TRẦN ĐÌNH QUẾ

GIÁO TRÌNH

KIẾN TRÚC
VÀ THIẾT KẾ PHẦN MỀM

HÀ NỘI - 2017

MỤC LỤC

LỜI GIỚI THIỆU

Kiến trúc phần mềm được xem là trung tâm của nghiên cứu và phát triển các hệ thống phần mềm cỡ lớn hiện nay. Quá trình của phát triển được xoay quanh khái niệm kiến trúc nhằm mục đích phát triển sản phẩm một cách có hiệu quả, năng suất và chất lượng cao. Tài liệu này được dành cho sinh viên năm cuối khi học môn kiến trúc và thiết kế phần mềm. Tài liệu được xây dựng dựa trên cơ sở là bạn đọc đã có được kiến thức và kỹ năng cơ bản về Phân tích và thiết kế phần mềm, lập trình hướng đối tượng và có thể biểu diễn cấu trúc và hành vi của hệ thống với ngôn ngữ mô hình UML.

Mục đích của tài liệu là nâng cao kiến thức và kỹ năng về thiết kế phần mềm cho sinh viên ngành công nghệ thông tin. Nội dung tập trung vào các cách tiếp cận liên quan đến kiến trúc và thiết kế phần mềm cũng như mẫu thiết kế. Phần kiến trúc phần mềm chủ yếu trình bày cách tiếp cận dựa trên thành phần và các công nghệ thành phần được sử dụng rộng rãi trong công nghiệp phần mềm hiện nay. Phần mẫu thiết kế trình bày khá đầy đủ những mẫu thông dụng nhằm cung cấp cho sinh viên các kiến thức cơ bản và kỹ năng áp dụng. Nội dung của tài liệu được chia thành 11 Chương:

Chương 1: Khái niệm cơ bản về kiến trúc phần mềm

Giới thiệu một số khái niệm liên quan đến thiết kế và kiến trúc cũng như tính phức tạp của phần mềm. Qua đó thấy được lý do tại sao kiến trúc phần mềm được quan tâm đặc biệt trong việc phát triển các hệ phần mềm cỡ lớn. Đồng thời tập trung trình bày một số khái niệm cơ bản làm cơ sở cho trình bày các chương tiếp theo.

Chương 2: Thiết kế kiến trúc và các kiểu kiến trúc

Chương này tập trung trình bày một số chủ đề liên quan đến tiến trình thiết kế kiến trúc, các nguyên ly thiết kế và các kiểu kiến trúc.

Chương 3: Mô hình hóa kiến trúc

Chương này nhằm trình bày một số khái niệm cơ bản sẽ được sử dụng để thể hiện kiến trúc từ những thành phần đơn giản đến các tính chất phức tạp hơn của hệ thống như các mô hình hành vi. Mô hình kiến trúc có thể biểu diễn bởi ngôn ngữ tự nhiên hay ngôn ngữ rất hình thức và hạn chế ngữ nghĩa như ngôn ngữ mô tả kiến trúc Rapide. Biểu diễn kiến trúc với ngôn ngữ mô hình hóa quen thuộc UML được trình bày khá đầy đủ.

Chương 4: Kiến trúc phần mềm dựa trên thành phần

Nội dung chương này tập trung trình bày mô hình thành phần và đặc trưng của mô hình này để làm cơ sở cho hai chương tiếp theo. Mô hình thành phần với UML được tiếp tục trình bày với biểu đồ thành phần và biểu đồ triển khai.

Chương 5: Mô hình thành phần với EJB

Giới thiệu khung J2EE và EJB, các khái niệm về thành phần EJB và môi trường runtime của nó. Nội dung cũng trình bày về các kiểu cuả EJB, kết nối và việc triển khai của chúng, giới thiệu các tính năng của các phiên bản EJB và phân biệt giữa lời gọi hàm đồng bộ và không đồng bộ. Nội dung cũng đề cập hướng dẫn từng bước để xây dựng, triển khai và sử dụng

thành phần EJB.

Chương 6: Mô hình thành phần .NET

Giới thiệu khung .NET, các khái niệm chung của các thành phần .NET, các kiểu thành phần .NET, kết nối giữa các thành phần, và cách triển khai chúng, các thành phần cục bộ và phân tán, phân biệt các thành phần kết nối và hợp, gọi phương thức đồng bộ và không đồng bộ. Nội dung cũng trình bày hướng dẫn từng bước để xây dựng, triển khai, và sử dụng các thành phần .NET.

Chương 7: Kiến trúc và mẫu thiết kế

Chương này trình bày tổng quan về khái niệm mẫu thiết kế, các nhóm mẫu thiết kế và những đặc trưng của mẫu thiết kế. Một số vấn đề liên quan đến phân loại, tích hợp cũng đã được đề cập đến. Chi tiết mô hình và cài đặt các mẫu thiết kế dành cho các chương tiếp theo.

Chương 8: Các mẫu thiết kế tạo dựng

Chương này tập trung trình bày nhưng mẫu thiết kế tạo dựng được đề cập đến trong chương 7. Trong mỗi mẫu thiết kế đều có bàn đến lý do cho sự ra đời của mẫu, biểu đồ lớp cho mẫu và tình huống áp dụng cùng cài đặt.

Chương 9: Các mẫu thiết kế cấu trúc

Chương này tập trung trình bày nhưng mẫu thiết kế cấu trúc liên quan đến thiết kế cấu trúc các lớp. Trong mỗi mẫu thiết kế đều có bàn đến lý do cho sự ra đời của mẫu, biểu đồ lớp cho mẫu và tình huống áp dụng cùng cài đặt.

Chương 10: Các mẫu thiết kế hành vi

Chương này tập trung trình bày nhưng mẫu thiết kế hành vi liên quan đến thiết kế các thuật toán. Trong mỗi mẫu thiết kế đều có bàn đến lý do cho sự ra đời của mẫu, biểu đồ lớp cho mẫu và tình huống áp dụng cùng cài đặt.

Chương 11: Case study

Chương này trình bày hai Case Study nhằm minh họa áp dụng một số mẫu vào hai hệ khác nhau là Hệ quản lý Cơ sở dữ liệu và Hệ Quản lý bán sách trực tuyến BookStore.

Tài liệu này được ra đời dựa trên sự đóng góp của nhiều người. Trước hết, tác giả xin bày tỏ lòng biết ơn đến tất cả Thầy-Cô trong Khoa CNTT và các Thầy-Cô trong Hội đồng thẩm định đã có nhiều góp ý xây dựng. Đặc biệt cám ơn các tác giả tài liệu trích dẫn và các tác giả mã nguồn đã được sử dụng trong tài liệu này. Cám ơn các bạn sinh viên chuyên ngành Công nghệ phần mềm qua nhiều thế hệ của Học viện đã đóng góp cho tài liệu này. Tác giả xem tài liệu này chỉ là sự lắp ghép các mẫu tài liệu có sẵn được sắp xếp theo quan điểm của mình để giúp sinh viên nhanh chóng và dễ dàng tiếp cận môn học. Để hiểu một cách sâu sắc và đầy đủ hơn những chủ đề trong tài liệu, bạn đọc nên tham khảo các tài liệu gốc được liệt kê trong tài liệu tham khảo và chúng nó thực sự đáng để đọc hơn.

Tác giả

CHƯƠNG 1: KHÁI NIỆM CƠ BẢN VỀ KIẾN TRÚC PHẦN MỀM

Chương này tập trung trình bày một số nội dung sau đây:

- Độ phức tạp phần mềm
- Khái niệm kiến trúc phần mềm
- Một số khái niệm cơ bản khác

1.1 ĐỘ PHỨC TẠP PHẦN MỀM

Phần mềm được hiểu là bao gồm chương trình cùng với dữ liệu và tài liệu liên quan nhằm tự động thực hiện một số chức năng hoặc giải quyết một vấn đề cụ thể nào đó. Thông thường khi nói đến phần mềm người ta thường nhấn mạnh đến chương trình. Do đó, trong tài liệu này hai khái niệm phần mềm và chương trình được sử dụng cùng nhau một cách linh hoạt. Phần mềm thực thi các chức năng của nó bằng cách gửi các lệnh trực tiếp đến phần cứng hoặc cung cấp dữ liệu để phục vụ các chương trình hay phần mềm khác. Phần mềm là một khái niệm trừu tượng và khác với phần cứng ở chỗ là không thể sờ mó hay đụng chạm vào và cũng không bị mòn hay phai mờ theo thời gian, và nó cần phải có phần cứng mới có thể thực thi được. Để hiểu rõ phần mềm và các cách tiếp cận trong phát triển phần mềm, phần này sẽ dành trình bày một số đặc trưng về độ phức tạp của phần mềm.

1.1.1 Độ phức tạp phần mềm là gì?

Ngay từ những ngày sơ khai, ngành khoa học máy tính đã gặp phải những vấn đề liên quan đến sự phức tạp như chọn được một cấu trúc dữ liệu phù hợp, phát triển những thuật toán hiệu quả, cách phân rã các chức năng...Máy tính, mạng máy tính và các thiết bị thông minh di động với những kiến trúc phần cứng khác nhau ngày nay đã nâng độ phức tạp của phần mềm lên một mức độ mới với những kiểu phần mềm khác nhau và kiến trúc khác nhau. Độ phức tạp phần mềm (software complexity) có thể được xem xét từ những quan điểm khác nhau. Nó liên quan mật thiết đến kiến trúc phần mềm như trong định nghĩa sau đây của Taylor đã được nhiều người chấp nhận [18]:

Độ phức tạp là bản chất vốn có của một hệ phần mềm. Nó tỷ lệ với kích cỡ của hệ thống, số lượng thành phần cấu thành hệ thống, kích cỡ và cấu trúc bên trong của mỗi thành phần, số lượng và tính chất phụ thuộc nhau giữa các thành phần.

Để hiểu một cách trực giác hơn các khái niệm "kích cỡ", "cấu trúc bên trong", "tính chất phụ thuộc", chúng ta hãy lấy ví dụ từ những khái niệm của lập trình hướng đối tượng. Kích cỡ của một phần mềm hướng đối tượng có thể đo theo số dòng mã nguồn, số lớp, số các môđun hay số gói; cấu trúc bên trong và phụ thuộc được thể hiện ở cấu trúc của các lớp, gói và quan hệ giữa các lớp, gói.

1.1.2 Độ phức tạp và thành phần phần mềm

Trong kỹ thuật phần mềm truyền thống, mỗi kiểu tác vụ thực thi bởi hệ thống được xác định bởi một hay tập các thành phần liên quan nhau. Điều này xuất phát từ những nguyên lý cơ bản

về phát triển phần mềm dựa vào các khái niệm: *trừu tượng hóa, môđun hóa, tách các chức năng, cô lập thay đổi*. Rõ ràng rằng một hệ phần mềm với số lượng thành phần lớn hơn sẽ phức tạp hơn so với hệ có số lượng thành phần ít hơn.

Thoạt nhìn để giảm độ phức tạp phần mềm thì chỉ cần tối thiểu hóa số thành phần trong hệ thống. Nghĩa là để giảm số thành phần chúng ta chỉ cần phải gộp nhiều chức năng vào trong một thành phần. Tuy nhiên, việc giảm số thành phần trong hệ thống không đảm bảo rằng độ phức tạp sẽ thấp hơn vì điều này có thể làm tăng thêm sự phức tạp của mỗi thành phần riêng lẻ. Nói cách khác, một hệ thống được cấu tạo bởi các thành phần lớn hơn sẽ phức tạp hơn là được cấu tạo bởi các thành phần nhỏ hơn. Hơn nữa, nhiều nghiên cứu đã chỉ ra rằng độ phức tạp của hệ phần mềm còn liên quan đến cả tương tác giữa các thành phần cũng như cấu trúc bên trong và hành vi của riêng các thành phần đó [18].

Độ phức tạp phần mềm được xem là vấn đề căn bản cần phải được giải quyết và khắc phục bởi các công cụ và phương pháp tạo ra phần mềm. Nhiều nghiên cứu đã chỉ ra rằng chất lượng của các sản phẩm phần mềm cũng như tiến độ dự án phần mềm phụ thuộc rất nhiều vào độ phức tạp phần mềm. Tuy nhiên, cho đến nay chưa có một giải pháp nào thực sự hiệu quả để loại bỏ được độ phức tạp hoặc nâng cao năng suất cũng như chất lượng phần mềm. Điều mà những người phát triển có thể làm được là giảm thiểu hoặc loại bớt tính phức tạp đi mà thôi.

Môđun hóa được xem là nguyên tắc cơ bản để đạt được sự đơn giản trong thiết kế phần mềm và có thể làm giảm thiểu cũng như quản lý được độ phức tạp. Khi đó, độ phức tạp trong cấu trúc và xử lý có thể được tính toán dựa vào sự *liên kết* của các thành phần trong một hệ thống. Nghĩa là nếu một hệ thống hoặc một tiến trình càng bao gồm nhiều thành phần kết nối nhau theo nhiều kiểu cấu trúc khác nhau thì độ phức tạp của hệ thống và các tiến trình đó càng lớn. Khi đó các kết nối sẽ trở thành phụ thuộc lẫn nhau.

