muốn cân nhắc giới thiệu các mẫu, bạn cũng nên làm như vậy khi tái cấu trúc.

Thời gian tái cấu trúc chính là thời gian của các mẫu!

Tái cấu trúc là quá trình thực hiện các thay đổi đối với mã của bạn để cải thiện cách thức tổ chức mã. Mục tiêu là cải thiện cấu trúc mã, không phải thay đổi hành vi của mã. Đây là thời điểm tuyệt vời để xem xét lại thiết kế của bạn để xem liệu nó có thể được cấu trúc tốt hơn bằng các mẫu hay không. Ví dụ, mã chứa đầy các câu lệnh có điều kiện có thể báo hiệu nhu cầu về mẫu State. Hoặc, có thể đã đến lúc dọn dẹp các phụ thuộc cụ thể bằng Factory. Toàn bộ các cuốn sách đã được viết về chủ đề tái cấu trúc bằng các mẫu và kỹ năng của bạn phát triển, bạn sẽ muốn nghiên cứu thêm về lĩnh vực này.

Loại bỏ những gì bạn không thực sự cần. Đừng ngại xóa Mẫu thiết kế khỏi thiết kế của bạn.

Không ai từng nói về thời điểm xóa một mẫu. Bạn sẽ nghĩ đó là sự báng bổ! Không, chúng ta đều là người lớn ở đây; chúng ta có thể chấp nhận.

Vậy khi nào bạn xóa một mẫu? Khi hệ thống của bạn trở nên phức tạp và tinh linh hoạt mà bạn đã lên kế hoạch không còn cần thiết nữa. Nói cách khác, khi một giải pháp đơn giản hơn mà không có mẫu sẽ tốt hơn.

Nếu bạn không cần làm ngay bây giờ thì đừng làm nữa.

Mẫu thiết kế rất mạnh mẽ và dễ dàng thấy được mọi cách có thể sử dụng chúng trong các thiết kế hiện tại của bạn. Các nhà phát triển thường thích tạo ra những kiến trúc đẹp mắt, sành sảng thay đổi theo mọi hướng.

Hãy kiềm chế sự cảm động. Nếu bạn có nhu cầu thực tế là hỗ trợ thay đổi trong thiết kế ngày hôm nay, hãy tiếp tục và sử dụng một mẫu để xử lý thay đổi đó.

Tuy nhiên, nếu lý do chỉ là giả thuyết thì đừng thêm mẫu, nó chỉ làm hệ thống của bạn phức tạp hơn và bạn có thể không bao giờ cần đến nó!



các mẫu hình xuất hiện một cách tự nhiên



Thầy và trò...

Master: Grasshopper, khóa huấn luyện ban đầu của bạn đã gần hoàn tất. Bạn có kế hoạch gì?

Học sinh: Tôi sẽ đến Disneyland! Và sau đó tôi sẽ bắt đầu tạo nhiều mã với các mẫu!

Sư phụ: Ô, khanh đã. Đừng bao giờ sử dụng vũ khí lớn trừ khi bắt buộc.

Học viên: Ý thầy là sao? Bây giờ em đã học được các mẫu thiết kế rồi, em không nên sử dụng chúng trong mọi thiết kế của mình để đạt được sức mạnh, tinh linh hoạt và khả năng quản lý tối đa sao?

Master: Không; các mẫu là một công cụ và là một công cụ chỉ nên sử dụng khi cần thiết. Bạn cũng đã dành nhiều thời gian để học các nguyên tắc thiết kế.

Luôn bắt đầu từ các nguyên tắc của bạn và tạo ra mã đơn giản nhất có thể để thực hiện công việc. Tuy nhiên, nếu bạn thấy cần phải có một mẫu, hãy sử dụng nó.

Học sinh: Vậy em không nên xây dựng thiết kế của mình theo mẫu sao?

Master: Đó không phải là mục tiêu của bạn khi bắt đầu thiết kế. Hãy để các mẫu xuất hiện một cách tự nhiên khi thiết kế của bạn tiến triển.

Học sinh: Nếu các mẫu mã tuyệt vời như vậy, tại sao em phải cẩn thận khi sử dụng chúng?

Master: Các mẫu có thể tạo ra sự phức tạp, và chúng ta không bao giờ muốn sự phức tạp ở nơi không cần thiết. Nhưng các mẫu rất mạnh mẽ khi được sử dụng ở nơi cần thiết. Như bạn đã biết, các mẫu là kinh nghiệm thiết kế đã được chứng minh có thể được sử dụng để tránh các lỗi thường gặp. Chúng cũng là một vốn từ vựng chung để truyền đạt thiết kế của chúng ta cho người khác.

Học sinh: Vậy khi nào chúng ta biết được thời điểm thích hợp để áp dụng các mẫu thiết kế?

Thạc sĩ: Giới thiệu một mẫu khi bạn chắc chắn rằng nó cần thiết để giải quyết một vấn đề trong thiết kế của bạn hoặc khi bạn khá chắc chắn rằng nó cần thiết để giải quyết sự thay đổi trong tương lai về các yêu cầu của ứng dụng.

Học sinh: Em đoán là việc học của em vẫn sẽ tiếp tục mặc dù em đã hiểu rất nhiều mô hình.

Master: Đúng vậy, grasshopper; học cách quản lý sự phức tạp và thay đổi trong phần mềm là một quá trình theo đuổi suốt đời. Nhưng giờ bạn đã biết một bộ mẫu tốt, đã đến lúc áp dụng chúng vào những nơi cần thiết trong thiết kế của bạn và tiếp tục học thêm nhiều mẫu hơn.

Học sinh: Đợi đã, ý thầy là em không biết TẤT CẢ họ sao?

Thầy: Grasshopper, con đã học các mẫu cơ bản; con sẽ thấy còn nhiều mẫu khác nữa, bao gồm các mẫu chỉ áp dụng cho các miền cụ thể như hệ thống dòng thời và hệ thống doanh nghiệp.

Nhưng bây giờ bạn đã biết những điều cơ bản, bạn có thể thoải mái học chúng!

sống tốt hơn với các mẫu

Tâm trí của bạn về các mẫu



Tâm trí người mới bắt đầu

Người mới bắt đầu sử dụng các mẫu ở mọi nơi. Điều này là tốt: người mới bắt đầu có nhiều kinh nghiệm và thực hành sử dụng các mẫu. Người mới bắt đầu cũng nghĩ rằng, "Tôi sử dụng càng nhiều mẫu, thiết kế càng tốt". Người mới bắt đầu sẽ học được rằng điều này không đúng, rằng tất cả các thiết kế nên càng đơn giản càng tốt. Sự phức tạp và các mẫu chỉ nên được sử dụng khi cần thiết cho khả năng mở rộng thực tế.

"Tôi cần một mẫu cho Hello World."



Khi quá trình học tiến triển, trí tuệ trung gian bắt đầu thấy được nơi nào cần có các mẫu và nơi nào không. Trí tuệ trung gian vẫn cố nhớ quá nhiều mẫu hình vuông vào lỗ tròn, nhưng cũng bắt đầu thấy rằng các mẫu có thể được điều chỉnh để phù hợp với các tinh huống mà mẫu hình chuẩn không phù hợp.

Trung cấp
Tâm trí

"Có lẽ tôi cần một Singleton ở đây."



Tâm Thiền

"Đây là nơi lý tưởng cho Decorator."

Tâm Thiền có khả năng nhìn thấy những khuôn mẫu phù hợp một cách tự nhiên. Tâm trí Thiền không bị ám ảnh bởi việc sử dụng các mô hình; thay vào đó, nó tìm kiếm các giải pháp đơn giản để giải quyết vấn đề tốt nhất. Tâm trí Thiền suy nghĩ theo các nguyên tắc đối tượng và sự đánh đổi của chúng. Khi nhu cầu về một mẫu tự nhiên này sinh, tâm trí Thiền áp dụng nó khi biết rõ rằng nó có thể cần phải thích nghi. Tâm trí Thiền cũng nhìn thấy mối quan hệ với các mẫu tương tự và hiểu được sự tinh tế của sự khác biệt trong ý định của các mẫu liên quan. Tâm trí Thiền cũng là tâm trí của Người mới bắt đầu – nó không để tắt cả kiến thức về mẫu ánh hưởng quá mức đến các quyết định thiết kế.

khi nào không nên sử dụng mẫu

CẢNH BÁO: Việc sử dụng quá nhiều các mẫu thiết kế có thể dẫn đến mã được thiết kế quá mức. Luôn luôn sử dụng giải pháp đơn giản nhất có thể thực hiện được công việc và giới thiệu các mẫu ở nơi cần thiết xuất hiện.



Tất nhiên là chúng tôi muốn bạn sử dụng Mẫu thiết kế!

Nhưng chúng tôi muốn bạn trở thành một nhà thiết kế OO giỏi hơn.

Khi một giải pháp thiết kế yêu cầu một mẫu, bạn sẽ nhận được lợi ích khi sử dụng một giải pháp đã được nhiều nhà phát triển kiểm tra theo thời gian. Bạn cũng đang sử dụng một giải pháp được ghi chép đầy đủ và các nhà phát triển khác sẽ nhận ra (Bạn biết đấy, toàn bộ thứ từ vựng được chia sẻ đó).

Tuy nhiên, khi bạn sử dụng Design Patterns, cũng có thể có một nhược điểm. Design Patterns thường giới thiệu các lớp và đối tượng bổ sung, do đó chúng có thể làm tăng độ phức tạp của thiết kế của bạn.

Các mẫu thiết kế cũng có thể thêm nhiều lớp hơn vào thiết kế của bạn, điều này không chỉ làm tăng thêm tính phức tạp mà còn làm giảm hiệu quả.

Ngoài ra, đôi khi sử dụng Mẫu thiết kế có thể là quá mức cần thiết. Nhiều lần bạn có thể quay lại với các nguyên tắc thiết kế của mình và tìm ra giải pháp đơn giản hơn nhiều để giải quyết cùng một vấn đề. Nếu điều đó xảy ra, đừng chống lại nó. Hãy sử dụng giải pháp đơn giản hơn.

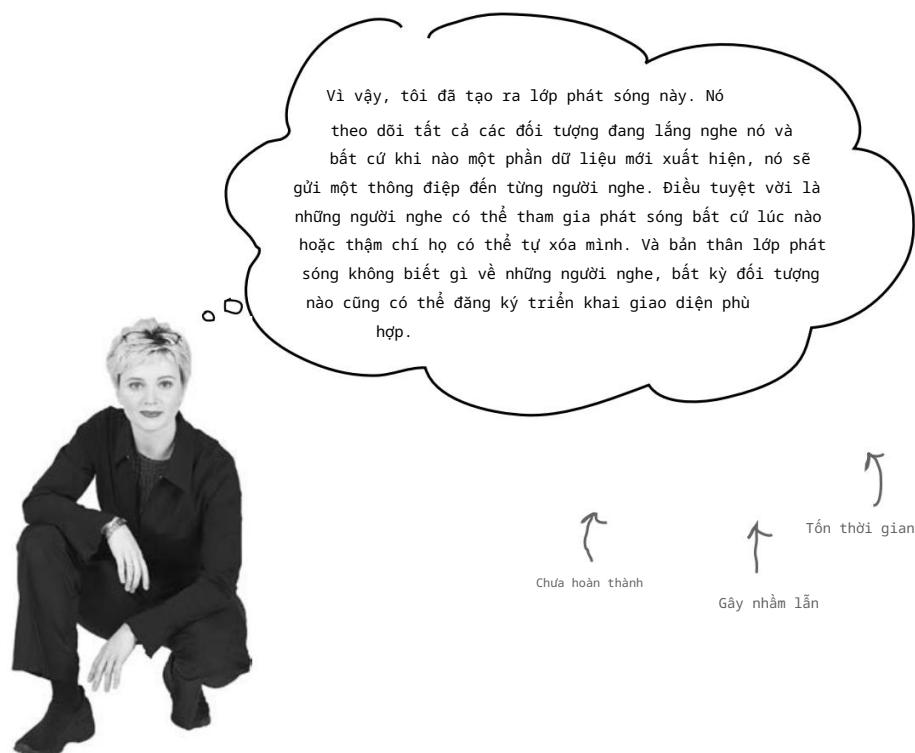
Tuy nhiên, đừng để chúng tôi làm bạn nản lòng. Khi Design Pattern là công cụ phù hợp cho công việc, nó có rất nhiều lợi thế.

Đừng quên sức mạnh của vốn từ vựng chung

Chúng ta đã dành quá nhiều thời gian trong cuốn sách này để thảo luận về các vấn đề cốt lõi của OO đến nỗi chúng ta dễ quên mất khía cạnh con người của Design Patterns - chúng không chỉ giúp nạp vào não bạn các giải pháp mà còn cung cấp cho bạn vốn từ vựng chung với các nhà phát triển khác. Đừng đánh giá thấp sức mạnh của vốn từ vựng chung, đó là một trong những lợi ích lớn nhất của Design Patterns.

Hãy nghĩ xem, có điều gì đó đã thay đổi kể từ lần cuối chúng ta nói về vốn từ vựng chung; giờ đây bạn đã bắt đầu xây dựng được một vốn từ vựng kha khá cho riêng mình! Chưa kể, bạn còn được học toàn bộ các nguyên tắc thiết kế hướng đối tượng, từ đó bạn có thể dễ dàng hiểu được động cơ và cách thức hoạt động của bất kỳ mô hình mới nào mà bạn gặp phải.

Bây giờ bạn đã nắm được những điều cơ bản về Design Pattern, đã đến lúc bạn ra ngoài và truyền bá thông tin này đến những người khác. Tại sao? Bởi vì khi các nhà phát triển đồng nghiệp của bạn biết về các mẫu và cũng sử dụng chung một vốn từ vựng, điều đó sẽ dẫn đến các thiết kế tốt hơn, giao tiếp tốt hơn và, tuyệt vời nhất là, nó sẽ giúp bạn tiết kiệm rất nhiều thời gian để dành cho những thứ thú vị hơn.



năm cách để chia sẻ vốn từ vựng của bạn

Năm cách hàng đầu để chia sẻ vốn từ vựng của bạn

- 1 Trong các cuộc họp thiết kế: Khi bạn họp với nhóm của mình để thảo luận về thiết kế phần mềm, hãy sử dụng các mẫu thiết kế để giúp duy trì "thiết kế" lâu hơn. Thảo luận về thiết kế theo quan điểm của các Mẫu thiết kế và các nguyên tắc 00 giúp nhóm của bạn không bị sa lầy vào các chi tiết triển khai và ngăn ngừa nhiều hiểu lầm.
- 2 Với các nhà phát triển khác: Sử dụng các mẫu trong các cuộc thảo luận của bạn với các nhà phát triển khác. Điều này giúp các nhà phát triển khác tìm hiểu về các mẫu mới và xây dựng một cộng đồng. Phản tuyệt vời nhất khi chia sẻ những gì bạn đã học được là cảm giác tuyệt vời khi người khác "hiểu được!"
- 3 Trong tài liệu kiến trúc: Khi bạn viết tài liệu kiến trúc, việc sử dụng các mẫu sẽ giảm lượng tài liệu bạn cần viết và cung cấp cho người đọc hình ảnh rõ ràng hơn về thiết kế.
- 4 Trong chú thích mã và quy ước đặt tên: Khi bạn viết mã, hãy xác định rõ các mẫu bạn đang sử dụng trong chú thích. Ngoài ra, hãy chọn tên lớp và phương thức để bộ nhớ nào bên dưới. Các nhà phát triển khác phải đọc mã của bạn sẽ cảm ơn bạn vì đã cho phép họ nhanh chóng hiểu được cách triển khai của bạn.
- 5 Đến các nhóm nhà phát triển quan tâm: Hãy chia sẻ kiến thức của bạn. Nhiều nhà phát triển đã nghe về các mẫu nhưng không hiểu rõ chúng là gì. Hãy tình nguyện tổ chức một bữa trưa nhẹ về các mẫu hoặc nói chuyện tại nhóm người dùng địa phương của bạn.



sống tốt hơn với các mẫu

Du ngoạn Objectville cùng Gang of Four

Bạn sẽ không tìm thấy Jets hay Sharks quanh Objectville, nhưng bạn sẽ tìm thấy Gang of Four. Như bạn có thể đã nhận thấy, bạn không thể đi xa trong World of Patterns mà không chạm trán với họ.