Ví dụ, trong một cấu trúc phần mềm, thành phần B gọi là *phụ thuộc* vào thành phần A, nếu sửa đổi thành phần A thì thành phần B cũng sẽ bị ảnh hưởng theo. Vì vậy, cần phải sửa đổi thành phần B theo hướng đã sửa đổi thành phần A để hoàn thiện sản phẩm và không gây lỗi. Nghĩa là, nếu thay đổi thành phần A thì cũng phải thay đổi thành phần B. Trong trường hợp hai thành phần này phụ thuộc lẫn nhau tức là thành phần A phụ thuộc vào thành phần B và thành phần B cũng phụ thuộc vào thành phần A thì sự thay đổi của một trong hai thành phần này sẽ dẫn đến một vòng lặp sửa đổi. Điều này dễ hiểu vì khi chỉnh sửa A thì sau đó phải chỉnh sửa B và việc chỉnh sửa B lại dẫn đến việc phải chỉnh sửa A và quá trình này cứ lặp đi lặp lại.

1.1.3 Ảnh hưởng của độ phức tạp phần mềm

Độ phức tạp nảy sinh qua nhiều pha phát triển và thể hiện ở nhiều công đoạn trong quá trình phát trình phần mềm, bao gồm:

- Pha xác định và phân tích yêu cầu
- Phát thảo giao diện người dùng
- Thiết kế kiến trúc
- Thiết kế chi tiết
- Tạo mã nguồn

- Tích hợp…

Chúng ta có thể làm giảm độ phức tạp của thiết kế hệ thống bằng cách lựa chọn ngôn ngữ thiết kế hay chọn giao diện phù hợp hoặc cả hai. Nếu giảm thiểu được một câu hay thuật ngữ của một định nghĩa bằng cách thay đổi ngữ nghĩa của những từ cần được mô tả thì rõ ràng có thể giảm được độ phức tạp. Với ngôn ngữ tự nhiên, có thể giảm độ phức tạp bằng cách đưa vào các thuật ngữ chuẩn. Một từ duy nhất hoặc một tên hợp lý có thể chứa đựng rất nhiều thông tin. Như vậy, *toàn bộ công việc nhằm giảm độ phức tạp phần mềm xoay quanh việc làm sao tạo ra các thuật ngữ để thể hiện các khái niệm và các mối liên quan giữa các khái niệm.*

Cho đến nay, giới công nghệ phần mềm đã đề xuất một số khái niệm như *mẫu kiến trúc* (architectural pattern), *mẫu thiết kế* (design pattern), *thành phần* (component) với hy vọng giảm được độ phức tạp này. Tuy nhiên, việc có giảm được độ phức tạp hay không còn phụ thuộc vào điều các mẫu kiến trúc, mẫu thiết kế hay thành phần đã được hiểu một cách rõ ràng và chính xác chưa. Mặc dù cấu trúc bên trong của hệ thống lúc này vẫn còn phức tạp do sự tương tác giữa các thành phần nhưng nó khiến cho hệ thống được nhìn một cách đầy đủ và tổng quát hơn. Cho đến nay *các khái niệm mẫu thiết kế, mẫu kiến trúc hay thành phần thường được sử dụng để mô tả cấu trúc của phần mềm cũng như bàn luận về thiết kế và kiến trúc phần mềm*. Các khái niệm này sẽ được trình bày đầy đủ hơn trong phần còn lại của chương cũng như các chương sau và có thể xem là những chủ đề xuyên suốt trong tài liệu này.

1.2 KIẾN TRÚC PHẦN MỀM

Thuật ngữ *kiến trúc phần mềm* (software architecture) lần đầu tiên được sử dụng[1] vào cuối những năm 1960 nhưng mãi đến những năm 1990 khái niệm này mới được xem xét, nghiên cứu rộng rãi trong công nghệ phần mềm. Kiến trúc phần mềm được hiểu như một khái niệm quan trọng bắt nguồn từ những nghiên cứu của Dijkstra vào năm 1968 và Parnas vào đầu những năm 1970. Các nghiên cứu này đã chỉ ra rằng *kiến trúc của hệ phần mềm đóng một vai trò quan trọng và làm thế nào có được một kiến trúc hợp lý là vấn đề cốt lõi trong quá trình phát triển phần mềm*. Những năm 1990 đã chứng kiến nhiều nỗ lực nghiên cứu nhằm đề xuất các kiểu kiến trúc phần mềm, các ngôn ngữ mô tả kiến trúc, các mẫu thiết kế, các mô hình thành phần phần mềm, các phương pháp hình thức để mô hình hóa kiến trúc…Động lực chính cho sự quan tâm mạnh mẽ này là mã nguồn của nhiều hệ phần mềm phổ biến đã lên đến hàng trăm nghìn hay hàng triệu dòng lệnh.

Kiến trúc hệ thống (system architecture) là một sự gắn kết các bộ phận trong một hệ thống với nhau bao gồm cấu trúc, giao diện và các cơ chế để phối hợp hoạt động giữa các bộ phận. Khi xác định được kiến trúc phù hợp, chúng ta sẽ dễ dàng chuyển đổi, tìm được vị trí của một chức năng hay chỉ ra được nơi có thể thêm một chức năng mới phù hợp với kiến trúc chung. Một kiến trúc tốt [18] phải được chi tiết hóa đủ để có thể ánh xạ thành mã nguồn thực sự và cho phép thêm những chức năng mới hay những khái niệm mới mà không ảnh hưởng đến phần còn lại của hệ thống.

[1] http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF

Kiến trúc một hệ phần mềm được hiểu là một quá trình xác định một giải pháp để đưa ra một cấu trúc thích hợp nhằm đáp ứng được các yêu cầu về kỹ thuật và hoạt động của một tổ chức hay doanh nghiệp đồng thời có thể tối ưu hóa về mặt hiệu năng, an toàn và khả năng quản lý.

Quá trình này liên quan đến một loạt các quyết định và mỗi quyết định có thể có tác động thực sự đến chất lượng, hiệu năng, bảo trì, và thành công chung của một hệ ứng dụng. Để hiểu rõ hơn khái niệm kiến trúc, chúng ta sẽ xem xét một số định nghĩa đã được đưa ra gần đây [13]:

- Booch đã định nghĩa kiến trúc phần mềm là một loạt các quyết định quan trọng về tổ chức một hệ thống phần mềm: Lựa chọn các thành phần của cấu trúc, giao diện của hệ thống; Xác định kiểu tương tác giữa các thành phần; Xác định các yếu tố cơ bản cấu thành hệ thống; Xác định kiểu kiến trúc của hệ thống; Khả năng tái xử dụng...
- Martin Fowler cũng đã chỉ ra một số chủ đề phổ biến khi giải thích về kiến trúc như sau: việc phân rã hệ thống ở mức cao thành các bộ phận, những quyết định khó thay đổi, nhiều kiến trúc trong hệ thống, cái gì có thể thay đổi trong vòng đời của hệ thống...
- Bass et al. [3] đã định nghĩa kiến trúc phần mềm là một hay nhiều cấu trúc của hệ thống bao gồm các thành phần phần mềm và các quan hệ giữa chúng.

Như vậy, kiến trúc một hệ thống là tìm cách xây dựng một liên kết giữa các yêu cầu nghiệp vụ và các yêu cầu kỹ thuật bằng việc hiểu các ca sử dụng (use case) và sau đó tìm cách cài đặt những ca sử dụng này vào trong phần mềm.

Mục tiêu của kiến trúc là chỉ ra được các yêu cầu có ảnh hưởng như thế nào đến cấu trúc của ứng dụng. Kiến trúc tốt có thể giảm được rủi ro về nghiệp vụ trong khi xây dựng một giải pháp kỹ thuật và có thể xử lý được các sự cố xảy ra liên quan đến phần cứng hay phần mềm trong vòng đời của nó. Kiến trúc sư phần mềm cần phải xem xét ảnh hưởng tổng thể của những quyết định thiết kế, sự cân bằng vốn có của các thuộc tính chất lượng (như hiệu năng và bảo mật) và sự cân bằng cần thiết của yêu cầu người dùng, hệ thống và yêu cầu nghiệp vụ. Sau đây là một số hướng dẫn khi kiến trúc một hệ thống:

- Kiến trúc cần thể hiện được cấu trúc (structure) của hệ thống nhưng ẩn giấu các chi tiết cài đặt.
- Kiến trúc phải hiện thực hóa được tất cả các ca sử dụng và kịch bản.
- Kiến trúc phải thể hiện được sự quan tâm đến các yêu cầu của các bên liên quan.
- Kiến trúc phải đáp ứng cả yêu cầu về chức năng lẫn yêu cầu về phi chức năng

1.3 MỘT SỐ KHÁI NIỆM CƠ BẢN

Trong các phần trình bày trên, chúng ta đã bàn một số vấn đề liên quan đến khái niệm của kiến trúc phần mềm như tính phức tạp, thành phần...Phần này nhằm trình bày chi tiết hơn một số định nghĩa về các thuật ngữ và những ý tưởng cơ bản về kiến trúc phần mềm để làm cơ sở cho trình bày các chương về sau.

1.3.1 Kiến trúc

Theo Taylor [18] *kiến trúc một hệ phần mềm là một tập các quyết định thiết kế chủ yếu được đưa ra cho hệ thống cần xây dựng*. Như vậy, kiến trúc phần mềm được xem là những định hướng cho phát triển và tiến hóa một hệ phần mềm. Khái niệm *quyết định thiết kế* (design decision) là trung tâm của kiến trúc phần mềm nhằm kết nối các khái niệm khác liên quan đến kiến trúc. Sau đây là một số khái niệm liên quan đến quyết định thiết kế:

- Các quyết định liên quan đến *cấu trúc* (structure) của hệ thống. Ví dụ, các thành phần kiến trúc cần phải tổ chức và hợp thành các gói phân cấp theo kiểu 3 tầng.
- Các quyết định liên quan đến *hành vi chức năng* (functional behavior). Ví dụ, xử lý dữ liệu, lưu trữ và hiển thị sẽ được thực thi theo một thứ tự nghiêm ngặt.
- Các quyết định liên quan đến *tương tác* (interaction). Ví dụ, giao tiếp giữa các thành phần hệ thống chỉ xảy ra bằng cách sử dụng thông báo, sự kiện hay giao thức RMI trong java.
- Các quyết định liên quan đến *các tính chất phi chức năng* (nonfunctional properties) của hệ thống. Ví dụ, khả năng phụ thuộc của hệ thống sẽ được đảm bảo bằng cách phục hồi các môdun xử lý.
- Các quyết định liên quan đến *cài đặt* (implementation) hệ thống. Ví dụ, thành phần giao diện người sử dụng sẽ được xây dựng bằng cách sử dụng Java Swing hay JSP.

Rõ ràng các quyết định quan trọng khác nhau sẽ dẫn đến các kiến trúc khác nhau. Tuy nhiên, một số quyết định không được xem là chủ yếu và không ảnh hưởng đến kiến trúc của hệ thống như chi tiết của thuật toán hay cấu trúc dữ liệu.

1.3.2 Thành phần

Các quyết định tạo nên kiến trúc một hệ phần mềm bao gồm việc kết hợp nhiều phần tử khác biệt nhưng có liên quan với nhau. Các phần tử này bao gồm những khía cạnh khác nhau:

- Xử lý: liên quan đến chức năng hay hành vi
- Trạng thái: liên quan đến thông tin hay dữ liệu
- Tương tác: liên quan đến cộng tác, phối hợp…

Những phần tử đóng gói xử lý và dữ liệu bên trong kiến trúc hệ thống gọi là *thành phần phần mềm* (software component). Theo Taylor [18]:

Thành phần phần mềm là một thực thể kiến trúc: (i) đóng gói một tập con chức năng và dữ liệu của hệ thống; (ii) chỉ có thể truy nhập vào thực thể thông qua một giao diện; (iii) xác định các phụ thuộc tùy theo ngữ cảnh mà nó yêu cầu.

Như vậy, thành phần phần mềm là sự tích hợp của tính toán và trạng thái trong hệ thống. Tùy theo kiểu kiến trúc, quan điểm người thiết kế hay yêu cầu của hệ thống, thành phần có thể đơn giản như một thao tác, một phương thức hay phức tạp như một phần hệ thống. Đặc trưng quan trọng nhất của thành phần là chỉ được nhìn thấy từ bên ngoài qua giao diện. Thành phần phần mềm được cho là thể hiện đầy đủ các nguyên lý *đóng gói* (encapsulation), *trừu*

tượng hóa (abstraction) và *môdun hóa* (modularity) của kỹ thuật phần mềm. Như vậy, các thành phần hàm ý có khả năng kết hợp, sử dụng lại và tiến hóa.