Vậy, băng đảng bí ẩn này là ai?

Nói một cách đơn giản, "GoF", bao gồm Erich Gamma, Richard Helm, Ralph Johnson và John Vlissides, là nhóm những người đã cùng nhau lập ra danh mục mẫu đầu tiên và trong quá trình đó, đã bắt đầu một phong trào hoàn toàn mới trong lĩnh vực phần mềm!

Họ lấy cái tên đó bằng cách nào? Không ai biết chắc; đó chỉ là cái tên gắn liền với họ. Nhưng hãy nghĩ xem: nếu bạn định để một "thành phần băng đảng" chạy quanh Objectville, bạn có thể nghĩ ra một nhóm người tốt bụng hơn không? Trên thực tế, họ thậm chí còn đồng ý đến thăm chúng ta...

GoF đã phát động phong trào
mẫu phần mềm, nhưng nhiều người
khác cũng có những đóng góp đáng
kể, bao gồm Ward Cunningham,
Kent Beck, Jim Coplien, Grady Booch,
Bruce Anderson, Richard Gabriel,
Doug Lea, Peter Coad và Doug
Schmidt, chỉ kể tên một vài người.

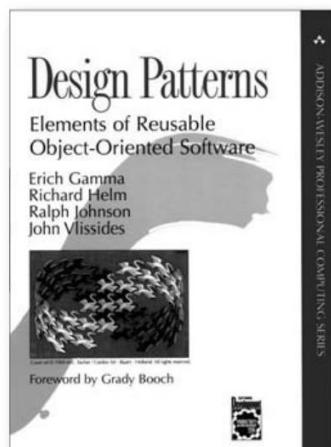


bạn đang ở đây 4 601

mẫu tài nguyên

Hành trình của bạn vừa mới bắt đầu...

Bây giờ bạn đã nắm được các Mẫu thiết kế và sẵn sàng tìm hiểu sâu hơn, chúng tôi có ba văn bản chính mà bạn cần thêm vào giá sách của mình...



Văn bản mẫu thiết kế xác định

Đây là cuốn sách đã khởi đầu cho toàn bộ lĩnh vực Design Patterns khi nó được phát hành vào năm 1995. Bạn sẽ tìm thấy tất cả các mẫu cơ bản ở đây. Trên thực tế, cuốn sách này là cơ sở cho bộ mẫu mà chúng tôi sử dụng trong Head First Design Patterns.

Bạn sẽ không thấy cuốn sách này là lời cuối cùng về Mẫu thiết kế - lĩnh vực này đã phát triển đáng kể kể từ khi xuất bản - nhưng đây là cuốn đầu tiên và mang tính quyết định nhất.

Chọn một bàn sao của Design Patterns là một cách tuyệt vời để bắt đầu khám phá các mẫu sau khóa học Head First.

Các tác giả của Design Patterns được biết đến với cái tên triết mến là "Băng nhóm bốn người" hoặc viết tắt là GoF.

Christopher Alexander đã phát minh ra các mẫu hình, tạo cảm hứng cho việc áp dụng các giải pháp tương tự vào phần mềm.

Các văn bản mẫu xác định

Các mẫu không bắt đầu từ GoF; chúng bắt đầu từ Christopher Alexander, Giáo sư Kiến trúc tại Berkeley - đúng vậy, Alexander là một kiến trúc sư, không phải là nhà khoa học máy tính. Alexander đã phát minh ra các mẫu để xây dựng kiến trúc sống (như nhà ở, thị trấn và thành phố).

Lần tới khi bạn muốn đọc một cuốn sách sâu sắc và hấp dẫn, hãy chọn The Timeless Way of Building. Bạn sẽ thấy khởi đầu thực sự của Mẫu thiết kế và nhận ra sự tương tự trực tiếp giữa việc tạo ra "kiến trúc sống" và phần mềm linh hoạt, có thể mở rộng.

Vậy hãy lấy một tách cà phê Starbuzz, ngồi xuống và thưởng thức...



sống tốt hơn với các mẫu

Các nguồn tài nguyên về mẫu thiết kế khác Bạn sẽ thấy có một cộng đồng người dùng và người viết mẫu thiết kế sôi động, thân thiện và họ rất vui khi bạn tham gia cùng họ.
Sau đây là một số tài nguyên giúp bạn bắt đầu...



Trang web

Portland Patterns Repository, do Ward Cunningham điều hành, là một WIKI dành riêng cho mọi thứ liên quan đến mẫu. Bất kỳ ai cũng có thể tham gia. Bạn sẽ tìm thấy các chủ đề thảo luận về mọi chủ đề bạn có thể nghĩ đến liên quan đến mô hình và hệ thống OO.

<http://c2.com/cgi/wiki?WelcomeVisitors>

Hillside Group thúc đẩy các hoạt động lập trình và thiết kế chung và cung cấp một nguồn tài nguyên trung tâm cho công việc về mẫu. Trang web bao gồm thông tin về nhiều nguồn tài nguyên liên quan đến mẫu như bài viết, sách, danh sách gửi thư và công cụ.

<http://hillside.net/>



Hội nghị và Hội thảo

Và nếu bạn muốn có thời gian gặp mặt trực tiếp với cộng đồng mẫu thiết kế, hãy nhớ tham gia nhiều hội nghị và hội thảo liên quan đến mẫu thiết kế.

Trang web Hillside duy trì danh sách đầy đủ. Ít nhất bạn sẽ muốn xem OOPSLA, Hội nghị ACM về Hệ thống, Ngôn ngữ và Ứng dụng hướng đối tượng.

bạn đang ở đây 4 603

mẫu sở thú

Vườn thú Patterns

Như bạn vừa thấy, các mẫu không bắt đầu bằng phần mềm; chúng bắt đầu bằng kiến trúc của các tòa nhà và thị trấn. Trên thực tế, khái niệm mẫu có thể được áp dụng trong nhiều lĩnh vực khác nhau. Hãy đi dạo quanh Sở thú mẫu để xem một số...



Các mẫu kiến trúc được sử dụng
để tạo nên kiến trúc sống
động cho các tòa nhà, thi
trấn và thành phố.
Đây chính là nơi các mẫu hình bắt
đầu xuất hiện.

Môi trường sống: được tìm thấy trong các tòa
nhà mà bạn thích sống, thích nhìn và thích ghé thăm.

Môi trường sống: thường xuất hiện
xung quanh các kiến trúc 3 tầng,
hệ thống máy khách-máy chủ và web.

Mẫu ứng dụng là mẫu dùng để
tạo kiến trúc cấp hệ thống.



Nhiều kiến trúc đa
tầng thuộc loại này.

Ghi chú: MVC được
biết đến là một
mô hình ứng dụng.



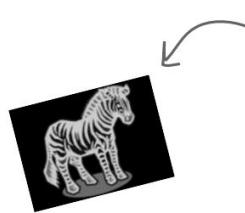
Mẫu dành riêng cho miền là
mẫu liên quan đến các vấn
đề trong miền cụ thể, như hệ
thống đồng thời hoặc hệ
thống thời gian thực.

Giúp tìm môi trường sống

J2EE

sống tốt hơn với các mẫu

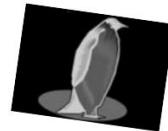
Mẫu quy trình kinh doanh mô tả
sự tương tác giữa doanh
nghiệp, khách hàng và dữ liệu và có
thể được áp dụng cho các vấn đề
như cách đưa ra và truyền đạt
quyết định hiệu quả.



Thường xuyên lui tới các phòng
hợp của công ty và các
cuộc họp quản lý dự án.

- Giúp tìm môi trường sống
-
- Đội ngũ phát triển
-
- Đội ngũ hỗ trợ khách hàng
-
-

Các mô hình tổ chức mô tả các
cấu trúc và hoạt động của các
tổ chức con người. Hầu hết
các nỗ lực cho đến nay
đều tập trung vào các tổ
chức sản xuất và/hoặc hỗ trợ
phản mềm.



Các mẫu thiết kế
giao diện người
dùng giải
quyết các vấn đề về
cách thiết kế các
chương trình phần mềm tương tác.

Môi trường sống: thường thấy ở gần
các nhà thiết kế trò chơi điện tử,
người xây dựng GUI và nhà sản xuất.

Ghi chú thực địa: vui lòng thêm quan sát của bạn về miền mẫu ở đây:

bạn đang ở đây 4 605

mẫu chống đối

Tiêu diệt cái ác bằng Anti-Patterns

Vũ trụ sẽ không thể hoàn thiện nếu chúng ta chỉ có các khuôn mẫu và không có phản khuôn mẫu, đúng không?

Nếu một Mẫu thiết kế cung cấp cho bạn giải pháp chung cho một vấn đề thường gặp trong một bối cảnh cụ thể thì một phản mẫu sẽ cung cấp cho bạn điều gì?

Một mô hình phản biện cho bạn biết cách chuyển từ một vấn đề sang một giải pháp KHÔNG TỐT.

Có lẽ bạn đang tự hỏi, "Tại sao người ta lại tốn thời gian vào việc ghi chép lại những giải pháp tồi?"

Hãy nghĩ về điều này như thế này: nếu có một giải pháp tệ hại lặp đi lặp lại cho một vấn đề phổ biến, thì bằng cách ghi chép lại, chúng ta có thể ngăn chặn các nhà phát triển khác mắc phải lỗi tương tự. Sau cùng, tránh các giải pháp tệ hại cũng có giá trị như tìm ra giải pháp tốt!

Hãy cùng xem xét các yếu tố của một mô hình phản diện:

Một mẫu phản biện cho bạn biết tại sao một giải pháp tồi lại hấp dẫn. Hãy nhìn nhận thực tế, sẽ không ai chọn một giải pháp tồi nếu không có điều gì đó hấp dẫn ngay từ đầu.

Một trong những nhiệm vụ lớn nhất của mô hình phản biện là cảnh báo bạn về khía cạnh hấp dẫn của giải pháp.

Một mẫu phản biện cho bạn biết tại sao giải pháp đó về lâu dài lại tệ. Để hiểu tại sao nó là một mẫu phản biện, bạn phải hiểu nó sẽ có tác động tiêu cực như thế nào trong tương lai. Mẫu phản biện mô tả nơi bạn sẽ gặp rắc rối khi sử dụng giải pháp.

Một mẫu phản đè xuất các mẫu khác có thể áp dụng và có thể cung cấp các giải pháp tốt. Để thực sự hữu ích, một mẫu phản cần chỉ cho bạn đúng hướng; nó phải đè xuất các khả năng khác có thể dẫn đến các giải pháp tốt.

Chúng ta hãy cùng xem xét một mô hình phản biện.



Một mô hình phản biện luôn có vẻ như là một giải pháp tốt, nhưng sau đó lại trở thành một giải pháp tồi khi nó được áp dụng.

Bằng cách ghi lại các phản mô hình, chúng ta giúp người khác nhận ra các giải pháp tồi trước khi họ triển khai chúng.

Giống như các mẫu, có nhiều loại phản mẫu bao gồm các phản mẫu phát triển, 00, tổ chức và phản mẫu theo miền cụ thể.

sóng tốt hơn với các mẫu

Đây là một ví dụ về phản mô hình phát triển phần mềm.

Giống như một Mẫu thiết kế, một phản mẫu cũng có tên để chúng ta có thể tạo ra một vốn từ vựng chung.

Vấn đề và bối cảnh giống như mô tả của Mẫu thiết kế.

Cho bạn biết lý do tại sao giải pháp này hấp dẫn.

Giải pháp tồi nhưng lại hấp dẫn.

Làm thế nào để có được giải pháp tốt.

Ví dụ về nơi mà mô hình phản diện này được quan sát.

Được chuyển thể từ WIKI của Portland Pattern Repository tại <http://c2.com/>, nơi bạn sẽ tìm thấy nhiều mẫu phản đối và thảo luận.



Mẫu chông

Tên: Golden Hammer

Vấn đề: Bạn cần lựa chọn công nghệ cho quá trình phát triển của mình và bạn tin rằng chỉ có một công nghệ duy nhất chiếm ưu thế trong kiến trúc.

Bối cảnh: Bạn cần phát triển một số hệ thống hoặc phần mềm mới không phù hợp với công nghệ mà nhóm phát triển đã quen thuộc.

Lực lượng:

- Nhóm phát triển cam kết sử dụng công nghệ mà họ biết.
- Nhóm phát triển không quen thuộc với các công nghệ khác.
- Các công nghệ không quen thuộc được coi là rủi ro.
- Dễ dàng lập kế hoạch và ước tính cho quá trình phát triển bằng công nghệ quen thuộc.

Giải pháp được cho là: Sử dụng công nghệ quen thuộc. Công nghệ được áp dụng một cách ám ảnh vào nhiều vấn đề, bao gồm cả những nơi rõ ràng là không phù hợp.

Giải pháp tái cấu trúc: Mở rộng kiến thức của các nhà phát triển thông qua các nhóm giáo dục, đào tạo và nghiên cứu sách giúp các nhà phát triển tiếp cận với các giải pháp mới.

Ví dụ:

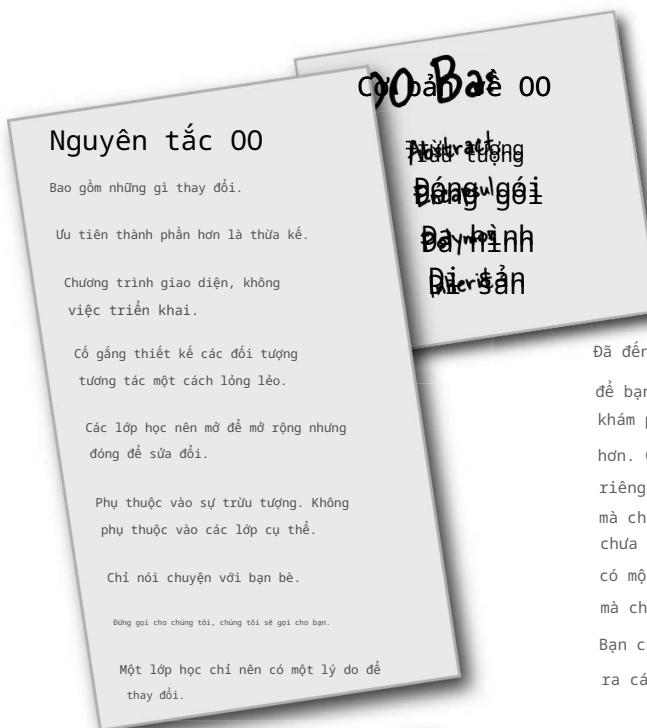
Các công ty web vẫn tiếp tục sử dụng và duy trì hệ thống lưu trữ đám mây riêng họ khi sử dụng các giải pháp thay thế nguồn mở.

hộp công cụ thiết kế



Công cụ cho hộp công cụ thiết kế của bạn

Bạn đã đạt đến điểm mà bạn không còn phù hợp với chúng tôi nữa
Bây giờ là lúc bạn bước ra thế giới và tự mình khám phá
các mô hình...



Đã đến lúc rồi
để bạn có thể tự mình
khám phá thêm nhiều mẫu
hơn. Có nhiều mẫu dành
riêng cho từng miền
mà chúng tôi thậm chí
chưa đề cập đến và cũng
có một số mẫu cơ bản
mà chúng tôi chưa đề cập đến.
Bạn cũng có thể tự tạo
ra các mẫu của riêng mình.

Hãy xem phần
Phụ lục, chúng
tôi sẽ cung cấp
cho bạn thông
tin chi tiết
về một số mẫu
cơ bản mà bạn có
thể muốn xem qua

ĐIỂM ĐẦU TIÊN



β Hãy để các Mẫu thiết kế xuất hiện trong thiết kế của bạn, đừng áp đặt chúng chỉ vì mục đích sử dụng một mẫu thiết kế.

B Các mẫu thiết kế không được thiết lập trong
đá; điều chỉnh và tinh chỉnh chúng để
đáp ứng nhu cầu của bạn.

Luôn sử dụng giải pháp đơn giản nhất đáp ứng được nhu cầu của bạn, ngay cả khi nó không có mẫu.

β Nghiên cứu danh mục Mẫu thiết kế để làm quen với các mẫu và mối quan hệ giữa chúng.

β Phân loại mẫu (hoặc các danh mục) cung cấp các nhóm cho các mẫu. Khi chúng hữu ích, hãy sử dụng chúng.

Bạn cần phải cam kết trở thành người viết mẫu: điều này cần có thời gian và sự kiên nhẫn, và bạn phải săn sìng tinh chỉnh nhiều.

B Hãy nhớ rằng, hầu hết các mẫu bạn gặp phải đều là phiên bản cải biên của các mẫu hiện có chứ không phải mẫu mới.

B Xây dựng nhóm chia sẻ của bạn
từ vựng. Đây là một trong những lợi
ích mạnh mẽ nhất của việc sử dụng các
mẫu.

B Giống như bất kỳ công đồng nào,
công đồng mẫu có thuật ngữ riêng.
Đừng để điều đó cản trở bạn. Sau khi
đọc cuốn sách này, giờ bạn đã biết
hầu hết rồi.

sống tốt hơn với các mẫu

Rời khỏi Objectville...



Thật tuyệt khi được gặp bạn ở Objectville.

Chúng tôi chắc chắn sẽ nhớ bạn. Nhưng đừng lo lắng - trước khi bạn biết điều đó,
cuốn sách Head First tiếp theo sẽ ra mắt và bạn có thể ghé thăm lại. Bạn hỏi cuốn
sách tiếp theo là gì? Hmm, câu hỏi hay! Tại sao bạn không giúp chúng tôi quyết định?
Gửi email đến booksuggestions@wickedlysmart.com.

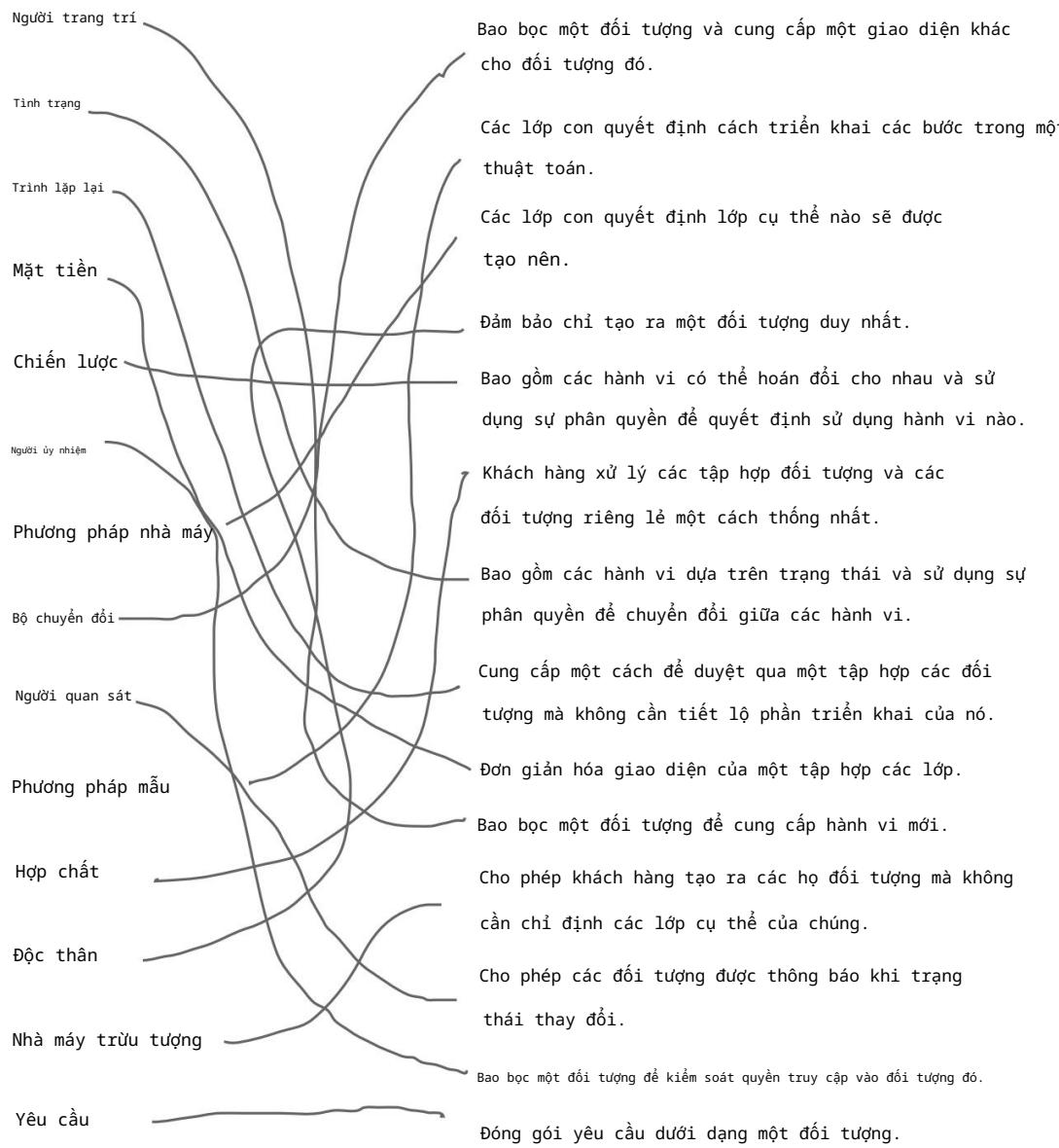
bạn đang ở đây 4 609

ai làm gì? giải pháp

Giải pháp bài tập



Ghép mỗi mẫu với mô tả của nó:

MẫuSự miêu tả

14 Phụ lục

Phụ lục: Các mẫu còn sót lại



Không phải ai cũng có thể là người nổi tiếng nhất. Rất nhiều thứ đã thay đổi 10 năm qua. Kể từ khi Design Patterns: Các yếu tố của Reusable Object-Oriented Phần mềm đầu tiên ra đời, các nhà phát triển đã áp dụng các mẫu này hàng nghìn lần của thời gian. Các mẫu chúng tôi tóm tắt trong phần phụ lục này là các mẫu có cạnh đầy đủ, thẻ mang theo, các mẫu GoF chính thức, nhưng không phải lúc nào cũng được sử dụng thường xuyên như các mẫu chúng ta đã khám phá cho đến nay. Nhưng những mẫu này thật tuyệt vời theo cách riêng của chúng, và nếu hoàn cảnh của bạn yêu cầu, bạn nên áp dụng chúng một cách ngang cao đầu. Mục tiêu của chúng tôi trong phần phụ lục này là cung cấp cho bạn ý tưởng cấp cao về những mẫu này đều liên quan đến.

đây là phần phụ lục

mẫu cầu

Cầu

Sử dụng Bridge Pattern để thay đổi không chỉ cách triển khai mà còn cả cách truy xuất.

Một kịch bản

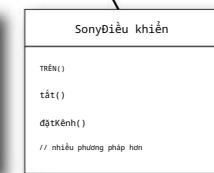
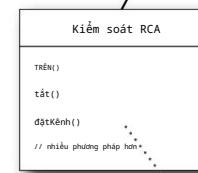
Hãy tưởng tượng bạn sắp cách mạng hóa "việc thư giãn cực độ". Bạn đang viết mã cho một chiếc điều khiển từ xa tiện dụng và thân thiện với người dùng mới dành cho TV. Bạn đã biết rằng bạn phải sử dụng các kỹ thuật OO tốt vì trong khi chiếc điều khiển từ xa dựa trên cùng một khái niệm truy xuất, sẽ có rất nhiều cách triển khai - một cách cho mỗi kiểu TV.

Mỗi điều khiển từ xa đều có cùng sự truy xuất.

Đây là một sự truy xuất. Nó có thể là một giao diện hoặc một lớp truy xuất.



Có nhiều cách triển khai cho một từng loại TV.



Sự tiện thoái lưỡng nan của bạn

Bạn biết rằng giao diện người dùng của điều khiển từ xa sẽ không đúng ngay từ lần đầu tiên. Trên thực tế, bạn mong đợi sản phẩm sẽ được tinh chỉnh nhiều lần khi dữ liệu khả năng sử dụng được thu thập trên điều khiển từ xa.

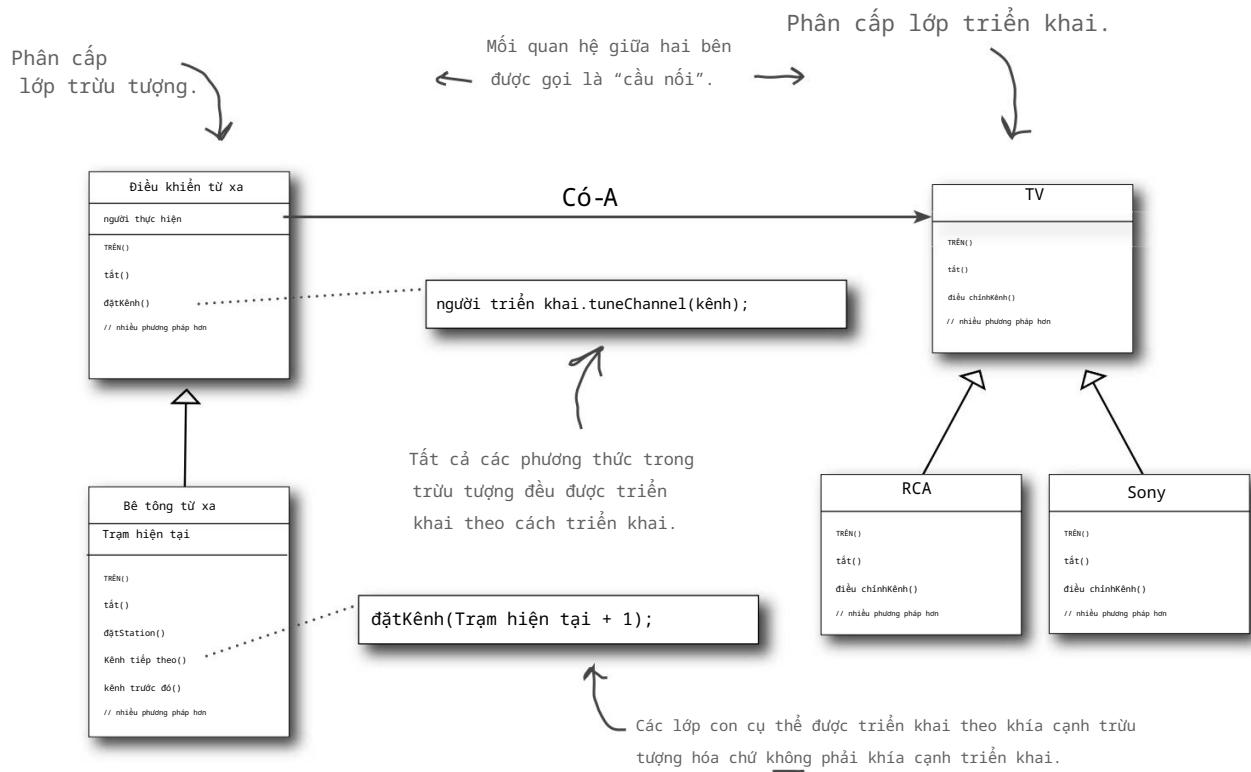
Vì vậy, tình thế tiễn thoái lưỡng nan của bạn là điều khiển từ xa sẽ thay đổi và TV cũng sẽ thay đổi. Bạn đã truy xuất hóa giao diện người dùng để có thể thay đổi việc triển khai trên nhiều TV mà khách hàng của bạn sẽ sở hữu. Nhưng bạn cũng sẽ cần phải thay đổi sự truy xuất hóa vì nó sẽ thay đổi theo thời gian khi điều khiển từ xa được cải thiện dựa trên phản hồi của người dùng.

Vậy làm thế nào để tạo ra một thiết kế hướng đối tượng cho phép bạn thay đổi cách triển khai và truy xuất hóa?

Khi sử dụng thiết kế này, chúng ta chỉ có thể thay đổi cách triển khai TV chứ không phải giao diện người dùng.

Tại sao nên sử dụng mẫu cầu nối?

Mẫu Bridge cho phép bạn thay đổi cách triển khai và trừu tượng hóa bằng cách đặt hai thứ này vào các phân cấp lớp riêng biệt.



Bây giờ bạn có hai hệ thống phân cấp, một cho các điều khiển từ xa và một hệ thống riêng cho các triển khai TV dành riêng cho nền tảng. Cầu nối cho phép bạn thay đổi cả hai bên của hai hệ thống phân cấp một cách độc lập.

Lợi ích của cầu nối

- β Tách rời một triển khai để nó không bị ràng buộc vĩnh viễn vào một giao diện.
- β Sự trừu tượng hóa và triển khai có thể được mở rộng một cách độc lập.
- β Những thay đổi đối với các lớp trừu tượng cụ thể không ảnh hưởng đến máy khách.

Công dụng và nhược điểm của Bridge

- β Hữu ích nhất khi nào bạn cần thay đổi giao diện và cách triển khai theo những cách khác nhau.
- β Tăng độ phức tạp.