Tuy nhiên, các khả năng này lại được thể hiện trong ngữ cảnh tạo ra thành phần như giao diện phần mềm, tính có sẵn của tài nguyên (dữ liệu…), thùng chứa thành phần, môi trường thực thi của chương trình, hệ điều hành, giao thức mạng, cấu hình phần cứng…

1.3.3 Kết nối

Một đặc trưng cơ bản quan trọng nữa của hệ phần mềm là *tương tác* (interaction) giữa các thành phần trong hệ thống. Nhiều hệ thống hiện đại được xây dựng với nhiều thành phần phức tạp, phân tán qua mạng và cập nhật liên tục suốt trong vòng đời của nó. Do đó, đối với người phát triển việc đảm bảo tương tác giữa các thành phần thậm chí quan trọng hơn chính chức năng của bản thân các thành phần.

Kết nối là một phần tử kiến trúc có nhiệm vụ thực thi tương tác giữa các thành phần.

Đối với hệ phần mềm trên máy để bàn truyền thống, các kết nối thường được thể hiện bởi lời *gọi thủ tục* hay *truy nhập dữ liệu chung*. Các kết nối này thường chỉ thể hiện sự tương tác giữa các cặp thành phần. Gọi thủ tục được cài đặt trực tiếp trong các ngôn ngữ lập trình để giúp trao đổi đồng bộ dữ liệu và điều khiển giữa các cặp thành phần. Khi đó, thành phần triệu gọi chuyển luồng điều khiển cũng như dữ liệu dưới dạng tham số đến thành phần được gọi và sau khi thực hiện yêu cầu, bên được gọi trả lại điều khiển cũng như kết quả tính toán cho bên gọi. Trong truy nhập dữ liệu, kiểu kết nối được thể hiện dưới dạng biến toàn cục hay bộ nhớ chung. Kiểu kết nối này cho phép nhiều thành phần tương tác bằng cách đọc hay ghi vào các phương tiện chung. Tương tác được phân bố không đồng bộ theo thời gian nghĩa là việc đọc không phụ thuộc hay ràng buộc theo thời gian của việc ghi và ngược lại.

Tuy nhiên, trong các hệ phần mềm phức tạp trên mạng, các kết nói thường xảy ra cùng một lúc giữa một thành phần với nhiều thành phần dịch vụ khác nhau. Dựa vào cách thực thi dịch vụ tương tác, người ta thường phân các kết nối thành tám kiểu khác nhau [18]:

- *Kết nối gọi thủ tục*: ví dụ như phương thức trong lập trình hướng đối tượng.
- *Kết nối sự kiện*: ví dụ trong ứng dụng windows, các input GUI xem như sự kiện kích hoạt hệ thống.
- *Kết nối truy nhập dữ liệu:* ví dụ các cơ chế truy vấn trong cơ sở dữ liệu SQL
- *Kết nối liên kết*: được sử dụng để ràng buộc các thành phần cùng giữ một trạng thái suốt trong quá trình tương tác với nhau.
- *Kết nối luồng:* được sử dụng để truyền một khối lượng lớn dữ liệu giữa các tiến trình tự chủ riêng. Ví dụ, các socket TCP/UDP hay các giao thức client-server như RMI, CORBA, FTP, SOAP.
- *Kết nối trung gian:* được sử dụng khi các thành phần biết sự có mặt của các thành phần khác nhưng không thể biết được nhu cầu, trạng thái của nó. Khi đó, bộ phận trung gian sẽ cung cấp dịch vụ phối hợp hay giải quyết tranh chấp…

- *Kết nối thích nghi:* cung cấp phương tiện để hỗ trợ tương tác giữa các thành phần không được thiết kế với tương tác. Ví dụ, hệ phân tán có thể dựa vào gọi thủ tục từ xa (RPC: remote procedure call) cho mọi tương tác.
- *Kết nối theo tuyến:* được sử dụng để chỉ ra các tuyến tương tác và thực thi định tuyến truyền thông cũng như phối hợp giữa các thành phần theo tuyến này.

Thông thường các hệ phân tán sử dụng nhiều cách kết nối với nhau. Ví dụ, trong tính toán lưới, kết nối phân tán dữ liệu dựa trên hợp của bốn kiểu kết nối: gọi thủ tục, truy nhập dữ liệu, luồng và kết nối theo tuyến. Các kết nối này phân phối một khối lượng lớn dữ liệu giữa các thành phần trong môi trường này. Trong các hệ P2P, kết nối phân tán gồm bốn kiểu: trung gian, truy nhập dữ liệu, luồng và kết nối theo tuyến. Trong các hệ phân tán dựa trên mô hình client-server, kết nối gồm bốn kiểu: gọi thủ tục, truy nhập dữ liệu, luồng và kết nối theo tuyến.

1.3.4 Cấu hình

Các thành phần và kết nối thường được kết hợp với nhau theo cách nào đó trong cấu trúc để xác lập mục tiêu của hệ thống. Cách kết hợp này thể hiện cấu hình hay tôpô của hệ thống. Như vậy, *cấu hình của kiến trúc là tập các cách kết hợp giữa các thành phần và kết nối của kiến trúc hệ phần mềm*.

Cấu hình có thể biểu diễn như một đồ thị trong đó các đỉnh biểu diễn thành phần và kết nối, các cạnh biểu diễn sự kết hợp.

1.3.5 Kiểu kiến trúc

Kiểu kiến trúc là là tập những quyết định thiết kế: (i) áp dụng được trong ngữ cảnh phát triển; (ii) ràng buộc những quyết định thiết kế cho hệ thống đó; (iii) thể hiện được chất lượng và lợi ích trong kiểu hệ thống như vậy.

Nhiều kinh nghiệm và nghiên cứu đã chỉ ra rằng trong các hệ phân tán, kiểu kiến trúc thỏa một số điều kiện sau đây sẽ đem nhiều lợi điểm:

- Tách biệt về mặt vật lý: các thành phần phần mềm yêu cầu dịch vụ tách biệt với các thành phần cung cấp dịch vụ. Điều này cho phép cấu trúc các thành phần một cách hợp lý, hiệu quả cả về số lượng yêu cầu và cung cấp dịch vụ.
- Trong suốt: làm cho nhà cung cấp không biết định danh của bên yêu cầu để dễ bề cung cấp dịch vụ một cách trong suốt kể cả khi thay đổi bên yêu cầu.
- Cô lập: tách biệt các bên yêu cầu với nhau để cho phép thêm, loại bỏ hay sửa đổi một cách độc lập.

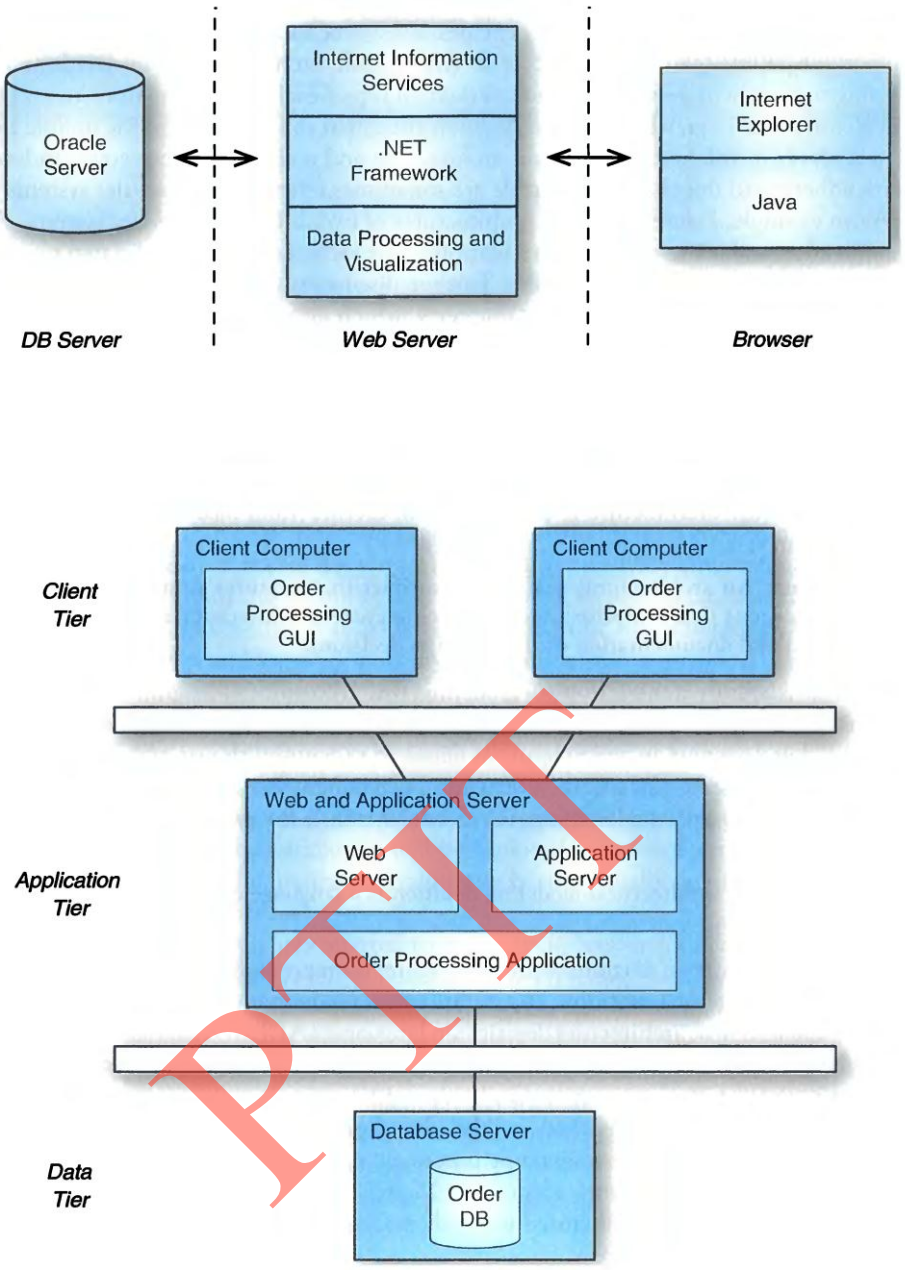
1.3.6 Mẫu kiến trúc

Các kiểu kiến trúc đưa ra những quyết định thiết kế và ràng buộc tổng quát. Chúng cần được mịn hóa thành các quyết định cụ thể dành riêng cho hệ thống cần xây dựng. Trong khi đó, mẫu kiến trúc (architectural pattern) cung cấp một tập các quyết định thiết kế liên quan đến các thành phần và kết nối với những đặc trưng cụ thể cho một lớp hệ phần mềm. Sự phân biệt

giữa mẫu và kiểu kiến trúc không phải bao giờ cũng rõ ràng. Một số đặc trưng sau đây có thể giúp ta hiểu hơn sự khác biệt này:

- *Phạm vi*: Kiểu kiến trúc áp dụng vào một ngữ cảnh phát triển như GUI, trong khi mẫu kiến trúc áp dụng vào một bài toán thiết kế đặc biệt như logic nghiệp vụ của hệ thống phải tách biệt với quản trị dữ liệu.
- *Trừu tượng hóa*: Kiểu giúp ràng buộc các quyết định thiết kế kiến trúc về hệ thống và khá trừu tượng nên đòi hỏi sự thể hiện của con người. Trong khi đó, mẫu là phần kiến trúc đã được tham số hóa và có thể xem như một mẫu thiết kế cụ thể.
- *Quan hệ*: Các mẫu không thể sử dụng ngay được mà phải được tham số hóa cho ngữ cảnh của bài toán cụ thể. Nghĩa là một mẫu cũng có thể áp dụng được cho những hệ thống với nhiều kiểu khác nhau. Ngược lại, một hệ thống được thiết kế theo một kiểu nào đó có thể sử dụng nhiều mẫu.

Ví dụ: Mẫu kiến trúc Hệ phân tán 3-tầng đã được sử dụng rộng rãi hiện nay cho các hệ thống phục vụ nghiên cứu khoa học (chẩn đoán ung thư, thiên văn, địa lý, dự đoán thời tiết..), các hệ thống dịch vụ ngân hàng, thương mại điện tử, đặt chỗ du lịch, chăm sóc y tế...Các hệ thống này có thể xử lý, lưu trữ, truy xuất một khối lượng lớn dữ liệu những người dùng khác nhau (Hình 1.1).



Hình 1.1: Biểu diễn kiến trúc 3 tầng [18]

Các hệ thống này gồm 3 tầng:

- Tầng client hay trình diễn chứa chức năng cần thiết để người dùng truy nhập dịch vụ hệ thống. Tầng này chứa giao diện GUI của hệ thống và có thể lấy dữ liệu và thực thi một số xử lý nhỏ. Tầng client thông thường được triển khai trên các máy với khả năng tính toán và lưu trữ hạn chế như máy để bàn hay các thiết bị cầm tay, di động...
- Tầng ứng dụng (application) hay logic nghiệp vụ hay còn gọi là tầng trung gian chứa chức năng chính của ứng dụng. Tầng này chịu trách nhiệm đáp ứng các yêu cầu dịch vụ hay xử lý yêu cầu từ tầng client và truy nhập, xử lý dữ liệu trong tầng dữ liệu. Tầng này được triển khai trên tập các máy chủ mạnh và lưu trữ lớn.

- Tầng dữ liệu (data) chứa chức năng lưu trữ và truy nhập dữ liệu cuả ứng dụng. Tầng này chứa cơ sở dữ liệu lớn và có khả năng cung cấp nhiều yêu cầu truy nhập dữ liệu song song.

Tương tác giữa các tầng về nguyên lý là tuân theo khuôn dạng "yêu cầu – đáp ứng" nhưng không quy định chi tiết những dạng tương tác này. Nó có thể cài đặt theo kiểu một yêu cầu - một đáp ứng hay nhiều yêu cầu - một đáp ứng và cũng có thể một yêu cầu – nhiều đáp ứng hay đáp ứng dựa trên cập nhật yêu cầu theo chu kỳ.

1.3.7 Mô hình kiến trúc

Kiến trúc của một hệ phần mềm có thể được thể hiện bởi *mô hình kiến trúc* (architectural model) bằng cách sử dụng những ký hiệu mô hình hóa. Mô hình kiến trúc là sản phẩm thể hiện một số hay tất cả những quyết định thiết kế tạo nên kiến trúc của hệ thống. Tập ký hiệu mô hình kiến trúc được gọi là ngôn ngữ mô tả kiến trúc. Ngôn ngữ này có thể là văn bản, đồ họa, biểu đồ… Ví dụ, kiến trúc hệ 3 tầng có thể thể hiện bởi hai tập ký hiệu mô hình hóa khác nhau (Hình 1.1). Mô hình hóa kiến trúc sẽ được trình bày chi tiết trong Chương 3.

1.4 KẾT LUẬN

Chương này đã trình bày một số khái niệm cơ bản liên quan đến kiến trúc phần mềm. Trước hết tài liệu đã đề cập đến khái niệm độ phức tạp phần mềm. Nó là cơ sở cho lý do tại sao kiến trúc phần mềm trở thành chủ đề quan trong trong phát triển phần mềm. Khái niệm kiến trúc phần mềm cùng những khaí niệm liên quan đến kiến trúc như thành phần, kết nối, mẫu thiết kế, mẫu kiến trúc cũng đã được đề cập một cách khái quát. Các khái niệm như vậy sẽ là cơ sở cho xây dựng kiến trúc phần mềm và sẽ được làm sáng tỏ và chi tiết hơn trong các chương sau.

BÀI TẬP

1. Trong một hệ hướng đối tượng, độ phức tạp phần mềm xác định dựa trên biến, phương thức, đối tượng, gói hay cả hệ thống. Trình bày những đặc trưng và độ đo phần mềm ở các mức độ khác nhau này. Tham khảo

   http://worldcomp-proceedings.com/proc/p2015/SER6097.pdf

2. Một hệ thống thương mại điện tử như eBay, Amazon có cấu trúc gồm các thành phần, kết nối, cấu hình... Hãy khảo sát các hệ trên mạng và chỉ ra các yếu tố của các hệ thống này.
3. Hãy trình bày các đặc trưng của các kiểu kết nối và cách cài đặt các kết nối.
4. Hãy liệt kê các kiểu kết nối (giao thức) có thể có giữa các tầng trong kiến trúc 3 tầng. Hãy trình bày thiết kế và cài đặt một số giao thức đó.
5. Trình bày phân loại các mẫu thiết kế và liệt kê tên của các mẫu thiết kế.
6. Phân biệt các khái niệm mẫu thiết kế, cấu trúc, thiết kế, kiến trúc
7. Chỉ ra một số mẫu thiết kế đã được xây dựng trong các thư viện của Java. Các mẫu này có thể được sử dụng để cài đặt những chức năng nào trong các hệ thống như thương mại điện tử, quản lý đăng ký học tín chỉ, quản lý thư viện?

8. Trình bày các mẫu thiết kế DAO và ví dụ code java tương ứng với lớp Book trong Hệ quản lý sách online. Các phương thức có thể là addBook(), updateBook(), deleteBook().
9. Sử dụng Câu 8 và biểu diễn mẫu kiến trúc MVC với các gói tương ứng.
10. Sử dụng công cụ VP để xây dựng các biểu đồ gói, triển khai của các hệ thống như thương mại điện tử, quản lý đăng ký học tín chỉ, quản lý thư viện…

CHƯƠNG 2: THIẾT KẾ KIẾN TRÚC VÀ CÁC KIỂU KIẾN TRÚC

Nội dung của chương này tập trung xem xét một số chủ đề sau đây:

- Tiến trình thiết kế kiến trúc
- Các nguyên lý thiết kế
- Các kiểu kiến trúc

2.1 TIẾN TRÌNH THIẾT KẾ KIẾN TRÚC

Phát triển phần mềm dù theo phương pháp luận nào cũng thường bao gồm các công đoạn xác định yêu cầu, phân tích, thiết kế, cài đặt-tích hợp và kiểm chứng. Thiết kế thường lại được chia thành hai giai đoạn là thiết kế kiến trúc và thiết kế chi tiết. Các phương pháp luận ngày nay như UP (Unified Process) và các phương pháp tiến hóa của nó đều dựa trên *phát triển lặp và tăng dần*. Vì vậy, giống như các công đoạn khác, thiết kế kiến trúc cần được tiến hành ngay từ khi khởi đầu dự án. Các hoạt động thiết kế kiến trúc xem như một quá trình lặp của một dãy các bước và mở rộng thêm khi có thông tin mới được phát hiện trong các pha trước đó. Tiến trình thiết kế kiến trúc thô ng thường gồm các bước cơ bản sau đây ([1][3]):

1. *Hiểu được vấn đề cần giải quyết*
2. *Xác định các phần tử của thiết kế và quan hệ của chúng*
3. *Đánh giá thiết kế kiến trúc*
4. *Biến đổi thiết kế kiến trúc*

Bước đầu tiên được cho là then chốt vì nếu không hiểu rõ bài toán đặt ra sẽ không thể đưa ra giải pháp hữu hiệu. Bước thứ hai nhằm chỉ ra các phần tử thiết kế và sự phụ thuộc của chúng. Bước thứ ba liên quan đánh giá kiến trúc theo yêu cầu thuộc tính của chất lượng kiến trúc.

Rõ ràng, chức năng của một ứng dụng không thể kiểm chứng được từ quá trình phân rã cấu trúc. Tuy nhiên, nhiều thuộc tính chất lượng khác có thể đánh giá được như xem xét kiểu thiết kế hay cài đặt bản mẫu cho tương tác giữa các thành phần quan trọng. Bước thứ tư liên quan đến áp dụng các thao tác thiết kế để biến đổi thiết kế kiến trúc thành một thiết kế mới với những yêu cầu thuộc tính chất lượng tốt hơn thiết kế trước. Bước này có thể lặp lại nhiều lần. Tiến trình thiết kế kiến trúc được xem là sự mở rộng của tiến trình thiết kế kỹ thuật thông thường.

Mục đích của thiết kế kiến trúc là tập trung vào phân rã hệ thống thành các thành phần và tương tác giữa các thành phần sao cho thỏa mãn các yêu cầu chức năng và phi chức năng được đặt ra. Một hệ phần mềm có thể xem như một phân cấp của các quyết định thiết kế trong đó mỗi mức phân cấp có một tập luật thiết kế để ràng buộc hay kết nối các thành phần ở mức đó.

Ví dụ trong kiến trúc kiểu Client/server, các quy luật thiết kế có thể liên quan đến đặc tả giao diện giữa client và server. Mức phân cấp kế tiếp sẽ mô tả mỗi client hay mỗi server như là một tập các thành phần tương tác. Mỗi nhánh phân cấp có những luật thiết kế riêng mà các

nhánh khác không nhìn thấy được. Như vậy phân rã của client là ẩn dấu đối với server và ngược lại. Đầu ra của tiến trình thiết kế kiến trúc là mô tả cấu trúc.

2.1.1 Hiểu được vấn đề cần giải quyết

Nhiều dự án hay sản phẩm phần mềm được cho là không thành công vì nó không giải quyết được vấn đề thực sự trong nghiệp vụ khách hàng hoặc không thỏa mãn lợi tức đầu tư mà phần mềm đem lại. Câu hỏi đặt ra là tại sao phải đầu tư một giải pháp mới mà chẳng đem lại lợi ích gì so với giải pháp cũ. Đa số kỹ sư phần mềm đều tập trung vào các các giải pháp kỹ thuật hơn là giải quyết vấn đề cơ bản mà nhà đầu tư yêu cầu mà thường được gọi là *bẫy cài đặt*. Để tránh cái bẫy như vậy, chúng ta nên tự đặt vấn đề lặp đi lặp lại là các quyết định thiết kế đưa ra nhằm giải quyết những vấn đề gì.

2.1.2 Xác định các phần tử thiết kế và quan hệ của chúng

Trong bước này, chúng ta xác lập một phân rã ban đầu của hệ thống dựa trên các yêu cầu chức năng bao gồm các phần tử thiết kế và các phụ thuộc giữa chúng. Kiến trúc của một hệ phần mềm thường được biểu diễn như một tập các module hay hệ thống con mà ta gọi là thành phần.

Theo cách phân cấp, các chương trình trong C hay Java cũng có thể xem như cấu trúc gồm những thành phần mức thấp hơn. Các ngôn ngữ lập trình xác định một tập các thành phần theo các từ khóa và cú pháp của ngôn ngữ đó. Khai báo số nguyên và gán giá trị cho nó cũng có thể xem như thành phần gồm một tập các thành phần mức thấp hơn là những lệnh và mẫu dữ liệu. Điều này tương tự như chip trong máy tính gồm những thành phần là cổng và cổng lại gồm những thành phần mức thấp hơn là các mạch. Như vậy, kiến trúc có thể xem như cấu trúc phân cấp của phần mềm.

Việc chỉ ra các phần tử thiết kế và quan hệ của chúng có thể phân chia hơn nữa thành các bước sau đây:

- Xác định ngữ cảnh hệ thống.
- Xác định các module
- Mô tả các thành phần và kết nối

Hai bước đầu sử dụng kỹ thuật thiết kế trừu tượng cho phép chúng ta tập trung vào mức chi tiết cụ thể mà không phải tập trung vào mọi khía cạnh của hệ thống. Bước thứ ba liên quan đến việc mô tả các thể hiện của các kiểu module trong cấu hình thời gian chạy.

2.1.3 Đánh giá thiết kế kiến trúc

Thiết kế kiến trúc có thể được đánh giá theo những cách khác nhau. Để đánh giá một thiết kế, chúng ta phải hình dung rõ ràng những yêu cầu thuộc tính chất lượng ảnh hưởng bởi thiết kế. Những phán đoán kiểu như "thiết kế phải chạy nhanh" hay "hệ thống phải dễ dàng sửa đổi hay hiệu chỉnh cho nhu cầu mới" chỉ là những yêu cầu ban đầu mà không đủ để đánh giá thiết kế. Đánh giá kiến trúc không phải là đánh giá chính sản phẩm phần mềm mà là đánh giá mô tả về mặt thiết kế kiến trúc của hệ thống. Đánh giá kiến trúc là nhằm đo chất lượng thiết kế sau

khi nó được tạo ra. Không phải bao giờ cũng có thể đánh giá thiết kế để hiểu thuộc tính chất lượng như khả năng chỉnh sửa và hiệu năng.

Thông thường để tạo một hệ thống mà sau này có thể chỉnh sửa được là phân rã hệ thống thành các tầng. Tuy nhiên, các tầng có thể dẫn đến khó hiểu về mặt tính toán. Cho đến nay việc đánh giá kiến trúc phần mềm là dựa vào kiểu không hình thức trong đó các bên liên quan dự án gặp nhau và đưa ra ý kiến của mình.

2.1.4 Biến đổi thiết kế

Bước này được tiến hành sau đánh giá thiết kế. Nếu thiết kế kiến trúc không thỏa mãn đầy đủ yêu cầu thuộc tính chất lượng, thì nó phải được thay đổi cho đến khi thỏa mãn yêu cầu. Như vậy, bắt đầu từ thiết kế hiện thời chúng ta thực hiện phép biến đổi bằng cách sử dụng các toán tử thiết kế, kiểu và mẫu thiết kế. Các toán tử thiết kế cơ bản bao gồm: phân rã hệ thống thành các thành phần; nén một số thành phần thành một thực thể để cải tiến hiệu năng; trừu tượng hóa thành phần để cải tiến khả năng chỉnh sửa và thích nghi; chia sẻ một thành phần với các thành phần khác để cải tiến khả năng tích hợp, chỉnh sửa.