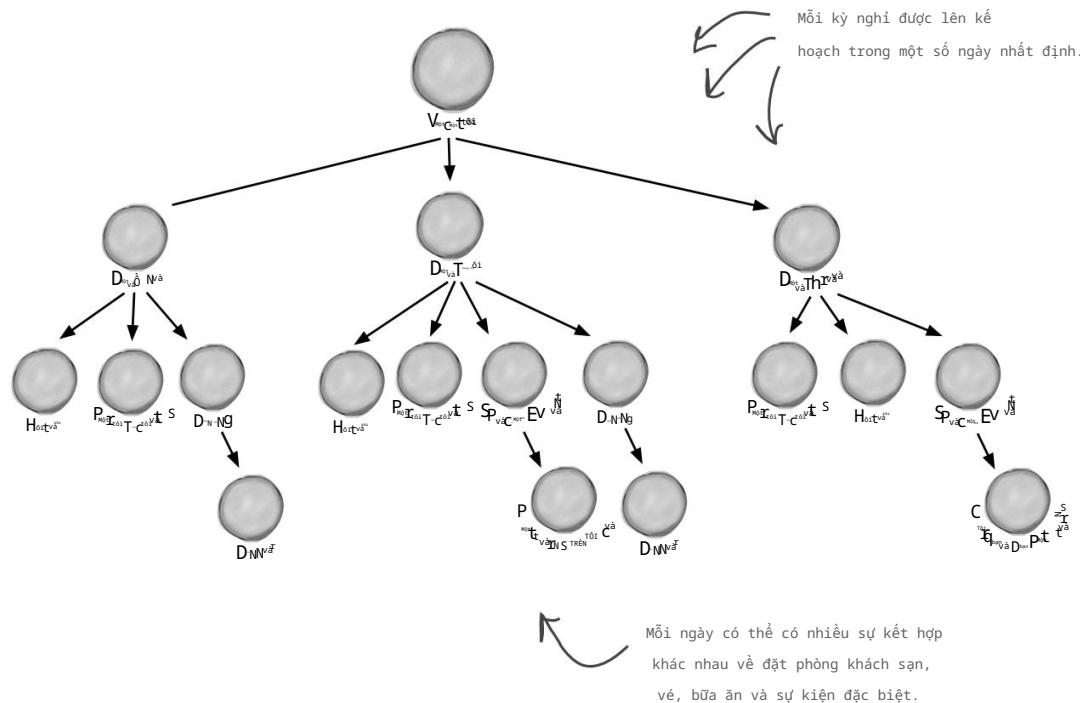
mẫu xây dựng

Người xây dựng

Sử dụng Builder Pattern để đóng gói quá trình xây dựng sản phẩm và cho phép xây dựng theo từng bước.

Một kịch bản

Bạn vừa được yêu cầu xây dựng một kế hoạch kỳ nghỉ cho Patternsland, một công viên giải trí mới nằm ngay bên ngoài Objectville. Du khách đến công viên có thể chọn khách sạn và nhiều loại vé vào cửa, đặt chỗ nhà hàng và thậm chí đặt các sự kiện đặc biệt. Để tạo một kế hoạch kỳ nghỉ, bạn cần có khả năng tạo các cấu trúc như sau:



Bạn cần một thiết kế linh hoạt

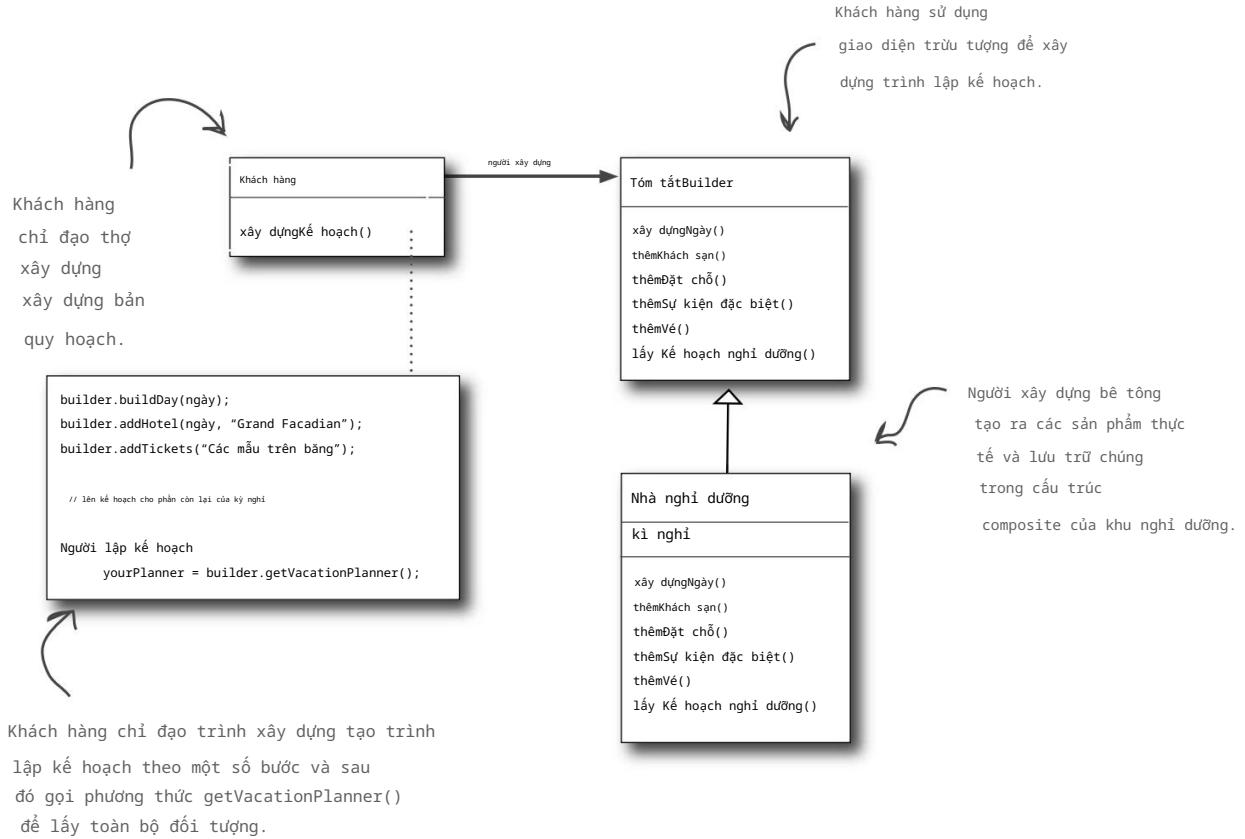
Kế hoạch của mỗi khách có thể khác nhau về số ngày và loại hoạt động.

Ví dụ, một cư dân địa phương có thể không cần khách sạn nhưng muốn đặt chỗ ăn tối và sự kiện đặc biệt. Một vị khách khác có thể bay đến Objectville và cần khách sạn, đặt chỗ ăn tối và vé vào cửa.

Vì vậy, bạn cần một cấu trúc dữ liệu linh hoạt có thể biểu diễn các nhà lập kế hoạch khách mời và tất cả các biến thể của họ; bạn cũng cần tuân theo một chuỗi các bước phức tạp tiềm ẩn để tạo ra nhà lập kế hoạch. Làm thế nào bạn có thể cung cấp một cách để tạo ra cấu trúc phức tạp mà không trộn lẫn nó với các bước để tạo ra nó?

Tại sao nên sử dụng Builder Pattern?

Bạn còn nhớ Iterator không? Chúng tôi đã đóng gói vòng lặp vào một đối tượng riêng biệt và ẩn biểu diễn bên trong của bộ sưu tập khỏi máy khách. Ý tưởng ở đây cũng giống vậy: chúng tôi đóng gói việc tạo ra trình lập kế hoạch chuyến đi trong một đối tượng (gọi là trình xây dựng) và yêu cầu máy khách của chúng tôi yêu cầu trình xây dựng xây dựng cấu trúc trình lập kế hoạch chuyến đi cho nó.



Lợi ích của người xây dựng

- Bao gồm cách xây dựng một đối tượng phức tạp.
- Cho phép xây dựng các đối tượng theo quy trình nhiều bước và thay đổi (khác với quy trình nhà máy một bước).
- Ẩn nội dung bên trong của sản phẩm khỏi máy khách.
- Việc triển khai sản phẩm có thể được hoán đổi vì khách hàng chỉ nhìn thấy một giao diện trừu tượng.

Công dụng và nhược điểm của Builder

- Thường được sử dụng để xây dựng các công trình phức hợp.
- Việc xây dựng các đối tượng đòi hỏi nhiều miền hơn hiểu biết của khách hàng hơn khi sử dụng Nhà máy.

mô hình chuỗi trách nhiệm

Chuỗi trách nhiệm

Sử dụng Mẫu Chuỗi trách nhiệm khi bạn muốn trao cho nhiều đối tượng cơ hội xử lý cùng một yêu cầu.

Một kịch bản

Mighty Gumball đã nhận được nhiều email hơn mức họ có thể xử lý kể từ khi phát hành Máy Gumball chạy bằng Java.

Theo phân tích của riêng họ, họ nhận được bốn loại email:
thư của người hâm mộ từ khách hàng yêu thích trò chơi 1
trong 10 mới, khiếu nại từ các bậc phụ huynh có con
nghiên cứu và yêu cầu đặt máy ở vị trí mới. Họ
cũng nhận được một lượng thư rác khá.

Mọi thư của người hâm mộ cần được chuyển thẳng đến CEO,
mọi khiếu nại đều được chuyển đến bộ phận pháp lý và mọi
yêu cầu về máy móc mới đều được chuyển đến bộ phận
phát triển kinh doanh. Thư rác cần phải được xóa.

Nhiệm vụ của bạn

Mighty Gumball đã viết một số máy dò AI có thể phân biệt
email là thư rác, thư của người hâm mộ, khiếu nại hay
yêu cầu, nhưng họ cần bạn tạo ra một thiết kế có thể sử dụng
máy dò để xử lý email đến.

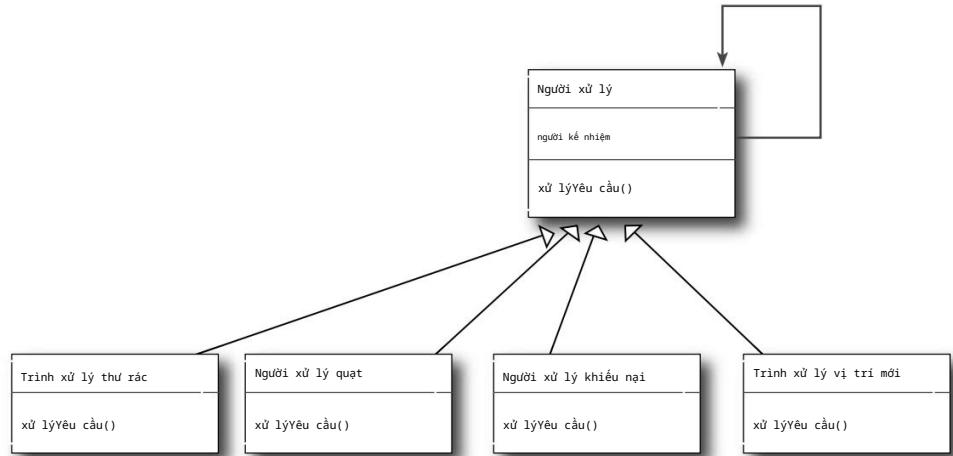
Bạn
phải giúp chúng
tôi xử lý lượng email
không lồ mà chúng tôi nhận
được kể từ khi Java
Gumball Machine ra
mắt.



Cách sử dụng Mô hình Chuỗi trách nhiệm

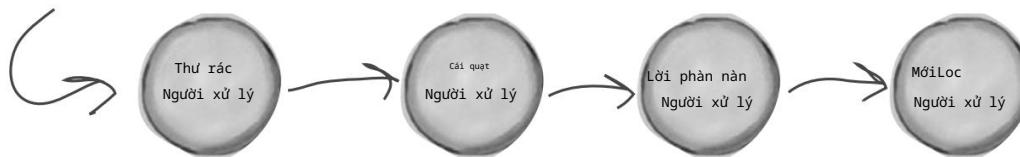
Với Chain of Responsibility Pattern, bạn tạo một chuỗi các đối tượng kiểm tra một yêu cầu. Mỗi đối tượng lần lượt kiểm tra yêu cầu và xử lý nó hoặc chuyển nó cho đối tượng tiếp theo trong chuỗi.

Mỗi đối tượng trong chuỗi hoạt động như một trình xử lý và có một đối tượng kế nhiệm. Nếu nó có thể xử lý yêu cầu, nó sẽ xử lý; nếu không, nó sẽ chuyển tiếp yêu cầu đến người kế nhiệm của nó.



Khi email được nhận, nó sẽ được chuyển đến trình xử lý đầu tiên: SpamHandler. Nếu SpamHandler không thể xử lý yêu cầu, nó sẽ được chuyển đến FanHandler. Và cứ thế...

Mỗi email được chuyển đến người xử lý đầu tiên.



Email sẽ không được xử lý nếu nó rời ra khỏi chuỗi

- mặc dù bạn luôn có thể triển khai trình xử lý tổng hợp.

Chuỗi trách nhiệm Lợi ích

- Þ Tách biệt người gửi yêu cầu và người nhận yêu cầu.
- Þ Đơn giản hóa đối tượng của bạn vì nó không cần phải biết cấu trúc của chuỗi và giữ liên lạc trực tiếp với các thành viên của chuỗi.
- Þ Cho phép bạn thêm hoặc xóa trách nhiệm một cách nhanh chóng bằng cách thay đổi các thành viên hoặc thứ tự của chuỗi.

Chuỗi trách nhiệm sử dụng và hạn chế

- Þ Thường được sử dụng trong hệ thống Windows để xử lý các sự kiện như nhấp chuột và sự kiện bàn phím.
- Þ Việc thực hiện yêu cầu không được đảm bảo; nó có thể thất bại ra khỏi cuối chuỗi nếu không có đối tượng nào xử lý nó (điều này có thể là lợi thế hoặc bất lợi).
- Þ Có thể khó quan sát các đặc điểm thời gian chạy và gỡ lỗi.

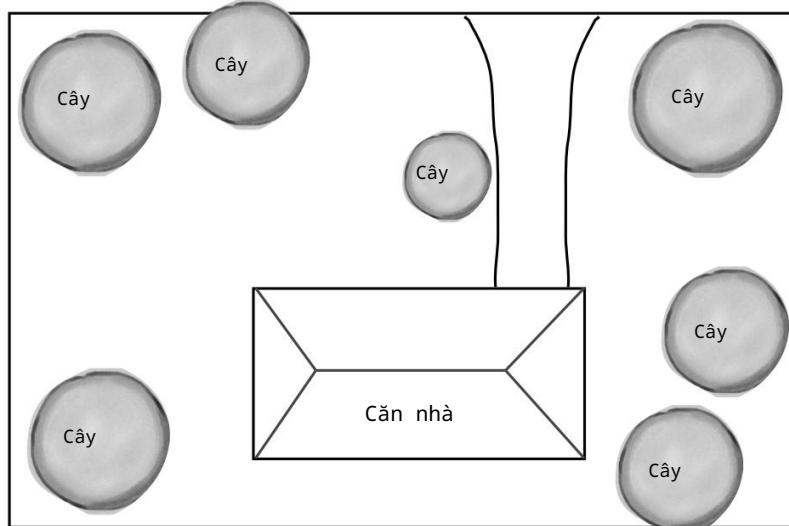
mẫu trọng lượng fl

cân ruồi

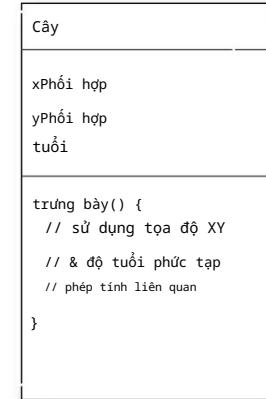
Sử dụng Mẫu Flyweight khi một thể hiện của một lớp có thể được sử dụng để cung cấp nhiều "thể hiện ảo".