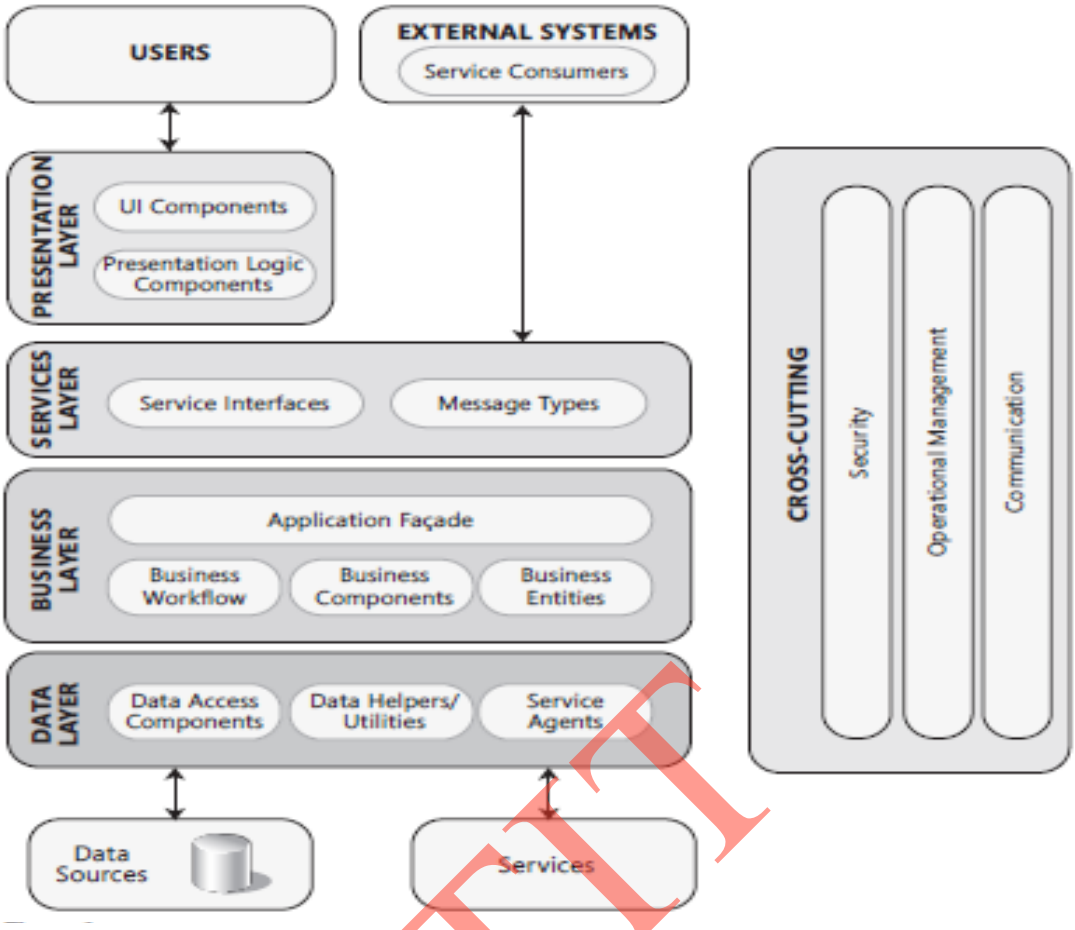
2.2 CÁC NGUYÊN LÝ THIẾT KẾ KIẾN TRÚC

Những nghiên cứu về kiến trúc cho rằng thiết kế phải tiến hóa theo thời gian vì chúng ta không thể biết được mọi điều trong khi tiến hành kiến trúc hệ thống. Do đó, thiết kế cần phải được tiến hóa suốt trong giai đoạn cài đặt ứng dụng vì khi đó chúng ta biết nhiều hơn qua kiểm chứng các yêu cầu thế giới thực. Xây dựng kiến trúc với sự tiến hóa sẽ làm cho nó có thể thích nghi được với yêu cầu chưa được biết đầy đủ trong giai đoạn đầu của quá trình thiết kế.

2.2.1 Kiến trúc phần mềm tổng quát

Một kiến trúc phần mềm thường được mô tả như một cấu trúc hay tổ chức của một hệ thống. Thuật ngữ hệ thống được hiểu như là một tập các thành phần xác lập một chức năng hay một tập các chức năng nào đó. Như vậy, kiến trúc phần mềm tập trung vào tổ chức các thành phần hay nhóm các thành phần thành những cụm có những mục tiêu khác nhau. Kiến trúc tổng quát (Hình 2.1) với các thành phần được nhóm thành những tầng có mục tiêu khác nhau [8]:

- Tầng trình diễn (presentation layer): Bao gồm các các thành phần giao diện người dùng (UI: User Interface) và cac thành phần logic trình diễn (Presenttation logic)
- Tầng dịch vụ (service layer): Bao gồm các giao diện dịch vụ và các kiểu dịch vụ.
- Tầng nghiệp vụ (business layer): Bao gồm các thành phần, thực thể nghiệp vụ và luồng hoạt động nghiệp vụ.
- Tầng dữ liệu (data layer): Bao gồm các thành phần truy nhập dữ liệu, các tiện ích dữ liệu và accs tác nhân dịch vụ.
- Tầng kết nối (cross cutting): Bao gồm an toàn, truyền thông và quản lý điều hành



Hình 2.1: Kiến trúc hệ phần mềm tổng quát [8]

Chương này sẽ xem xét những thao tác thiết kế tổng quát của thiết kế kiến trúc phần mềm và cách sử dụng chúng để phân rã hệ thống thành các thành phần và kết nối để đạt được thuộc tính chất lượng mong muốn. Các thao tác thiết kế thông thường bao gồm: phân rã, gộp, nén, trừu tượng hóa và chia sẻ tài nguyên. Các thao tác này lại được xem xét từ mức độ kiến trúc của thiết kế.

2.2.2 Nguyên lý thiết kế cơ bản

Khi bắt đầu thiết kế, cần phải chú ý đến những nguyên lý cơ bản nhằm giúp tạo ra một kiến trúc có thể tối thiểu hóa được chi phí cũng như yêu cầu bảo trì và tăng cường được khả năng mở rộng. Sau đây là một số nguyên lý cơ bản [8]:

- *Nguyên lý phân rã các chức năng:* Phân chia ứng dụng thành những nhóm sao cho tối thiểu hóa được sự chồng lắp các chức năng. Yếu tố quan trọng là giảm thiểu những điểm tương tác để đạt được *kết hợp cao* (high cohesion) và *kết dính thấp* (low coupling). Tuy nhiên, việc phân rã các chức năng không hợp lý có thể dẫn đến sự kết dính cao giữa các chức năng dù các chức năng liên quan không thực sự chồng lắp.
- *Nguyên lý đơn nhiệm:* Mỗi thành phần hay môdun nên chịu tránh nhiệm cho chỉ một chức năng hay một hợp chức năng có liên kết nhau.

- *Nguyên lý hiểu biết tối thiểu:* Một thành phần hay một đối tượng không nên biết về chi tiết bên trong của các thành phần hay các đối tượng khác.
- *Nguyên lý không tự lặp lại:* Chỉ nên đặc tả ý định tại một vị trí. Ví dụ, theo thiết kế ứng dụng, một chức năng cụ thể nên được cài đặt chỉ trong một thành phần. Nghĩa là chức năng đó không nên cài trong những thành phần nào khác.
- *Nguyên lý cực tiểu hóa thiết kế:* Chỉ thiết kế những gì cần thiết. Nguyên lý này đôi khi còn gọi là YAGNI (you ain't gonna need it). Nếu yêu cầu ứng dụng không rõ ràng hay nếu thiết kế cần phải tiến hóa theo thời gian thì tránh đưa ra thiết kế lớn.

Khi thiết kế một hệ thống hay một ứng dụng, kiến trúc sư phần mềm phải cực tiểu hóa độ phức tạp bằng cách tách thiết kế thành những quan tâm khác nhau. Ví dụ, các phần giao diện người dùng, xử lý nghiệp vụ và truy nhập dữ liệu là những thể hiện quan tâm khác nhau. Điều này được thể hiện trong mẫu thiết kế MVC (Model, View, Control) thường gặp.

Trong mỗi lĩnh vực ứng dụng, các thành phần nên tập trung vào một khía cạnh cụ thể và không nên trộn mã nguồn từ những quan tâm khác nhau. Ví dụ, các thành phần xử lý giao diện không nên chứa mã truy nhập trực tiếp nguồn dữ liệu mà nên sử dụng thành phần nghiệp vụ hay thành phần truy nhập dữ liệu để truy xuất dữ liệu.

2.3 CÁC KIỂU KIẾN TRÚC

Phần này dành trình bày những *kiểu kiến trúc* (architectural style) mà chúng được xem như những mẫu và nguyên lý thông dụng trong thiết kế các ứng dụng hiện nay. Với mỗi kiểu kiến trúc, tài liệu sẽ trình bày những nguyên lý chủ yếu, những ưu nhược điểm của từng kiểu kiến trúc nhằm giúp chọn lựa được kiểu kiến trúc phù hợp cho từng ứng dụng.

Việc hiểu biết về các kiểu kiến trúc đem lại nhiều lợi ích trong đó quan trọng nhất là cung cấp một ngôn ngữ chung cho các lớp hệ thống khác nhau. Mỗi kiểu mô tả các khía cạnh khác nhau trong ứng dụng nên thông thường mỗi ứng dụng cụ thể sử dụng một tổ hợp nhiều kiểu khác nhau. Bảng dưới đây liệt kê các khía cạnh chính của các kiểu kiến trúc tương ứng.

Phân loại	Kiểu kiến trúc
Giao tiếp	Kiến trúc hướng dịch vụ, Kiến trúc bus thông điệp
Triển khai	Client/Server, N-tầng, 3-Tầng
Miền	Thiết kế hướng miền
Cấu trúc	Dựa vào thành phần, Hướng đối tượng, Kiến trúc theo tầng

2.3.1 Kiểu kiến trúc Client/Server

Kiến trúc client/server mô tả hệ thống phân tán bao gồm các hệ thống máy chủ, các máy khách và các mạng lưới kết nối. Dạng đơn giản nhất của hệ thống client/server bao gồm một ứng dụng máy chủ được truy cập trực tiếp bởi nhiều máy khách như kiểu kiến trúc 2 tầng (2-tiers). Trước đây, kiến trúc client/server được biểu thị trong một ứng dụng có giao diện đồ họa được giao tiếp với server dữ liệu bao gồm rất nhiều logic nghiệp vụ dưới dạng thủ tục hoặc với một file trên server. Tuy nhiên, kiểu kiến trúc này lại mô tả mối quan hệ giữa một client với một hoặc nhiều server, trong đó client khởi tạo nhiều yêu cầu và gửi đến server sau đó đợi

trả lời từ server và xử lí thông tin nhận được. Server có thể gửi câu trả lời bằng cách sử dụng một loạt các giao thức và định dạng dữ liệu để truyền thông tin cho client.

Ngày nay, nhiều ứng dụng kiểu kiến trúc client/server bao gồm: Các chương trình dựa trên trình duyệt Web chạy trên Internet hoặc một mạng nội bộ như Microsoft Windows - ứng dụng hệ điều hành dựa trên truy cập các dịch vụ dữ liệu trên mạng; ứng dụng truy cập các ngân hàng dữ liệu từ xa (ví dụ như đọc e-mail, FTP khách hàng và các công cụ truy vấn cơ sở dữ liệu); các công cụ và tiện ích thao tác hệ thống từ xa (ví dụ như các công cụ quản lý hệ thống và các công cụ giám sát mạng). Các biến thể từ kiểu kiến trúc client/server bao gồm:

- Các hệ hàng đợi-client: Cách tiếp cận này cho phép client giao tiếp với client khác thông qua một hàng đợi dựa trên server. Client có thể đọc dữ liệu từ server và gửi dữ liệu đến một server và sử dụng một hàng đợi để lưu trữ dữ liệu. Điều này cho phép client phân phối và đồng bộ hóa các tập tin và thông tin. Điều này đôi khi được gọi là một kiến trúc hàng đợi thụ động.
- Hệ Peer-to-Peer (P2P) được phát triển từ kiểu hệ hàng đợi - client. Kiểu P2P cho phép client và server trao đổi vai trò của chúng trong quá trình phân phối và đồng bộ các file và thông tin thông qua nhiều client. Kiến trúc này mở rộng kiểu client/server thông qua việc có nhiều phản hồi cho các yêu cầu, chia sẻ dữ liệu, khai thác tài nguyên, và khả năng phục hồi để loại bỏ các bên.
- Ứng dụng server: là một kiểu kiến trúc đặc biệt trong đó server lưu trữ và thực hiện các ứng dụng và dịch vụ mà một client truy cập thông qua một trình duyệt hoặc một client được cài đặt phần mềm đặc biệt. Ví dụ như một client thực hiện một ứng dụng chạy trên server thông qua nền tảng như dịch vụ đầu cuối.

Những ưu/nhược điểm của kiểu kiến trúc client/server

- Có tính bảo mật cao: Tất cả dữ liệu được lưu trên server, nơi thường có sự quản lí, an toàn cao hơn các máy client.
- Truy cập dữ liệu tập trung: Bởi vì dữ liệu chỉ được lưu trữ trên server nên việc truy cập và cập nhật dữ liệu sẽ nằm dưới quyền người quản trị.
- Dễ dàng bảo trì: Vai trò và trách nhiệm của một hệ thống máy tính được phân bố trong một số server được biết đến với nhau thông qua một mạng lưới. Điều này đảm bảo cho các máy client không bị ảnh hưởng khi một máy server bị sửa chữa, nâng cấp hoặc di chuyển.
- Tuy nhiên, kiểu kiến trúc này cũng có một số bất tiện như việc xây dựng ứng dụng có dữ liệu và logic nghiệp vụ được kết hợp chặt chẽ trên các server. Điều này có thể tác động tiêu cực đến việc mở rộng hệ thống và dẫn đến sự phụ thuộc vào một server trung tâm, ảnh hưởng đến độ tin cậy của hệ thống. Để giải quyết những vấn đề này, kiểu kiến trúc client-server đã phát triển thành kiểu kiến trúc 3-tầng (hoặc N-tầng) sau này.

Hướng dẫn sử dụng

Trong quá trình thiết kế chúng ta cần cân nhắc xử dụng kiểu kiến trúc client/server nếu ứng dụng đang xây dựng dựa trên máy chủ và hỗ trợ nhiều khách hàng. Ví dụ, khi tạo ra các ứng

dụng web dựa trên một trình duyệt Web, việc xử lý quy trình nghiệp vụ sẽ được sử dụng bởi những người trong cùng một tổ chức, hoặc tạo ra các dịch vụ cho các ứng dụng khác sử dụng. Kiểu kiến trúc client/server cũng thích hợp cho rất nhiều kiểu mạng nếu muốn tập trung lưu trữ hoặc sao lưu dữ liệu và các chức năng quản lý, hoặc khi ứng dụng cần phải hỗ trợ các loại client và thiết bị khác nhau.

2.3.2 Kiểu kiến trúc dựa vào thành phần

Kiểu kiến trúc dựa vào thành phần thể hiện phương pháp tiếp cận thành phần cho thiết kế hệ thống. Nó tập trung vào phân rã kiến trúc thành các thành phần chức năng riêng biệt hoặc các thành phần tiếp xúc với giao diện chuẩn qua kết nối. Cách tiếp cận như vậy đem lại một mức độ trừu tượng cao hơn so với phương pháp thiết kế hướng đối tượng. Nguyên tắc chính của kiểu kiến trúc dựa vào thành phần là việc sử dụng các thành phần với các tính chất:

- *Khả năng tái sử dụng:* các thành phần thường được thiết kế để tái sử dụng trong những hoàn cảnh khác nhau với những ứng dụng khác nhau. Tuy nhiên, cũng có một số thành phần có thể chỉ được thiết kế để làm một công việc cụ thể.
- *Khả năng thay thế được:* Các thành phần có thể được thay thế bằng các thành phần tương tự.
- *Không có hoàn cảnh cụ thể:* Các thành phần hoạt động dưới những hoàn cảnh và môi trường khác nhau. Các thông tin cụ thể như dữ liệu trạng thái nên được chuyển về thành phần thay vì được bao gồm hoặc truy cập bởi thành phần này.
- *Khả năng mở rộng:* một thành phần có thể được mở rộng từ một thành phần cũ để cung cấp thêm các trạng thái và hoạt động mới.
- *Được đóng gói:* Thành phần tiếp xúc với giao diện cho phép người gọi sử dụng chức năng của nó và không tiết lộ chi tiết về tiến trình nội bộ hoặc bất kỳ một biến hoặc trạng thái nội bộ.
- *Tính độc lập:* Các thành phần được thiết kế để chỉ phụ thuộc tối thiểu vào các thành phần khác. Do đó, các thành phần có thể được triển khai trong bất kỳ môi trường thích hợp mà không ảnh hưởng đến các thành phần hoặc các hệ thống khác.

Các loại thành phần phổ biến được sử dụng trong các ứng dụng bao gồm các thành phần giao diện người dùng như hệ thống điều khiển với nút bấm; các thành phần trợ giúp hay thành phần tiện ích tiếp xúc với một nhóm cụ thể các chức năng được sử dụng trong các thành phần khác. Các loại thành phần phổ biến khác là những thành phần có nguồn tài nguyên riêng biệt, không được truy cập thường xuyên; các thành phần hàng đợi mà các phương thức có thể được thực hiện không đồng bộ bằng cách sử dụng thông điệp xếp hàng để lưu trữ hoặc chuyển tiếp.

Các thành phần phụ thuộc vào một cơ chế trên nền tảng cung cấp môi trường thực hiện cho chúng thường được gọi là kiến trúc thành phần. Ví dụ như mô hình đối tượng thành phần COM và thành phần phân tán DCOM trong .NET; CORBA hay EJB trong J2EE trên các nền tảng khác. Kiến trúc thành phần quản lý các cơ chế định vị các thành phần và giao diện của chúng, thông qua các thông điệp hoặc lệnh giữa các thành phần, và trong một số trường hợp duy trì trạng thái.

Tuy nhiên, các thuật ngữ trong thành phần thường được sử dụng theo nghĩa cơ bản nhiều hơn là một phần cấu thành, phần tử, hoặc thành phần. Khung làm việc .NET của Microsoft, Khung làm việc J2EE của Java hỗ trợ xây dựng ứng dụng bằng cách sử dụng cách tiếp cận dựa trên thành phần.

Những ưu điểm của kiểu kiến trúc dựa trên thành phần

Sau đây là những lợi ích chính của kiểu kiến trúc dựa trên thành phần:

- *Dễ triển khai:* Các phiên bản tương thích có sẵn nên có thể thay thế phiên bản hiện tại mà không ảnh hưởng đến các thành phần khác hoặc hệ thống nói chung.
- *Chi phí giảm:* Việc sử dụng các thành phần của bên thứ ba cho phép dàn trải chi phí phát triển và bảo trì.
- *Dễ dàng phát triển:* Các thành phần thực hiện các giao diện nổi tiếng cung cấp chức năng đã được xác định nên cho phép phát triển mà không ảnh hưởng các bộ phận khác của hệ thống.
- *Tái sử dụng:* Việc sử dụng các thành phần tái sử dụng có nghĩa là chúng có thể được sử dụng để giảm chi phí phát triển và chi phí bảo trì qua nhiều ứng dụng hoặc hệ thống.
- *Giảm thiểu sự phức tạp kỹ thuật:* Thành phần giảm thiểu sự phức tạp thông qua việc sử dụng vật chứa thành phần và dịch vụ của nó. Dịch vụ thành phần bao gồm kích hoạt thành phần, phương thức xếp hàng, các sự kiện, và các giao dịch.

Hướng dẫn sử dụng

Trong thiết kế, ta chọn kiểu kiến trúc dựa trên thành phần nếu có các thành phần phù hợp hoặc có thể có được các thành phần phù hợp từ nhà cung cấp bên thứ ba. Khi đó ứng dụng sẽ chủ yếu thực hiện chức năng thủ tục và với dữ liệu ít hoặc không có đầu vào; hoặc khi muốn có thể kết hợp các thành phần viết bằng ngôn ngữ mã khác nhau. Ta cũng có thể sử dụng kiểu này khi muốn tạo ra một kiến trúc hỗn hợp, cho phép thay thế và cập nhật dễ dàng các thành phần riêng rẽ.

2.3.3 Kiểu kiến trúc hướng miền

Thiết kế hướng miền là một cách tiếp cận hướng đối tượng để thiết kế phần mềm dựa trên các hoạt động nghiệp vụ với các thuộc tính, hành vi và quan hệ giữa chúng. Nó cho phép các hệ thống phần mềm thực hiện những chức năng cơ bản bằng cách xác định một mô hình miền thể hiện trong ngôn ngữ của các chuyên gia trong lĩnh vực nghiệp vụ. Mô hình miền có thể được xem như là một khuôn khổ để hợp lý hóa các giải pháp có thể có được.

Để áp dụng thiết kế dựa trên mô hình miền, chúng ta phải có một hiểu biết khá rõ về lĩnh vực nghiệp vụ mà mình muốn mô hình, hoặc có kỹ năng cũng như kiến thức nghiệp vụ liên quan. Đội ngũ phát triển sẽ thường xuyên làm việc với các chuyên gia trong lĩnh vực nghiệp vụ để mô hình miền cho đúng. Kiến trúc sư, nhà phát triển và chuyên gia thường xuất phát từ những môi trường khác nhau nên ngôn ngữ cũng như yêu cầu của họ khi mô tả các mục tiêu, thiết kế thường không giống nhau. Tuy nhiên, trong thiết kế hướng miền, các nhóm cần thống

nhất chỉ sử dụng một ngôn ngữ tập trung vào lĩnh vực nghiệp vụ hơn là các thuật ngữ kỹ thuật.

Cốt lõi của bất kỳ phần mềm nào cũng là mô hình miền. Việc tạo ra một ngôn ngữ chung không chỉ là để thể hiện thông tin từ các chuyên gia lĩnh vực đó mà còn là để áp dụng nó. Những vấn đề truyền thông trong nhóm phát triển không phải do hiểu lầm về ngôn ngữ của miền, mà thực sự là do ngôn ngữ để mô hình bản thân miền đó không rõ ràng. Các tiến trình thiết kế hướng miền nhắm đến mục tiêu không chỉ đưa các ngôn ngữ mà còn cải thiện và tinh chỉnh ngôn ngữ của miền. Điều này sẽ mang lại lợi ích cho phần mềm được xây dựng, vì mô hình là phép chiếu trực tiếp của ngôn ngữ miền.

Để giúp duy trì các mô hình như là một cấu trúc ngôn ngữ, chúng ta thường phải thực hiện rất nhiều cách phân rã và đóng gói trong mô hình miền. Do đó, một hệ thống dựa trên mô hình hướng miền sẽ khiến cho chi phí tăng cao. Việc thiết kế hướng miền đem lại nhiều lợi ích kỹ thuật như bảo trì, nhưng thường chỉ được áp dụng với các lĩnh vực phức tạp đòi hỏi có sự hiểu biết chung về miền.

Những ưu điểm của kiểu kiến trúc hướng miền

- *Truyền thông liên lạc:* Tất cả các thành viên trong các nhóm phát triển có thể sử dụng mô hình miền và các đối tượng được định nghĩa để truyền đạt kiến thức nghiệp vụ với một ngôn ngữ nghiệp vụ thông thường mà không cần các thuật ngữ kỹ thuật.
- *Mở rộng:* Mô hình miền thường là các mô-đun và linh hoạt, nó dễ dàng cập nhật và mở rộng khi các điều kiện và yêu cầu thay đổi.
- *Kiểm chứng:* Các đối tượng mô hình miền gắn kết nhau một cách lỏng lẻo nên chúng dễ dàng được kiểm định.

Hướng dẫn sử dụng

Nên sử dụng kiến trúc hướng miền nếu chúng ta phải xây dựng phần mềm cho một lĩnh vực phức tạp cũng như muốn cải thiện truyền thông và sự hiểu biết trong đội ngũ phát triển. Hoặc trong trường hợp chúng ta phải thể hiện các thiết kế của một ứng dụng trong một ngôn ngữ phổ biến mà tất cả các bên liên quan có thể hiểu được. Kiến trúc kiểu này cũng có thể là một phương pháp lý tưởng nếu bạn có kịch bản dữ liệu nghiệp vụ lớn và phức tạp mà khó có thể quản lý bằng cách sử dụng các kỹ thuật khác.

2.3.4 Kiểu kiến trúc phân tầng

Kiến trúc phân tầng nhằm các nhóm chức năng có liên quan trong một ứng dụng thành các tầng riêng biệt được xếp chồng lên nhau theo chiều dọc. Chức năng trong các tầng kết nối với nhau bởi vai trò hay trách nhiệm chung và khi đó thông tin liên lạc giữa các tầng là rõ ràng và liên kết lỏng lẻo. Việc phân tầng nếu áp dụng một cách đúng đắn sẽ giúp cho việc phân rã hệ phức tạp tốt hơn và sẽ hỗ trợ tính linh hoạt và bảo trì sau này. Kiểu kiến trúc phân tầng được mô tả như là một kim tự tháp ngược của việc tái sử dụng trong đó mỗi tầng tập hợp các nhiệm vụ và trừu tượng hóa tầng trực tiếp bên dưới nó. Với việc phân tầng nghiêm ngặt, các thành phần trong một tầng chỉ có thể tương tác với các thành phần trong chính tầng đó hoặc với các thành phần từ các tầng trực tiếp bên dưới nó.