Một kịch bản

Bạn muốn thêm cây làm đối tượng trong ứng dụng thiết kế cảnh quan mới hấp dẫn của mình. Trong ứng dụng của bạn, cây không thực sự có tác dụng gì nhiều; chúng có vị trí XY và có thể tự vẽ một cách động, tùy thuộc vào độ tuổi của chúng. Vấn đề là, người dùng có thể muốn có rất nhiều cây trong một trong những thiết kế cảnh quan nhà của họ. Nó có thể trông giống như thế này:



Mỗi phiên bản Tree đều duy trì trạng thái riêng của nó.



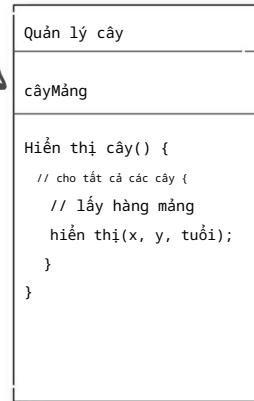
Sự tiến thoái lưỡng nan của khách hàng lớn của bạn

Bạn vừa có được "tài khoản tham khảo" của mình. Khách hàng chính mà bạn đã chào hàng trong nhiều tháng. Họ sẽ mua 1.000 chỗ ngồi trong ứng dụng của bạn và họ đang sử dụng phần mềm của bạn để thiết kế cảnh quan cho các cộng đồng được quy hoạch lớn. Sau khi sử dụng phần mềm của bạn trong một tuần, khách hàng phàn nàn rằng khi họ tạo những lùm cây lớn, ứng dụng bắt đầu chậm dần...

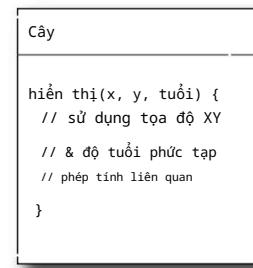
Tại sao nên sử dụng mẫu Flyweight?

Nếu thay vì có hàng ngàn đối tượng Tree, bạn có thể thiết kế lại hệ thống của mình để chỉ có một phiên bản Tree và một đối tượng máy khách duy trì trạng thái của TẤT CẢ các cây của bạn thì sao? Đó chính là Flyweight!

Toàn bộ trạng thái của TẤT
CẢ các đối tượng Tree ào
của bạn đều được lưu
trữ trong mảng 2D này



Một đối tượng Tree duy nhất,
không có trạng thái.



Lợi ích của Flyweight

- ß Giảm số lượng phiên bản đối tượng khi chạy, tiết kiệm bộ nhớ.
- ß Tập trung trạng thái của nhiều đối tượng “ào” vào một vị trí duy nhất.

Công dụng và nhược điểm của Flyweight

- ß Flyweight được sử dụng khi một lớp có nhiều trường hợp và tất cả đều có thể được kiểm soát giống hệt nhau.
- ß Một nhược điểm của mẫu Flyweight là một khi nếu bạn đã triển khai nó, các thẻ hiện logic đơn lẻ của lớp sẽ không thể hoạt động độc lập với các thẻ hiện khác.

mẫu thông dịch viên

Người phiên dịch

Sử dụng Mẫu thông dịch để xây dựng trình thông dịch cho một ngôn ngữ.

Một kịch bản

Bạn còn nhớ Duck Pond Simulator không? Bạn có linh cảm rằng nó cũng sẽ là một công cụ giáo dục tuyệt vời cho trẻ em học lập trình. Sử dụng trình mô phỏng, mỗi trẻ em sẽ được điều khiển một con vịt bằng một ngôn ngữ đơn giản. Sau đây là một ví dụ về ngôn ngữ:

Phải;
trong khi (ban ngày) bay;
lang băm;

Quay con vịt sang phải.
Bay cà ngày...
...và sau đó kêu quack quack.



Bây giờ, nhớ lại cách tạo ngữ pháp từ một trong những lớp lập trình cơ bản cũ của bạn, bạn viết ra ngữ pháp:

```
biểu thức ::= <lệnh> | <chuỗi> | <lặp lại>
chuỗi ::= <biểu thức> ';' <biểu thức>
lệnh ::= phải | quack | bay
sự lặp lại ::= trong khi '(' <biểu thức> ')' <biểu thức>
biểu thức ::= [AZ,az]+
```

Chương trình là một biểu thức bao gồm các chuỗi lệnh và lệnh lặp lại (câu lệnh "while").

Chuỗi là một tập hợp các biểu thức được phân tách bằng dấu chấm phẩy.

Chúng ta có ba lệnh: phải, quack và bay.

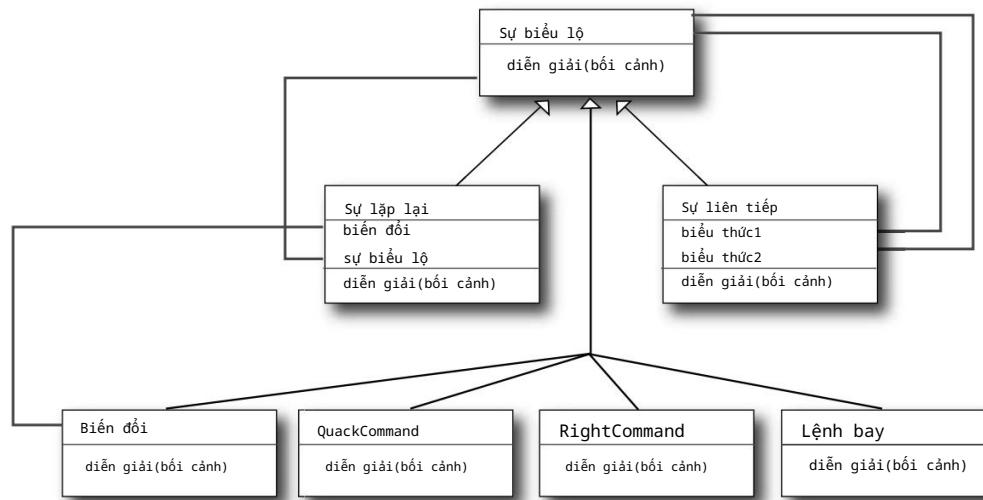
Một câu lệnh while chỉ là một biến điều kiện và một biểu thức.

Bây giờ thì sao?

Bạn đã có ngữ pháp; bây giờ tất cả những gì bạn cần là một cách để biểu diễn và diễn giải các câu trong ngữ pháp để học sinh có thể thấy được tác động của chương trình lên những chú vịt mô phỏng.

Làm thế nào để triển khai một trình thông dịch

Khi bạn cần triển khai một ngôn ngữ đơn giản, Mẫu thông dịch sẽ định nghĩa một biểu diễn dựa trên lớp cho ngữ pháp của ngôn ngữ đó cùng với một trình thông dịch để diễn giải các câu. Để biểu diễn ngôn ngữ, bạn sử dụng một lớp để biểu diễn từng quy tắc trong ngôn ngữ. Đây là ngôn ngữ vịt được dịch thành các lớp. Lưu ý ánh xạ trực tiếp đến ngữ pháp.



Để diễn giải ngôn ngữ, hãy gọi phương thức `interpret()` trên mỗi kiểu biểu thức. Phương thức này được truyền một ngữ cảnh - chứa luồng đầu vào của chương trình mà chúng ta đang phân tích cú pháp - và khớp với đầu vào và đánh giá nó.

Quyền lợi của phiên dịch viên

- Việc thể hiện từng quy tắc ngữ pháp trong một lớp giúp ngôn ngữ dễ triển khai hơn.
- Vì ngữ pháp được biểu diễn bằng các lớp nên bạn có thể dễ dàng thay đổi hoặc mở rộng ngôn ngữ.
- Bằng cách thêm các phương thức bổ sung vào cấu trúc lớp, bạn có thể thêm các hành vi mới ngoài việc diễn giải, như in đẹp và xác thực chương trình phức tạp hơn.

Sử dụng và hạn chế của trình thông dịch

- Sử dụng trình thông dịch khi bạn cần triển khai một ngôn ngữ đơn giản.
- Phù hợp khi bạn có ngữ pháp đơn giản và sự đơn giản quan trọng hơn hiệu quả.
- Được sử dụng cho ngôn ngữ lập trình và viết kịch bản.
- Mẫu này có thể trở nên cồng kềnh khi số lượng quy tắc ngữ pháp lớn. Trong những trường hợp này, trình tạo trình phân tích cú pháp/trình biên dịch có thể phù hợp hơn.

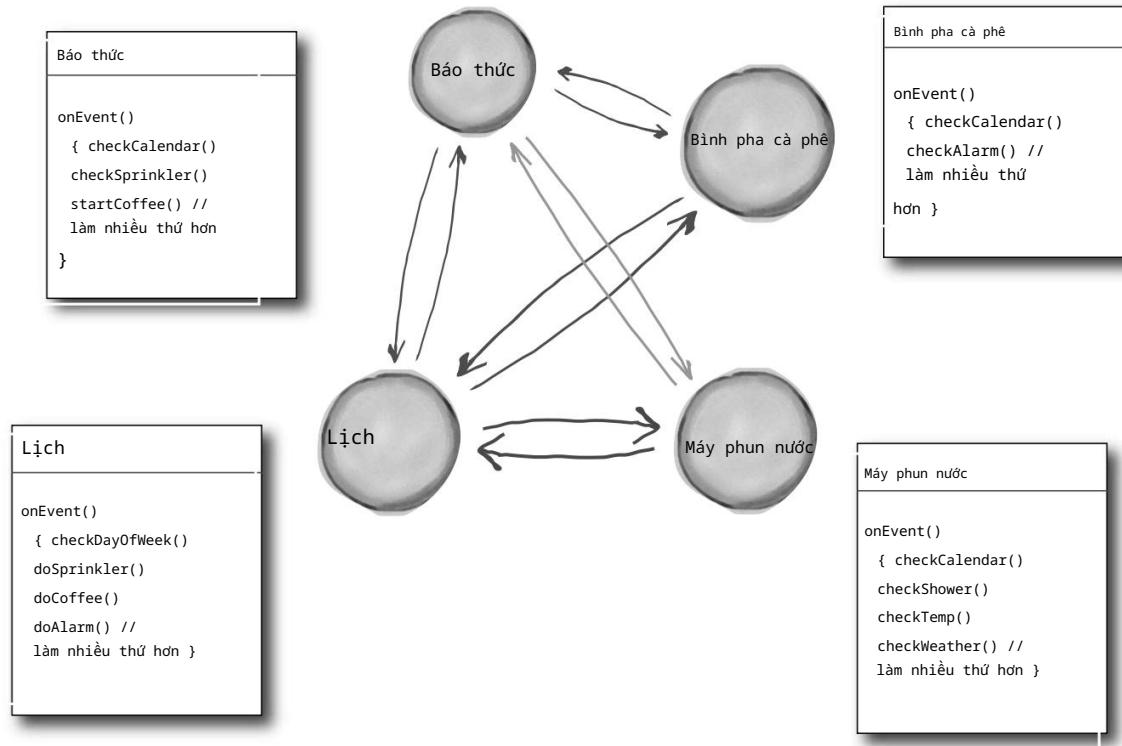
mẫu trung gian

Người trung gian

Người trung gian Sử dụng Mô hình người trung gian để tập trung các giao tiếp phức tạp và kiểm soát giữa các đối tượng liên quan.

Kịch bản Bob

có một ngôi nhà tự động hỗ trợ Java, nhờ những người tốt bụng tại HouseOfTheFuture. Tất cả các thiết bị của anh ấy đều được thiết kế để giúp cuộc sống của anh ấy dễ dàng hơn. Khi Bob ngừng nhấn nút báo lại, đồng hồ báo thức của anh ấy sẽ báo cho máy pha cà phê bắt đầu pha. Mặc dù cuộc sống của Bob rất tốt, anh ấy và những khách hàng khác luôn yêu cầu nhiều tính năng mới: Không uống cà phê vào cuối tuần... Tất vòi phun nước 15 phút trước khi tắm theo lịch... Đặt báo thức sớm vào những ngày đồ rác...



Thé tiền thoái lưỡng nan của HouseOfTheFuture

Thật sự rất khó để theo dõi quy tắc nào nằm trong đối tượng nào và các đối tượng khác nhau nên liên quan đến nhau như thế nào.

mẫu còn sót lại

Người hòa giải đang hành động...

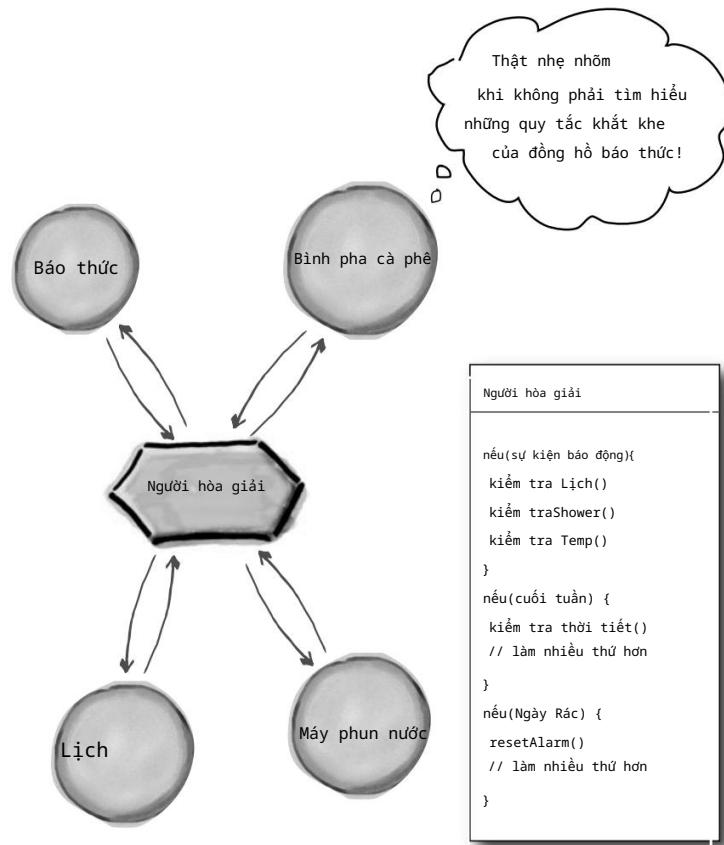
Khi thêm Mediator vào hệ thống, tất cả các đối tượng của thiết bị có thể được đơn giản hóa đáng kể:

Họ sẽ báo cho Người hòa giải khi trạng thái của họ thay đổi.

↪ Họ trả lời các yêu cầu từ Người hòa giải.

Trước khi thêm Mediator, tất cả các đối tượng thiết bị cần biết về nhau... tất cả chúng đều được liên kết chặt chẽ. Với Mediator tại chỗ, tất cả các đối tượng thiết bị đều được tách biệt hoàn toàn khỏi nhau.

Mediator chứa tất cả logic điều khiển cho toàn bộ hệ thống. Khi một thiết bị hiện có cần một quy tắc mới hoặc một thiết bị mới được thêm vào hệ thống, bạn sẽ biết rằng tất cả logic cần thiết sẽ được thêm vào Mediator.



Quyền lợi của người hòa giải

- ↪ Tăng khả năng tái sử dụng các đối tượng được Mediator hỗ trợ bằng cách tách chúng khỏi hệ thống.
- ↪ Đơn giản hóa việc bảo trì hệ thống bằng cách tập trung logic điều khiển.
- ↪ Đơn giản hóa và giảm bớt sự đa dạng của các tin nhắn được gửi giữa các đối tượng trong hệ thống.