Các tầng của một ứng dụng có thể nằm trên các máy tính vật lý như nhau hoặc có thể được phân phối trên các máy tính riêng biệt. Khi đó các thành phần trong mỗi tầng giao tiếp với các thành phần trong các tầng khác thông qua giao thức được xác lập rõ ràng. Ví dụ, một thiết kế ứng dụng Web điển hình bao gồm tầng trình diễn có chức năng liên quan đến giao diện người dùng, một tầng nghiệp vụ để xử lí quy tắc nghiệp vụ, và một tầng dữ liệu liên quan đến truy cập dữ liệu. Nguyên tắc chung cho các thiết kế dựa trên kiểu kiến trúc phân tầng bao gồm:

- *Trừu tượng:* Kiến trúc phân tầng thể hiện cái nhìn toàn bộ hệ thống đồng thời cũng cung cấp đầy đủ chi tiết để hiểu được vai trò và trách nhiệm của các tầng riêng biệt và mối quan hệ giữa chúng.
- *Đóng gói:* Không cần phải đưa ra giả định về các loại dữ liệu, phương thức và thuộc tính, hoặc thực thi trong quá trình thiết kế, các tính năng này không được tiếp xúc ở ranh giới tầng.
- *Xác định rõ ràng các tầng chức năng:* Việc tách biệt chức năng trong mỗi tầng là rõ ràng. Tầng trên cùng như các tầng trình diễn gửi lệnh đến tầng dưới, chẳng hạn như các tầng nghiệp vụ và dữ liệu, và có thể xử lí các sự kiện trong các tầng, cho phép dữ liệu chuyển lên và xuống giữa các tầng.
- *Kết dính cao:* Kiến trúc này xác định rõ ranh giới trách nhiệm cho từng tầng và đảm bảo rằng mỗi tầng có chức năng liên quan trực tiếp đến nhiệm vụ của tầng đó. Điều đó sẽ giúp tối đa hóa sự gắn kết bên trong tầng.
- *Tái sử dụng:* Tầng thấp hơn không có phụ thuộc vào tầng cao hơn, do đó nó có khả năng cho phép chúng có thể tái sử dụng trong các tình huống khác.
- *Kết nối lỏng lẻo:* Việc liên lạc thông tin giữa các tầng dựa trên khái niệm trừu tượng và các sự kiện để cung cấp kết nối lỏng lẻo giữa các tầng.

Ví dụ, kiểu kiến trúc này có thể áp dụng cho các hệ thống kế toán và quản lý khách hàng, các ứng dụng nghiệp vụ dựa trên web, máy tính để bàn, thiết bị cầm tay…

Mẫu thiết kế hỗ trợ các kiểu kiến trúc tầng phổ biến là mô hình MVC có 3 vai trò mô hình (Model), Khung nhìn (View) và Điều khiển (Controller). Mô hình thể hiện dữ liệu và có thể mô hình nguồn bao gồm các thực thể, luật nghiệp vụ; Khung nhìn thể hiện giao diện người dùng và Điều khiển xử lí các yêu cầu, thao tác với mô hình và thực hiện các lệnh:

- *Dựa trên sự kiện:* Các mô hình quan sát thường được sử dụng để cung cấp thông báo cho View khi dữ liệu thay đổi bởi Controller.
- *Phân cấp xử lý sự kiện:* Bộ điều khiển xử lý các sự kiện nhận được từ giao diện điều khiển trong các View.

MVC xem là một mẫu tách biệt phần trình diễn bao gồm một loạt các mô hình xử lý các tương tác của người dùng từ giao diện đến trình diễn và nghiệp vụ logic, các dữ liệu ứng dụng. Cách tiếp cận này cho phép các nhà thiết kế đồ họa tạo ra một giao diện người dùng, trong khi các nhà phát triển tạo ra các mã dẫn đến điều khiển. Việc chia các chức năng thành vai trò riêng biệt theo cách này cung cấp cách quản lý tiến trình phát triển tốt hơn.

Những ưu điểm của mô hình kiến trúc phân tầng

- *Trừu tượng:* phân tầng cho phép các thay đổi được thực hiện ở mức độ trừu tượng của các tầng.
- *Cô lập:* cho phép nâng cấp công nghệ để cô lập các tầng riêng biệt nhằm giảm thiểu rủi ro và giảm thiểu tác động đến toàn bộ hệ thống.
- *Quản lý:* tách mối quan tâm cốt lõi giúp xác định phụ thuộc, và tổ chức tốt mã nguồn.
- *Hiệu suất:* phân phối các lớp trên nhiều tầng vật lý có thể cải thiện khả năng mở rộng, khả năng chịu lỗi, nâng cao hiệu suất và tang cường vai trò tái sử dụng. Ví dụ, trong MVC, Controller thường có thể được tái sử dụng tương thích với những lần đọc khác để cung cấp một vai trò cụ thể hoặc một quan điểm người dùng tùy chỉnh trên cùng một dữ liệu và chức năng.
- *Dễ kiểm thử:* khả năng kiểm thử được tăng cường nhờ giao diện xác định giữa các tầng. Nghĩa là chúng ta có thể xây dựng các đối tượng giả bắt chước các hành vi của các đối tượng cụ thể như mô hình, điều khiển...

Hướng dẫn sử dụng

Kiểu kiến trúc phân tầng nên được sử dụng khi chúng ta có các tầng thích hợp để tái sử dụng trong các ứng dụng khác. Hơn nữa, khi chúng ta có một quy trình nghiệp vụ phù hợp thì áp dụng kiến trúc phần tầng để tách các tầng riêng biệt cho các nhóm phát triển khác nhau sẽ đem lại hiệu quả hơn. Các kiểu kiến trúc phân tầng cũng được sử dụng khi ứng dụng phải hỗ trợ các loại khách hàng cũng như các thiết bị khác nhau, hoặc nếu muốn thực hiện các quy tắc nghiệp vụ có cấu hình và quy trình phức tạp.

2.3.5 Kiểu kiến trúc bus thông điệp

Kiến trúc bus thông điệp mô tả các nguyên tắc sử dụng một hệ thống phần mềm có thể nhận và gửi thông điệp qua một hoặc nhiều kênh truyền thông. Đó là một kiểu thiết kế các ứng dụng mà tương tác giữa các ứng dụng được thực hiện thông qua thông điệp thường là không đồng bộ trên một bus thông thường. Những triển khai phổ biến nhất là sử dụng kiến trúc với một router gửi thông điệp hoặc một mô hình công bố/đăng ký và thường được thực hiện bằng cách sử dụng một hệ thống thông điệp theo kiểu hang đợi. Thường các triển khai cũng bao gồm các ứng dụng giao tiếp với lược đồ chung và cơ sở hạ tầng dùng chung cho việc gửi và nhận thông điệp. Một bus thông thường cung cấp khả năng xử lý:

- *Giao tiếp hướng thông điệp:* Tất cả các giao tiếp giữa các ứng dụng được dựa trên các thông điệp được biết đến như lược đồ.
- *Xử lý logic phức tạp:* Các lệnh phức tạp có thể được thực hiện bằng cách kết hợp nhiều lệnh nhỏ trong đó mỗi lệnh nhỏ xử lý một công việc cụ thể.
- *Chỉnh sửa cho xử lý logic:* Bởi vì các tương tác giữa bus dựa trên các lược đồ là câu lệnh chung nên có thể thêm hoặc bớt ứng dụng trong bus để làm thay đổi logic được sử dụng để xử lý thông điệp.
- *Kết hợp nhiều môi trường khác nhau:* Do việc sử dụng mô hình giao tiếp dựa trên thông điệp theo một chuẩn chung nên các ứng dụng có thể tương tác với nhau trong những môi trường khác nhau như Java, .NET hay C.

Thiết kế bus thông điệp đã từng được sử dụng để hỗ trợ cho việc xử lý phức tạp. Thiết kế này cung cấp một kiến trúc có khả năng lặp lại nên cho phép chúng ta thêm ứng dụng vào trong quá trình xử lý hoặc cải thiện khả năng mở rộng bằng cách thêm nhiều ứng dụng cho bus. Một số biến đổi của kiểu bus thông điệp bao gồm:

- ESB (Enterprise Service Bus): Dựa trên thiết kế MS, kiểu ESB sử dụng dịch vụ cho truyền thông giữa bus và các thành phần được gắn với bus. ESB thường cung cấp dịch vụ chuyển giao thông điệp từ định dạng này sang định dạng khác. Đồng thời cho phép client sử dụng các định dạng thông điệp khác nhau để giao tiếp với nhau.
- ISB (Internet Service Bus). Tương tự như ESB nhưng với ứng dụng là máy chủ trong mạng lưới thay vì trong một mạng. Ý tưởng cốt lõi của ISB là sử dụng URI và có các chính sách để điều khiển dò tìm ứng dụng và dịch vụ trong mạng lưới.

Những ưu điểm của kiểu trúc bus thông điệp

- Khả năng mở rộng: Các ứng dụng có thể được thêm vào hoặc bớt đi trong bus mà không gây ảnh hưởng đến các ứng dụng khác. Nhiều trường hợp của cùng một ứng dụng có thể được gắn vào bus để xử lý nhiều yêu cầu cùng một lúc.
- Ít phức tạp: độ phức tạp của các ứng dụng sẽ giảm đi vì mỗi ứng dụng chỉ cần biết cách giao tiếp với bus mà thôi.
- Linh hoạt: việc gộp các ứng dụng khiến cho việc xử lý ít phức tạp hơn hoặc các mô hình truyền thông giữa các ứng dụng có thể thay đổi dễ dàng để đáp ứng được sự thay đổi trong nghiệp vụ hoặc yêu cầu người dùng. Những thay đổi này dẫn tới thay đổi trong cấu hình hoặc tham số cho những yếu tố điều khiển quá trình dò tìm.
- Kết nối lỏng: miễn là các ứng dụng tiếp xúc với một giao diện phù hợp dựa trên thông báo của thông tin liên lạc với các bus. Do không có sự phụ thuộc vào các ứng dụng cụ thể nên cho phép thay đổi, cập nhật, và thay thế với hiển thị cùng một giao diện.
- Ứng dụng đơn giản: việc thực hiện bus giảm tính phức tạp cho cơ sở hạ tầng, mỗi ứng dụng cần hỗ trợ chỉ có một kết nối duy nhất với bus thông báo thay vì nhiều kết nối cho các ứng dụng khác.

Hướng dẫn sử dụng

Kiến trúc theo kiểu bus thông điệp nên được sử dụng nếu chúng ta có các ứng dụng hiện thời tương thích với nhau để thực hiện nhiệm vụ hoặc chúng ta muốn kết hợp nhiều nhiệm vụ vào một hoạt động đơn lẻ. Kiểu kiến trúc này cũng thích hợp khi cần thực hiện một ứng dụng đòi hỏi sự tương tác với các ứng dụng bên ngoài, hoặc các ứng dụng được lưu trữ trong các môi trường khác nhau.

2.3.6 Kiểu kiến trúc N-tầng

N-tầng hay 3-tầng là các kiến trúc kiểu triển khai. Kiểu này mô tả việc tách chức năng thành các phần theo cách tương tự như kiểu phân tầng, trong đó mỗi tầng có thể được đặt trên một máy tính vật lý riêng biệt. Kiến trúc ứng dụng N-tầng được đặc trưng bởi sự phân rã chức năng của các ứng dụng thành các thành phần dịch vụ, phân phối và triển khai. Nó cải thiện khả năng mở rộng, cung cấp tính sẵn có, quản lý và sử dụng tài nguyên. Mỗi tầng hoàn toàn độc lập với tất cả các tầng khác, trừ trường hợp các tầng ngay ở trên và dưới nó. Tầng thứ n

chỉ biết xử lý một yêu cầu từ tầng thứ n-1, làm thế nào để chuyển tiếp yêu cầu đó vào tầng thứ n+1 và làm thế nào để xử lý các kết quả của yêu cầu. Thông tin liên lạc giữa các tầng thường không đồng bộ để hỗ trợ khả năng mở rộng tốt hơn.

Kiến trúc N-tầng thường có ít nhất ba phần, mỗi phần nằm trên một máy chủ vật lý riêng biệt và chịu trách nhiệm cho các chức năng cụ thể. Khi sử dụng cách tiếp cận thiết kế phân tầng, một tầng được triển khai trên một tầng nếu có nhiều hơn một dịch vụ hay ứng dụng phụ thuộc vào các chức năng tiếp xúc của tầng.

Một ví dụ điển hình về kiểu kiến trúc N-tầng là ứng dụng Web tài chính. Do vấn đề quan trọng nhất là an toàn nên các lớp nghiệp vụ phải được triển khai tường lửa, trong đó lực lượng triển khai các tầng trình diễn nằm trên một tầng riêng trong mạng. Một ví dụ khác là một ứng dụng kết nối khách hàng, trong đó các tầng trình diễn được triển khai trên các máy khách và các tầng nghiệp vụ và tầng truy cập dữ liệu được triển khai trên một hoặc nhiều máy chủ.

Những ưu điểm của kiểu kiến trúc N-tầng

- *Bảo trì:* Bởi vì mỗi tầng độc lập với các lớp khác, nên cập nhật hoặc thay đổi không ảnh hưởng đến các ứng dụng như một toàn thể.
- *Khả năng mở rộng:* Vì các tầng được dựa trên việc triển khai tiếp tầng dưới nó nên nhân rộng ra một ứng dụng là đơn giản hơn.
- *Tính linh hoạt:* Bởi vì mỗi tầng có thể được quản lý hoặc thu nhỏ một cách độc lập, nên tính linh hoạt được tăng lên.
- *Sẵn sàng:* Các ứng dụng có thể khai thác kiến trúc mô-đun của hệ thống nên cho phép mở rộng các thành phần dễ dàng và làm tăng tính sẵn sàng.