Sử dụng và hạn chế của Mediator

- ↪ Mediator thường được sử dụng để phối hợp các thành phần GUI có liên quan.
- ↪ Một nhược điểm của mô hình Mediator là không có thiết kế phù hợp, bản thân đối tượng Mediator có thể trở nên quá phức tạp.

mẫu lưu niệm

Kỹ vật

Sử dụng Memento Pattern khi bạn cần có khả năng trả một đối tượng về một trong các trạng thái trước đó của nó; ví dụ, nếu người dùng của bạn yêu cầu "hoàn tác".

Một kịch bản

Trò chơi nhập vai tương tác của bạn cực kỳ thành công và đã tạo ra một đội quân nghiện ngập, tất cả đều cố gắng đạt đến "cấp độ 13" trong truyền thuyết. Khi người dùng tiến triển đến các cấp độ trò chơi đầy thử thách hơn, khả năng gấp phải tình huống kết thúc trò chơi sẽ tăng lên. Những người hâm mộ đã dành nhiều ngày để tiến triển đến cấp độ nâng cao sẽ dễ hiểu là bức bối khi nhân vật của họ bị giết và họ phải bắt đầu lại từ đầu. Tiếng kêu gọi lệnh "lưu tiến trình" vang lên, để người chơi có thể lưu tiến trình trò chơi của mình và ít nhất là khôi phục lại hầu hết những nỗ lực của họ khi nhân vật của họ bị tiêu diệt một cách bất công. Chức năng "lưu tiến trình" cần được thiết kế để đưa người chơi được hồi sinh trở lại cấp độ cuối cùng mà cô ấy đã hoàn thành thành công.

Chỉ cần cẩn thận khi bạn lưu trạng thái trò chơi. Nó khá phức tạp và tôi không muốn bất kỳ ai khác có quyền truy cập vào nó làm hỏng và phá vỡ mã của tôi.



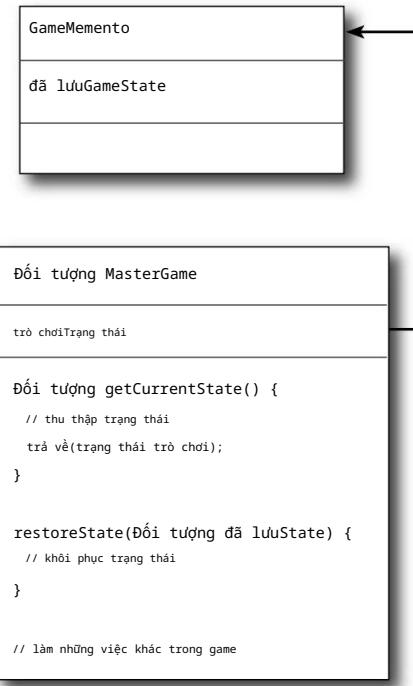
Memento đang hoạt động

Memento có hai mục tiêu:

β Lưu trạng thái quan trọng của đối tượng chính trong hệ thống.

β Duy trì việc đóng gói đối tượng chính.

Ghi nhớ nguyên tắc trách nhiệm duy nhất, bạn cũng nên giữ trạng thái mà bạn đang lưu tách biệt với đối tượng khóa. Đối tượng riêng biệt này giữ trạng thái được gọi là đối tượng Memento.



Mặc dù đây không phải là một triển khai quá thú vị, nhưng hãy lưu ý rằng Client không có quyền truy cập vào dữ liệu của Memento.



Lợi ích của Memento

- β Giữ trạng thái đã lưu bên ngoài đối tượng khóa giúp duy trì sự gắn kết.
- β Giữ cho dữ liệu của đối tượng chính được đóng gói.
- β Cung cấp khả năng phục hồi dễ triển khai.

Sử dụng và Nhược điểm của Memento

- β Memento được sử dụng để lưu trạng thái.
- β Một nhược điểm khi sử dụng Memento là việc lưu và khôi phục trạng thái có thể tốn nhiều thời gian.
- β Trong các hệ thống Java, hãy cân nhắc sử dụng Serialization để lưu trạng thái của hệ thống.

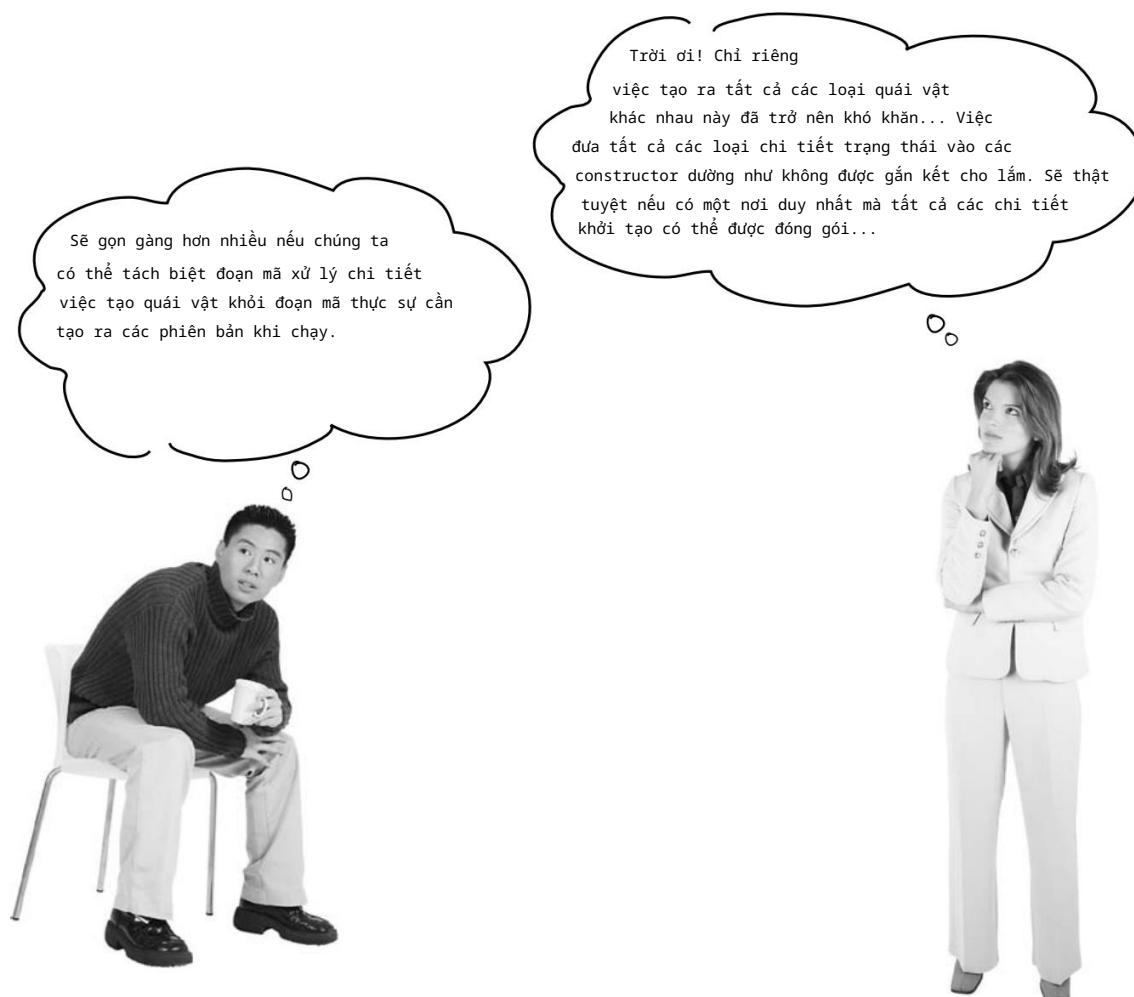
mẫu nguyên mẫu

Nguyên mẫu

Sử dụng Mẫu nguyên mẫu khi việc tạo một thẻ hiện của một lớp nhất định tốn kém hoặc phức tạp.

Một kịch bản

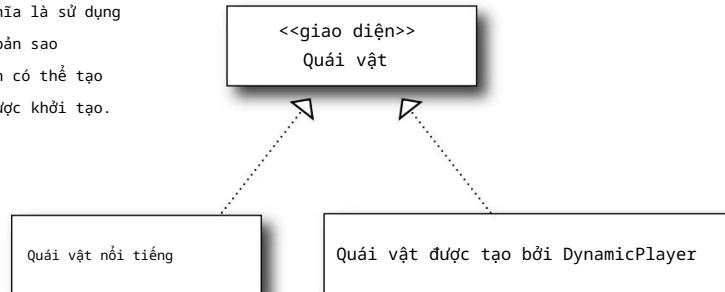
Trò chơi nhập vai tương tác của bạn có một sự thèm khát vô độ đối với quái vật. Khi các anh hùng của bạn thực hiện hành trình của họ qua một cảnh quan được tạo ra một cách năng động, họ sẽ chạm trán với một chuỗi kẻ thù vô tận mà bạn phải khuất phục. Bạn muốn các đặc điểm của quái vật phát triển theo cảnh quan thay đổi. Sẽ không hợp lý khi những con quái vật giống chim đi theo các nhân vật của bạn vào thế giới dưới nước. Cuối cùng, bạn muốn cho phép những người chơi nâng cao tạo ra quái vật tùy chỉnh của riêng họ.



mẫu còn sót lại

Nguyên mẫu để giải cứu

Mẫu Prototype cho phép bạn tạo các phiên bản mới bằng cách sao chép các phiên bản hiện có. (Trong Java, điều này thường có nghĩa là sử dụng phương thức `clone()` hoặc hủy tuẫn tự hóa khi bạn cần các bản sao sâu.) Một khía cạnh quan trọng của mẫu này là mã máy khách có thể tạo các phiên bản mới mà không cần biết lớp cụ thể nào đang được khởi tạo.



Người tạo ra quái vật

```

làmRandomMonster() {
    Quái vật m =
        MonsterRegistry.getMonster();
}
  
```

Khách hàng cần một con quái vật mới phù hợp với tình hình hiện tại. (Khách hàng sẽ không biết mình sẽ nhận được loại quái vật nào.)

Đăng ký quái vật

```

Quái vật getMonster() {
    // tìm đúng quái vật
    trả về correctMonster.clone();
}
  
```

Sở đăng ký sẽ tìm ra quái vật phù hợp, tạo bản sao của quái vật đó và trả về bản sao đó.

Lợi ích của nguyên mẫu

- β Ánh đi sự phức tạp khi tạo phiên bản mới từ máy khách.
- β Cung cấp tùy chọn cho khách hàng để tạo các đối tượng có loại chưa biết.
- β Trong một số trường hợp, sao chép một đối tượng có thể hiệu quả hơn là tạo một đối tượng mới.

Sử dụng và nhược điểm của nguyên mẫu

- β Nguyên mẫu nên được xem xét khi hệ thống phải tạo các đối tượng mới thuộc nhiều loại trong một hệ thống phân cấp lớp phức tạp.
- β Một nhược điểm khi sử dụng Nguyên mẫu là việc tạo bản sao của một đối tượng đôi khi có thể phức tạp.

mẫu khách truy cập

Khách thăm

Sử dụng Visitor Pattern khi bạn muốn thêm khả năng vào một hợp chất các đối tượng và việc đóng gói không quan trọng.

Một kịch bản

Những khách hàng thường xuyên lui tới Objectville Diner và Objectville Pancake House gần đây đã có ý thức hơn về sức khỏe. Họ yêu cầu thông tin dinh dưỡng trước khi gọi món. Vì cả hai cơ sở đều sẵn lòng tạo đơn đặt hàng đặc biệt nên một số khách hàng thậm chí còn yêu cầu thông tin dinh dưỡng cho từng thành phần.

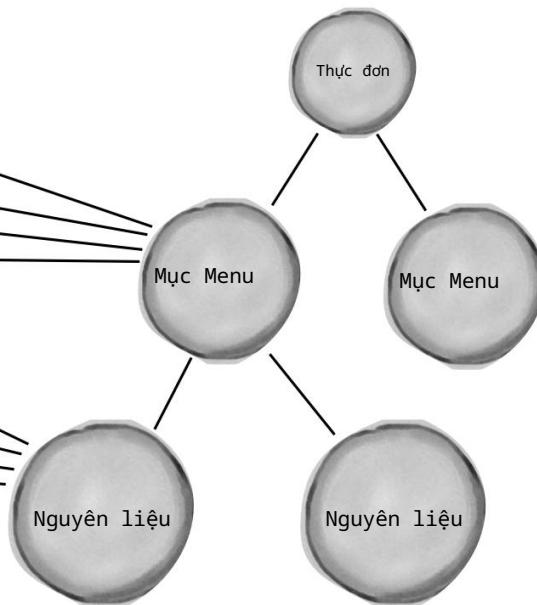
Giải pháp được Lou đề xuất:

// phương pháp mới

Nhận Xếp Hạng Sức Khỏe
lấy Calo
lấy Protein
lấy Carbs

// phương pháp mới

Nhận Xếp Hạng Sức KhỎe
lấy Calo
lấy Protein
lấy Carbs



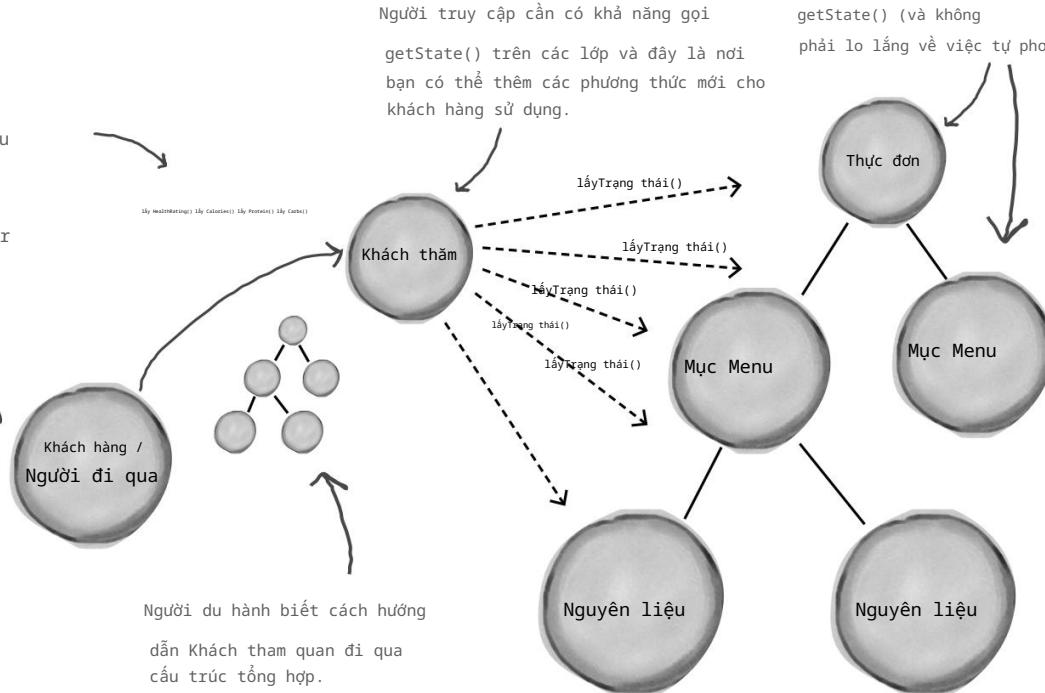
Mỗi quan tâm của Mel...

"Cậu bé, có vẻ như chúng ta đang mở hộp Pandora. Ai biết chúng ta sẽ phải thêm phương pháp mới nào tiếp theo, và mỗi lần chúng ta thêm một phương pháp mới, chúng ta phải thực hiện ở hai nơi. Thêm vào đó, nếu chúng ta muốn nâng cao ứng dụng cơ sở bằng, chẳng hạn, một lớp công thức nấu ăn thì sao? Sau đó, chúng ta sẽ phải thực hiện những thay đổi này ở ba địa điểm khác nhau..."

Người khách ghé thăm

Visitor phải truy cập từng phần tử của Composite; chức năng đó nằm trong đối tượng Traverser. Visitor được Traverser hướng dẫn để thu thập trạng thái từ tất cả các đối tượng trong Composite. Sau khi trạng thái đã được thu thập, Client có thể yêu cầu Visitor thực hiện nhiều thao tác khác nhau trên trạng thái. Khi cần chức năng mới, chỉ cần nâng cấp Visitor.

Khách hàng yêu cầu Người truy cập lấy thông tin từ cấu trúc Composite... Có thể thêm phương thức mới vào Visitor mà không ảnh hưởng đến Composite.



Quyền lợi của khách tham quan

- Þ Cho phép bạn thêm các thao tác vào cấu trúc Composite mà không cần thay đổi cấu trúc đó.
- Þ Việc thêm hoạt động mới tương đối dễ dàng.
- Þ Mã cho các hoạt động do Khách truy cập thực hiện được tập trung hóa.

Nhược điểm của khách truy cập

- Þ Việc đóng gói các lớp Composite bị hỏng khi sử dụng Visitor.
- Þ Vì có liên quan đến chức năng duyệt nên việc thay đổi cấu trúc Composite trở nên khó khăn hơn.

g h Chỉ số g

MỘT

Mẫu Nhà máy Trùu tượng 156. Xem thêm Mẫu Nhà máy

mẫu quy trình kinh doanh 605

Mẫu bộ chuyển đổi

Trình xem bìa CD 463

Ưu điểm 242

Mẫu Chuỗi Trách Nhiệm 616-617

bộ điều hợp lớp 244

thay đổi 339

sơ đồ lớp 243

định nghĩa 14

kết hợp các mẫu 504

hàng số trong phát triển phần mềm 8 xác

định nghĩa 243

định 53

nam châm vịt 245

Công ty Choc-O-Holic 175

Bộ điều hợp lặp lại liệt kê 248

lớp bùng nổ 81

Bài tập 251

mã nam châm 69, 179, 245, 350

giải thích 241

sự gắn kết 339-340

trò chuyện bên lò sưởi 247, 252-253

Kết hợp các mẫu 500

giới thiệu 237

Mẫu nhà máy trùu tượng 508

bộ điều hợp đối tượng 244

Biểu đồ lớp Adapter

Alexander, Christopher 602

Pattern 504 524

tiêu diệt cái ác 606

Mẫu tổng hợp 513

Mẫu chống 606-607

Mẫu trang trí 506

Búa Vàng 607

Mẫu quan sát 516

mẫu ứng dụng 604

Mẫu lệnh

mẫu kiến trúc 604

sơ đồ lớp 207 đối

Mẫu cầu 612-613

tượng lệnh 203

Mẫu xây dựng 614-615

định nghĩa 206-207

các điểm chính 32, 74, 105, 162, 186, 230, 270, 311, 380,

giới thiệu 196

423, 491, 560, 608

đang tài Invoker 201

B

đây là chỉ số

631

Tổng quan đặc

Mẫu lệnh, tiếp tục	trò chuyện bên lò sưởi 252-253
yêu cầu ghi nhật ký 229	phòng vấn 104
lệnh macro 224	giới thiệu 88
Đối tượng Null 214	trong Java I/O 100-101
yêu cầu xếp hàng 228	mẫu cấu trúc 591
hoàn tác 216, 220, 227	Nguyên lý đảo ngược phụ thuộc 139-143
Mẫu tổng hợp	và Nguyên tắc Hollywood 298
và Mẫu Iterator 368	Mẫu thiết kế
sơ đồ lớp 358	Mẫu nhà máy trừu tượng 156
kết hợp các mẫu 513 hành	Mẫu bộ chuyển đổi 243
vi tổng hợp 363	lợi ích 599
hành vi mặc định 360	Mẫu cầu 612-613
định nghĩa 356	Mẫu xây dựng 614-615
phóng vấn 376-377	các loại 589, 592-593
an toàn 367	Mẫu Chuỗi Trách Nhiệm 616-617
an toàn so với minh bạch 515 minh	mẫu lớp 591
bạch 367, 375	Mẫu lệnh 206
thành phần 23, 85, 93, 247, 309 mẫu	Mẫu tổng hợp 356
hợp chất 500, 522	Mẫu trang trí 91
kiểm soát truy cập 460. Xem thêm Proxy Pattern tạo	định nghĩa 579, 581
đối tượng 134	khám phá 586-587 của riêng bạn
ô chữ 33, 76, 163, 187, 231, 271, 310, 378,	Mẫu mặt tiền 264
490	Mẫu phuơng pháp nhà máy 134
cuộc trò chuyện trong ô 55, 93, 195, 208, 387, 397,	Mẫu Flyweight 618-619
433, 583-584	Mẫu thông dịch viên 620-621
M	Mẫu lắp lại 336
Mẫu trang trí	Mẫu trung gian 622-623
và sơ đồ lớp Proxy Pattern 472-	Mẫu Memento 624-625
473 91	Đối tượng Null 214
kết hợp các mẫu 506	mẫu vật thể 591
cuộc trò chuyện trong phòng 93	Mẫu quan sát 51
định nghĩa 91	tổ chức 589
nhược điểm 101, 104	Mẫu nguyên mẫu 626-627
	Mẫu Proxy 460

Chi số 632

Nhà máy đơn giản 114	định nghĩa 264
Mẫu đơn 177	giới thiệu 258
Mẫu trạng thái 410	Mẫu Factory Method 134. Xem thêm Mẫu Factory
Mẫu chiến lược 24	Mẫu nhà máy
Mẫu Phương pháp Mẫu 289	Nhà máy trừu tượng
sử dụng 29	và Phương pháp Nhà máy 158-159, 160-161 sơ
so với các khuôn khổ 29	đồ lớp 156-157
so với thư viện 29	kết hợp các mẫu 508
Mẫu khách truy cập 628-629	định nghĩa 156
Nguyên tắc thiết kế. Xem Nguyên tắc thiết kế hướng đối tượng	phỏng vấn 158-159
Câu đố thiết kế 25, 133, 279, 395, 468, 542	giới thiệu 153
Hộp công cụ thiết kế 32, 74, 105, 162, 186, 230, 270, 311, 380, 423, 491, 560, 608	Phương pháp nhà máy
DJ Xem 534	lợi thế 135
mẫu miền cụ thể 604	và sơ đồ lớp Abstract Factory
E	160-161 134
Elvis 526	định nghĩa 134
bao gồm những gì thay đổi 8-9, 75, 136, 397, 612	phỏng vấn 158-159
thuật toán đóng gói 286, 289 hành vi	giới thiệu 120, 131-132
đóng gói 11	cận cảnh 125
đóng gói lặp lại 323 đóng gói	Nhà máy đơn giản
gọi phương thức 206	định nghĩa 117
đóng gói xây dựng đối tượng 614-615 đóng gói	giới thiệu 114
tạo đối tượng 114, 136	họ các thuật toán. Xem Mẫu chiến lược
đóng gói yêu cầu 206 đóng gói	họ sản phẩm 145 ưu tiên
trạng thái 399	thành phần hơn thừa kế 23, 75
F	trò chuyện bên lò sưởi 62, 247, 252, 308, 418, 472-473
Mẫu mặt tiền	Phim truyền hình dài năm phút 48, 478
lợi thế 260 và	Mẫu Flyweight 618-619
Nguyên tắc kiến thức tối thiểu 269	lực lượng 582
sơ đồ lớp 264	Friedman, Đan Mạch 171
G	
	Gamma, Erich 601

Mã lực

Băng đảng bốn người 583, 601

Gamma, Erich 601

Helm, Richard 601

Johnson, Ralph 601

Vlissides, John 601

điểm truy cập toàn cầu

177 gobble gobble 239

Búa Vàng 607

hướng dẫn sóng tốt hơn với Design Patterns 578

Máy theo dõi Gumball 431

H

CÓ-A 23

Nguyên tắc học tập Head First xxx

Helm, Richard 601

Nhóm đôi 603

Nguyên tắc Hollywood, 296

và Nguyên lý đảo ngược phụ thuộc 298

Home Automation or Bust, Inc. 192

Rạp hát tại nhà ngọt ngào 255

Nóng hay không 475

TÔI

đi săn

Nhược điểm 5

để tái sử dụng 5-6

so với thành phần 93

giao diện 12

Mẫu thông dịch viên 620-621

đảo ngược 141-142

IS-A 23

Mẫu lắp lại

lợi thế 330

Chi số 634

và bộ sưu tập 347-349

và Mẫu tổng hợp 368

và Liệt kê 338

và Hashtable 343, 348

sơ đồ lớp 337

nam châm mã 350

định nghĩa 336

bài tập 327

trình lập bên ngoài 338

cho/trong 349

bộ lắp nội bộ 338

giới thiệu 325

java.util.Iterator 332

Null Iterator 372

lặp lại đa hình 338

xóa đối tượng 332

J

Johnson, Ralph 601

K

HÔN 594

L

Luật Demeter. Xem Nguyên lý ít kiến thức nhất khởi

tạo lười biếng 177

khớp nối lồng lèo 53

TÔI

vị trí đạn ma thuật 594

thạc sĩ và sinh viên 23, 30, 85, 136, 592, 596

Ghép đôi ở Objectville 475

- Mẫu trung gian 622-623
Mẫu Memento 624-625
trung gian 237
Công ty Mighty Gumball, Inc. 386
Mô hình-Xem-Bộ điều khiển
Bộ chuyển đổi mẫu 546
và mẫu thiết kế 532
và Web 549
Mẫu tổng hợp 532, 559
giới thiệu 529
Mẫu trung gian 559
Mẫu quan sát 532
mã nướng săn 564-576
bài hát 526
Mẫu chiến lược 532, 545
cận cảnh 530
Mô hình 2 549. Xem thêm Model-View-Controller
và các mẫu thiết kế 557-558
MVC. Xem Model-View-Controller
- Nguyên lý kiến thức tối thiểu 265
chương trình cho một giao diện, không phải là một triển khai 11, 243, 335
phần đầu cho các thiết kế kết hợp lồng lèo giữa các đối tượng tương tác 53
Có thể quan sát được 64, 71
Mẫu quan sát
sơ đồ lớp 52 mã
nam châm 69
kết hợp các mẫu 516
cuộc trò chuyện trong phòng 55
định nghĩa 51-52
trò chuyện bên lò sưởi 62
Kịch năm phút 48
giới thiệu 44
trong Swing 72-73
Hỗ trợ Java 64
kéo 63
dẩy 63
mối quan hệ môt-nhiều 51-52
OOPSLA 603
Nguyên tắc đóng mở 86-87
bánh quy oreo 526
mô hình tổ chức 605

N

Đối tượng Null 214, 372

Ò

Quán ăn Objectville 26, 197, 316, 628
Nhà hàng bánh kếp Objectville 316, 628
Nguyên tắc thiết kế hướng đối tượng 9, 30-31
Nguyên tắc đảo ngược phụ thuộc 139-143 bao gồm những gì thay đổi 9, 111
ứng hộ thành phần hơn là thừa kế 23, 243, 397
Nguyên tắc Hollywood 296
một lớp, một trách nhiệm 185, 336, 339, 367
Nguyên tắc đóng mở 86-87, 407

hệ thống phân cấp một phần-toàn bộ 356. Xem thêm mẫu mã Composite Pattern danh mục 581, 583, 585
Các mẫu được phơi bày 104, 158, 174, 377-378
các mẫu trong tự nhiên 299, 488-489
mẫu sở thú 604
Mẫu giải thưởng danh dự 117, 214
Cửa hàng pizza 112
Kho lưu trữ mẫu Portland 603

bạn đang ở đây 4 635

QY

Nguyên tắc ít kiến thức nhất 265-268

nhược điểm 267

chương trình để thực hiện 12, 17, 71

chương trình tới một giao diện 12

chương trình tới một giao diện, không phải là một triển khai 11, 75

Mẫu nguyên mẫu 626-627

Mẫu Proxy

và Bộ chuyển đổi mẫu 471

và Mẫu trang trí 471, 472-473

Sơ đồ lớp Caching

Proxy 471 461

định nghĩa 460

Proxy động 474, 479, 486

và RMI 486

bài tập 482

trò chuyện bên lò sưởi 472-473

java.lang.reflect.Proxy 474

Proxy bảo vệ 474, 477

Vườn thú Proxy 488-489

mã sẵn sàng nướng 494

Proxy từ xa 434

biến thẻ 471

Proxy ảo 462

proxy hình ảnh 464

nhà xuất bản/người đăng ký 45

Hỏi

Chất lượng, The. Xem Chất lượng không có tên

Chất lượng không có tên. Xem Chất lượng,

tái cấu trúc 354, 595

điều khiển từ xa 193, 209

Chỉ số 636

Gọi phương thức từ xa. Xem RMI

proxy từ xa 434. Xem thêm Mẫu Proxy

tài sử dụng 13, 23, 85

RMI 436

S

từ vựng chung 26-28, 599-600

gọt bút chì của bạn 5, 42, 54, 61, 94, 97, 99, 124, 137, 148, 176, 183, 205, 225, 242, 268, 284, 322, 342, 396, 400, 406, 409, 421, 483, 511, 518, 520, 589

Nhà máy đơn giản 117

SimUDuck 2, 500

Mẫu Singleton

lợi thế 170, 184 và

thu gom rác thải 184

và các biến toàn cục 185

và đa luồng 180-182

sơ đồ lớp 177

định nghĩa 177

nhược điểm 184 khóa

kiểm tra kép 182

phóng vấn 174

cận cảnh 173

Nguyên tắc trách nhiệm duy nhất 339. Xem thêm Đối tượng
Nguyên tắc thiết kế định hướng: một lớp, một trách
nhiệm

bộ xương 440

Cà phê Starbuzz 80, 276

máy trạng thái 388-389

Mẫu trạng thái

và Mẫu Chiến lược 411, 418-419

sơ đồ lớp 410

định nghĩa 410

nhược điểm 412, 417
giới thiệu 398

chia sẻ trạng thái 412
nhà máy tĩnh 115
Mẫu chiến lược 24
và Mẫu Tiêu bang 411, 418-419
và Mẫu Phương pháp Mẫu 308-309
đóng gói hành vi 22 họ thuật
toán 22
trò chuyện bên lò suối 308
gốc 440

T

Mẫu Phương pháp Mẫu
Ưu điểm 288 và
Applet 307
và java.util.Arrays 300 và
Strategy Pattern 305, 308-309
và Swing 306 và
Nguyên tắc Hollywood 297
sơ đồ lớp 289
định nghĩa 289
trò chuyện bên lò suối 308-309
móc 292, 295
giới thiệu 286

cận cảnh 290-291

Little Lisper 171 suy
nghĩ theo khuôn mẫu 594-595
liên kết chặt chẽ 53

Bạn

hoàn tác 216, 227
mẫu thiết kế giao diện người dùng 605

V

thay đổi. Xem đóng gói những gì thay đổi
Mẫu khách truy cập 628-629
Vlissides, John 601

T

Thời tiết-O-Rama 38
khi nào không nên sử dụng các mẫu 596-
598 Ai làm gì? 202, 254, 298, 379, 422, 487, 588
Tại sao lại là một
con vịt? 500 vật thể bao bọc 88, 242, 252, 260, 473, 508. Xem
cũng như Mẫu Adapter, Mẫu Decorator, Mặt tiền
Mẫu, Mẫu Proxy

Có

tâm trí của bạn về các mẫu 597

bạn đang ở đây 4 637

h g Bản quyền g



Tất cả các bộ cục nội thất đều được thiết kế bởi Eric Freeman, Elisabeth Freeman, Kathy Sierra và Bert Bates. Kathy và Bert đã tạo ra diện mạo và cảm nhận của series Head First.

Cuốn sách được sản xuất bằng Adobe InDesign CS (một công cụ thiết kế cực kỳ tuyệt vời mà chúng ta không thể có được đủ) và Adobe Photoshop CS. Cuốn sách được sắp chữ bằng cách sử dụng Uncle Stinky, Mister Frisky (Bạn nghĩ chúng tôi đùa thôi), Ann Satellite, Baskerville, Comic Sans, Myriad Pro, Skippy Sharp, Savoye LET, Jokerman Các kiểu chữ LET, Courier New và Woodrow.

Thiết kế nội thất và sản xuất đều diễn ra độc quyền trên Apple Macintoshes-tại Head First, tất cả chúng tôi về "Think Different" (kể cả khi nó không đúng ngữ pháp). Tất cả mã Java được tạo ra bằng James Gosling IDE ưa thích nhất là vi, mặc dù chúng ta thực sự nên thử Eclipse của Erich Gamma.

Những ngày dài viết lách được tiếp thêm năng lượng bởi chất caffeine trong trà Honest Tea và Tejava, không khí trong lành của Santa Fe, và những âm thanh sôi động của Banco de Gaia, Cocteau Twins, Buddha Bar I-VI, Delerium, Enigma, Mike Oldfield, Olive, Orb, Orbital, LTJ Bukem, Massive Attack, Steve Roach, Sasha và Digweed, Thievery Corporation, Zero 7 và Neil Finn (trong tất cả các lần hóa thân của anh ấy) cùng với rất nhiều acid trance và thêm nhiều bản nhạc thập niên 80 mà bạn có thể muốn biết.

Và bây giờ, lời cuối cùng từ Head First Labs...

Các nhà nghiên cứu đẳng cấp thế giới của chúng tôi đang làm việc ngày đêm trong một cuộc đua điện cuồng để

khám phá những bí ẩn của Sự sống, Vũ trụ và Mọi thứ - trước khi quá muộn.

Chưa bao giờ có một nhóm nghiên cứu nào có những mục tiêu cao cả và khó khăn như vậy

lắp ráp. Hiện tại, chúng tôi đang tập trung năng lượng tập thể và sức mạnh trí tuệ của mình vào

tạo ra cỗ máy học tập tối ưu. Khi đã hoàn thiện, bạn và những người khác sẽ tham gia

chúng tôi trong hành trình tìm kiếm của mình!

Bạn thật may mắn khi được cầm trên tay một trong những nguyên mẫu đầu tiên của chúng tôi. Nhưng chỉ

thông qua sự tinh chỉnh liên tục mục tiêu của chúng ta có thể đạt được. Chúng tôi yêu cầu bạn, một người tiên phong

người sử dụng công nghệ, gửi cho chúng tôi các báo cáo thực địa định kỳ về tiến độ của bạn, tại

fieldreports@headfirstlabs.com



bạn đang ở đây 4 639

Bây giờ bạn đã áp dụng phương pháp Head First vào Mẫu thiết kế, tại sao không áp dụng nó vào cuộc sống của mình?

Hãy tham gia cùng chúng tôi tại trang web Head First Labs, nơi bạn có thể tìm thấy các tài nguyên của Head First bao gồm podcast, diễn đàn, mã và nhiều nội dung khác.

Nhưng bạn không chỉ là khán giả; chúng tôi còn khuyến khích bạn tham gia cuộc vui bằng cách thảo luận và đóng góp.

Bạn được lợi ích gì?

- Nhận tin tức mới nhất về những gì xảy ra ở Thế giới ưu tiên trí óc.
- Tham gia vào các cuốn sách và công nghệ sắp ra mắt của chúng tôi.
- Học cách giải quyết những khó khăn đó cách chủ đề kỹ thuật (nói nhanh ba lần) trong thời gian ngắn nhất có thể.

➤ Hãy nhìn vào hậu trường để xem Head

Những cuốn sách đầu tiên đã được viết ra.

- Gặp gỡ các tác giả Head First và đội ngũ hỗ trợ đảm bảo mọi việc diễn ra suôn sẻ.

Tại sao bạn không thử sức mình để trở thành tác giả của Head First?



<http://www.headfirstlabs.com>

The screenshot shows the Head First Labs homepage with four main book sections:

- Head First Design Patterns**: Describes the book as a way to learn design patterns through puzzles and stories.
- Head First Java (2nd edition)**: Describes Java as a complex language that's easy to learn and fun to master.
- Head First Services & J2EE**: Describes the book as a way to pass the Sun Certified Web Component Developer exam.
- Head First EJB**: Describes the book as a way to pass the Sun Certified Business Component Developer exam.

Each book section includes a small thumbnail image of the book cover, a brief description, and a "Read more" link.

Tốt hơn

sách điện tử

Tìm kiếm

bên trong các phiên bản điện tử
của hàng ngàn cuốn sách

Duyệt

sách theo thể loại.

Với Safari, việc nghiên cứu bất kỳ

chủ đề nào cũng trở nên dễ dàng

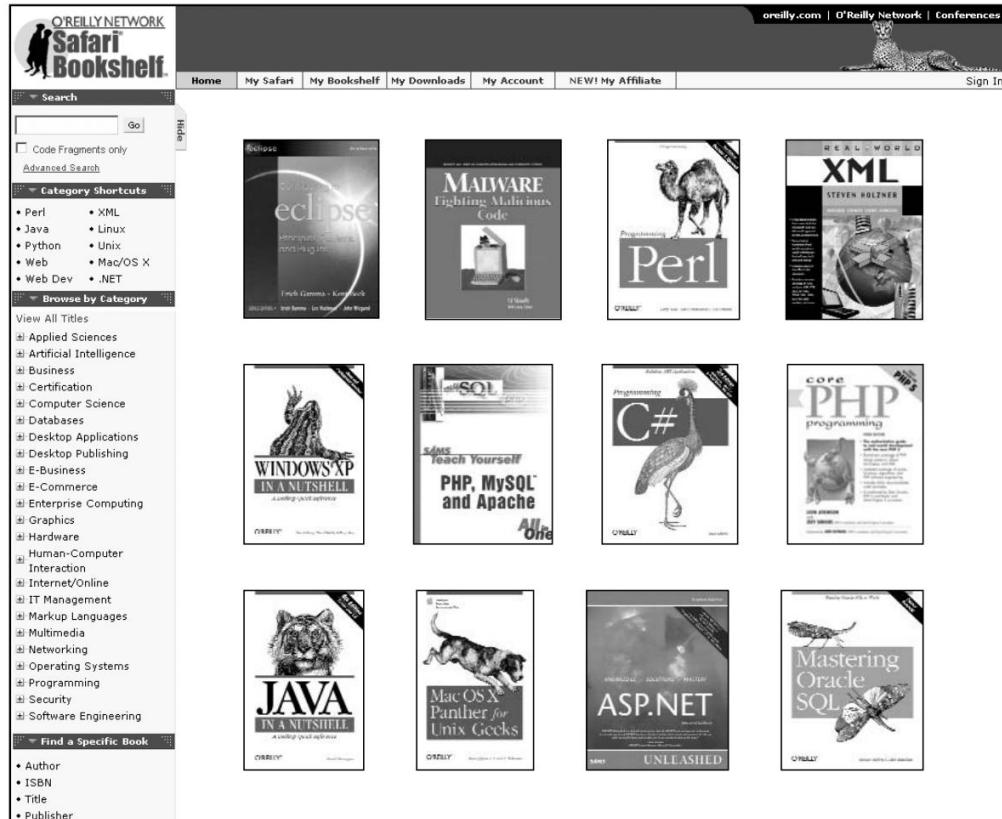
Tìm thấy

trả lời ngay lập tức

Đọc sách từ đầu đến cuối.

Hoặc chỉ cần nhấp vào trang

bạn cần.



Tìm kiếm Safari! Thư viện tham khảo điện tử hàng
đầu dành cho lập trình viên và chuyên gia CNTT



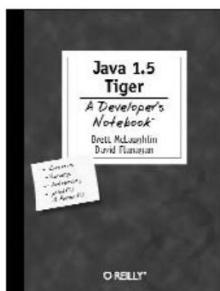
Addison
Wesley
AdobePress

Sum
independent
O'REILLY
SAMS
New Riders

ALPHA
Java
Sun Microsystems
QUE
Cisco Press

Microsoft
Press
Peachpit
Press
macromedia
PRESS
PRINCE
REAL
PTR

Các tựa sách liên quan có sẵn từ O'Reilly



Java

- Ant: Hướng dẫn đầy đủ
- Java tốt hơn, nhanh hơn, nhẹ hơn
- Nhật thực
- Sách dạy nau ân Eclipse
- Enterprise JavaBeans, Phiên bản thứ 4
- Hardcore Java
- Head First Java
- Head First Servlets & JSP Head
- First EJB
- Hibernate:
- Sổ tay của nhà phát triển
- Các mẫu thiết kế J2EE
- Java 1.5 Tiger:
- Sổ tay của nhà phát triển
- Liên kết dữ liệu Java & XML Java & XML Sách
- hướng dẫn Java, Phiên bản thứ 2
- Đối tượng dữ liệu
- Java Các phương pháp hay nhất về cơ sở dữ liệu Java Các phương pháp hay nhất về Java Enterprise Java Enterprise tóm tắt, Phiên bản thứ 2
- Ví dụ Java tóm tắt, Phiên bản thứ 3
- Lập trình Java cực đỉnh
- Sách dạy nau ân
- Java tóm tắt, ấn bản lần thứ 4
- Phần mở rộng quản lý Java
- Dịch vụ tin nhắn Java
- Lập trình mạng Java, Phiên bản 2
- JavaNIO
- Điều chỉnh hiệu suất Java, Phiên bản thứ 2
- RMI Java
- Bảo mật Java, Phiên bản 2
- Khuôn mặt JavaServer
- Java ServerPages, Phiên bản thứ 2
- Sách hướng dẫn Java Servlet & JSP
- Lập trình Java Servlet, Phiên bản 2
- Java Swing, Phiên bản thứ 2
- Tóm tắt về dịch vụ web Java
- Học Java, Phiên bản 2
- Mac OS X dành cho Java Geeks
- Lập trình Jakarta Struts phiên bản 2
- Tomcat: Hướng dẫn đầy đủ
- WebLogic:
- Hướng dẫn đứt khoát

O'REILLY®

Sách của chúng tôi có bán tại hầu hết các hiệu sách bán lẻ và trực tuyến.

Để đặt hàng trực tiếp: 1-800-998-9938 • order@oreilly.com • www.oreilly.com Các phiên bản trực tuyến của hầu hết các tựa sách của O'Reilly đều có sẵn bằng cách đăng ký tại safari.oreilly.com

Giữ liên lạc với O'Reilly

1. Tải xuống các ví dụ từ sách của chúng tôi Để tìm tệp vi
dụ cho một cuốn sách, hãy truy cập: www.oreilly.com/
catalog chọn cuốn sách và nhấp
vào liên kết "Ví dụ".

2. Đăng ký sách O'Reilly của bạn Đăng ký sách
của bạn tại register.oreilly.com Tại sao phải đăng ký
sách của bạn?
Sau khi bạn đã đăng ký sách O'Reilly, bạn có thể: • Thắng sách O'Reilly,
áo phông hoặc phiếu giảm giá trong đợt rút thăm hàng tháng
của chúng tôi. • Nhận các ưu đãi đặc biệt chỉ
dành cho khách hàng đã đăng ký của O'Reilly.

- Nhận danh mục thông báo về sách mới (chỉ dành cho Hoa
Ky và Vương quốc Anh).
- Nhận thông báo qua email về các phiên bản mới của
Sách O'Reilly mà bạn sở hữu.

3. Tham gia danh sách email của chúng tôi

Đăng ký để nhận thông báo qua email theo chủ đề về sách và hội nghị
mới, các ưu đãi đặc biệt và bản tin công nghệ của O'Reilly Network
tại: elists.oreilly.com. Bạn có thể dễ dàng tùy chỉnh đăng
ký elists miễn phí để
nhận được chính xác tin tức O'Reilly mà mình mong muốn.

4. Nhận tin tức, mẹo và công cụ mới nhất www.oreilly.com

- "100 trang web hàng đầu trên web"-Tạp chí PC • Giải thưởng Web Business 50 của Tạp
chi CIO Trang web của chúng tôi chứa thư viện thông tin sản
phẩm toàn diện (bao gồm trích đoạn sách và mục lục), phần mềm có
thể tải xuống, bài viết cơ bản, phòng vấn với các nhà lãnh đạo công
nghệ, liên kết đến các trang web có liên quan, ảnh bìa sách, v.v.

5. Làm việc cho O'Reilly Hãy

truy cập trang web của chúng tôi để biết các cơ hội việc làm hiện
tại: jobs.oreilly.com

6. Liên hệ với chúng tôi

O'Reilly và công sự
1005 Đường cao tốc Gravenstein phía Bắc
Sebastopol, CA 95472 Hoa Kỳ
ĐT: 707-827-7000 hoặc 800-998-9938

(6 giờ sáng đến 5 giờ chiều theo giờ PST)

SỐ FAX: 707-829-0104

order@oreilly.com
Để được giải đáp các vấn đề liên quan đến đơn hàng của bạn hoặc sản phẩm
của chúng tôi. Để đặt mua sách trực tuyến, hãy truy cập:
www.oreilly.com/order_new

catalog@oreilly.com
Để yêu cầu một bản sao danh mục mới nhất của chúng tôi.

booktech@oreilly.com

Đối với các câu hỏi kỹ thuật hoặc chỉnh sửa nội dung sách.

corporate@oreilly.com

Dành cho mục đích giáo dục, thư viện, chính phủ và bán
hang cho doanh nghiệp.

suggestions@oreilly.com

Để gửi đề xuất sách mới cho biên tập viên và quản lý sản
phẩm của chúng tôi.

international@oreilly.com

Để biết thông tin về các nhà phân phối quốc tế hoặc các câu
hỏi về bản dịch. Để biết danh sách các nhà phân phối của
chúng tôi bên ngoài Bắc Mỹ, hãy xem:

international.oreilly.com/distributors.html

adoption@oreilly.com Để
biết thông tin về việc sử dụng sách của O'Reilly cho mục đích học
thuật, hãy truy

cập: academic.oreilly.com



Sách của chúng tôi có bán tại hầu hết các hiệu sách bán lẻ và trực tuyến.

Để đặt hàng trực tiếp: 1-800-998-9938 • order@oreilly.com • www.oreilly.com Các phiên bản trực tuyến
của hầu hết các tựa sách của O'Reilly đều có sẵn bằng cách đăng ký tại safari.oreilly.com