Kiểu kiến trúc N-tầng hay 3-tầng nên áp dụng khi các yêu cầu xử lý của các tầng trong ứng dụng khác nhau hoặc nếu các yêu cầu an ninh của các tầng ứng dụng khác nhau. Ví dụ, các tầng trình diễn không nên lưu trữ dữ liệu nhạy cảm, trong khi điều này có thể được lưu trữ trong các tầng nghiệp vụ và dữ liệu. Kiểu kiến trúc N-tầng hay 3-tầng cũng thích hợp nếu chúng ta muốn chia sẻ nghiệp vụ logic giữa các ứng dụng và có đủ phần cứng để phân bổ số lượng yêu cầu của máy chủ cho mỗi tầng.

Kiến trúc ba tầng cũng có thể sử dụng cho phát triển một ứng dụng mạng nội bộ, trong đó tất cả các máy chủ được đặt trong mạng riêng; hoặc một ứng dụng Internet, với yêu cầu bảo mật không hạn chế việc triển khai các logic nghiệp vụ trên Web hoặc máy chủ ứng dụng. Chúng ta cũng có thể sử dụng kiến trúc ba tầng nếu yêu cầu bảo mật không cho phép logic nghiệp vụ được triển khai đến các mạng hoặc các ứng dụng.

2.3.7 Kiểu kiến trúc hướng đối tượng

Kiến trúc hướng đối tượng là một mô hình thiết kế dựa trên sự phân chia trách nhiệm một ứng dụng hoặc hệ thống cho các thành phần tái sử dụng. Một thiết kế hướng đối tượng xem như một hệ thống với các đối tượng cộng tác với nhau, thay vì một bộ các chương trình hoặc hướng dẫn thủ tục. Đối tượng là rời rạc, độc lập, và liên kết lỏng lẻo. Chúng giao tiếp thông qua các giao diện bằng cách gọi phương thức hoặc truy cập vào thuộc tính của các đối tượng khác. Các nguyên tắc cơ bản của kiểu kiến trúc hướng đối tượng là:

- *Trừu tượng:* Điều này cho phép chúng ta giảm độ phức tạp của hành động mà vẫn giữ được các đặc điểm cơ bản của hành động. Ví dụ, một giao diện trừu tượng có thể hỗ trợ các hoạt động truy cập dữ liệu bằng cách sử dụng phương thức đơn giản như *get()* và *set()*.
- *Thành phần:* Đối tượng có thể được lắp ráp từ các đối tượng khác, và có thể chọn để ẩn các đối tượng nội bộ từ các lớp khác hoặc tiếp xúc với chúng thông qua giao diện.
- *Kế thừa:* Đối tượng có thể kế thừa từ các đối tượng khác và sử dụng chức năng trong các đối tượng cơ bản hoặc ghi đè lên nó để thực hiện hành vi mới. Hơn nữa, kế thừa làm cho bảo trì và cập nhật dễ dàng hơn, như thay đổi các đối tượng cơ sở được truyền tự động cho các đối tượng kế thừa.
- *Đóng gói:* Đối tượng tiếp xúc với đối tượng khác chỉ thông qua các phương thức, thuộc tính, và các sự kiện, và ẩn các chi tiết nội bộ như các biến từ các đối tượng khác và trạng thái. Điều này làm cho nó dễ dàng cập nhật hoặc thay thế các đối tượng, miễn là giao diện tương thích không ảnh hưởng các đối tượng và mã khác.
- *Đa hình:* Kỹ thuật này cho phép ghi đè lên hành vi của lớp cơ sở bằng cách thực hiện hành vi mới được thay đổi cho đối tượng hiện tại.
- *Tách biệt:* Đối tượng có thể được tách biệt từ người dùng bằng cách định nghĩa một giao diện trừu tượng mà các đối tượng cụ thể và người dùng có thể hiểu được.

Kiểu hướng đối tượng được sử dụng phổ biến trong mô hình đối tượng ở các lĩnh vực hoạt động khoa học, nghiệp vụ phức tạp (chẳng hạn như thông tin khách hàng hoặc đơn đặt hàng).

Những ưu điểm của kiểu kiến trúc hướng đối tượng

- *Dễ hiểu:* mô hình đối tượng có liên hệ chặt chẽ với các đối tượng trong thế giới thực nên làm cho dễ hiểu kiến trúc hơn.
- *Tái sử dụng:* cung cấp cho việc sử dụng lại thông qua tính đa hình và trừu tượng.
- *Dễ kiểm chứng:* cung cấp cách để cải thiện khả năng kiểm tra thông qua đóng gói.
- *Dễ mở rộng:* đóng gói, đa hình, và trừu tượng đảm bảo cho một sự thay đổi trong dữ liệu mà không ảnh hưởng đến giao diện và các đối tượng khác.
- *Tính kết dính cao:* bằng cách định vị các phương thức và các thuộc tính của một đối tượng, các đối tượng khác nhau có thể sử dụng phương thức và các thuộc tính của nhau và như vậy đạt được sự kết dính cao.

Hướng dẫn sử dụng

Chọn kiểu kiến trúc hướng đối tượng nếu chúng ta muốn mô hình ứng dụng dựa trên các đối tượng thế giới thực, hoặc đã có những đối tượng và các lớp phù hợp với thiết kế và các yêu cầu hoạt động. Kiểu hướng đối tượng cũng thích hợp nếu chúng ta cần phải đóng gói logic và dữ liệu với nhau thành các thành phần tái sử dụng hoặc có logic nghiệp vụ phức tạp đòi hỏi trừu tượng hóa và mô hình.

2.3.8 Kiểu kiến trúc hướng dịch vụ

Kiến trúc hướng dịch vụ (SOA) cho phép ứng dụng cung cấp các chức năng như một tập hợp các dịch vụ và tạo ra các ứng dụng sử dụng dịch vụ phần mềm. Dịch vụ được liên kết lỏng lẻo vì chúng sử dụng giao diện dựa theo một chuẩn để có thể gọi, khám phá và lựa chọn dịch vụ. Kiến trúc dịch vụ SOA tập trung vào việc cung cấp một lược đồ và thông điệp tương tác dựa trên một ứng dụng thông qua giao diện không theo hướng thành phần hoặc theo đối tượng. Một dịch vụ SOA không được coi là một nhà cung cấp dịch vụ dựa trên thành phần. Kiểu SOA có thể gói quy trình nghiệp vụ vào các dịch vụ tương thích, sử dụng một loạt các giao thức và định dạng dữ liệu để truyền thông tin. Client và các dịch vụ khác có thể truy cập các dịch vụ địa phương đang chạy trên cùng một tầng, hoặc truy cập vào dịch vụ từ xa qua mạng. Các nguyên tắc cơ bản của kiểu kiến trúc SOA là:

- *Dịch vụ là tự trị:* Mỗi dịch vụ được duy trì, phát triển, triển khai và độc lập với phiên bản. Dịch vụ có thể được đặt bất cứ nơi nào trên mạng, tại địa phương hoặc từ xa, miễn là mạng hỗ trợ các giao thức truyền thông cần thiết.
- *Dịch vụ liên kết lỏng lẻo:* Mỗi dịch vụ là độc lập với dịch vụ khác và có thể được thay thế hoặc cập nhật mà không vi phạm các ứng dụng mà nó sử dụng miễn là giao diện vẫn còn tương thích.
- *Khả năng tương thích dựa trên chính sách:* Chính sách trong trường hợp này có nghĩa là các định nghĩa về các tính năng như giao tiếp, giao thức và an ninh.

Ví dụ thường gặp của các ứng dụng hướng dịch vụ bao gồm chia sẻ thông tin, xử lý các quy trình nhiều bước như hệ thống đặt phòng và cửa hàng trực tuyến, hiển thị dữ liệu hoặc dịch vụ cụ thể trên một mạng lớn và kết hợp thông tin từ nhiều nguồn.

Những ưu điểm của kiểu kiến trúc hướng dịch vụ

- *Liên kết miền:* Tái sử dụng các dịch vụ phổ biến với giao diện chuẩn làm tăng cơ hội kinh doanh và giảm chi phí.
- *Trừu tượng:* Dịch vụ hoàn toàn tự trị và truy cập thông qua một hợp đồng chính thức, cung cấp kết nối lỏng lẻo và trừu tượng.
- *Khả năng tìm kiếm:* Dịch vụ có thể cho phép các ứng dụng và các dịch vụ khác xác định vị trí của nó và tự động gọi.
- *Khả năng tương tác:* Bởi vì các giao thức và định dạng dữ liệu dựa trên các tiêu chuẩn, nên các server và client của các dịch vụ có thể được xây dựng và triển khai trên những nền tảng khác nhau.
- *Hợp lý hóa:* Dịch vụ có thể cung cấp những chức năng cụ thể, chứ không phải là sao chép các chức năng trong số các ứng dụng.

Hướng dẫn sử dụng

Kiến trúc kiểu SOA được sử dụng nếu hệ thống mới có quyền truy cập vào các dịch vụ phù hợp mà chúng ta muốn tái sử dụng; hoặc có thể mua dịch vụ phù hợp được cung cấp bởi một công ty lưu trữ; hoặc muốn xây dựng một ứng dụng tích hợp một số dịch vụ qua một giao diện duy nhất; hoặc bạn đang tạo ra phần mềm kết hợp các dịch vụ. Ví dụ phần mềm như một

dịch vụ (SaaS), hoặc các ứng dụng dựa trên tính toán đám mây. Kiểu SOA tỏ ra phù hợp khi chúng ta phải hỗ trợ truyền thông tin giữa các phần hay chức năng của các ứng dụng trên nền tảng độc lập; hoặc khi muốn tận dụng lợi thế của các dịch vụ như xác thực; hoặc muốn để lộ các dịch vụ có thể phát hiện thông qua thư mục và có thể được sử dụng bởi các khách hàng không có kiến thức về các giao diện.

2.4 KẾT LUẬN

Chương này tập trung trình bày một số vấn đề liên quan đến nguyên lý thiết kế kiến trúc và các kiểu kiến trúc. Việc hiểu rõ các kiểu kiến trúc có ý nghĩa quan trọng trong việc quyết định lựa chọn kiến trúc sao cho phù hợp với hệ thống định xây dựng. Việc phân loại các kiểu kiến trúc có tính chất tương đối vì kiến trúc một hệ thống thực sự thường là sự kết hợp nhiều kiểu kiến trúc với nhau.

BÀI TẬP

1. Xây dựng một Bảng đặc trưng cho việc phân loại các kiểu kiến trúc.
2. Trình bày yêu cầu về quyết định thiết kế. Tham khảo: https://arxiv.org/ftp/arxiv/papers/1610/1610.09240.pdf
3. Khảo sát kiến trúc ESB và trình bày các ứng dụng của nó. Tham khảo: http://www.service-architecture.com/articles/web-services/enterprise_service_bus_esb.html
4. Trình bày hiểu biết của mình về tính kết hợp (cohesion), kết dính (coupling) giữa các lớp, thành phần và nêu lên đặc trưng này với các kiểu kiến trúc.
5. Hãy đề xuất 20 lớp trong hệ quản lý Bán sách online và phân loại thành các gói tương ứng. Dựa trên mô hình MVC để phân kiến trúc hệ thống thành 3 tầng.
6. Các kiểu giao tiếp giữa các tầng trong kiến trúc N/3 tầng
7. Các kiểu kiến trúc dựa trên J2EE cho eBay. Tham khảo: http://www.softwaresecretweapons.com/jspwiki/resources/presentations/Sun_eBay6-2_forWeb.pdf và http://www.slideshare.net/tcng3716/ebay-architecture
8. Khảo sát kiến trúc với đám mây (cloud) của Amazon. Tham khảo: https://media.amazonwebservices.com/AWS_Cloud_Best_Practices.pdf
9. Khảo sát vấn đề an toàn thông tin trong Amazon. Tham khảo: https://media.amazonwebservices.com/AWS_Cloud_Best_Practices.pdf
10. Trình bày các nguyên lý kiến trúc trong eBay. Tham khảo: http://www.ece.ubc.ca/~matei/EECE417/ebay.pdf
11. Khảo sát kiểu kiến trúc dịch vụ cỡ lớn như Google và eBay. Tham khảo: https://www.infoq.com/presentations/service-arch-scale-google-ebay

CHƯƠNG 3: MÔ HÌNH HÓA KIẾN TRÚC

Chương này nhằm trình bày một số khái niệm cơ bản nhằm thể hiện kiến trúc từ những thành phần đơn giản đến các tính chất phức tạp hơn của hệ thống như các hành vi. Các ký hiệu mô hình kiến trúc có thể khá phong phú và nhập nhằng như ngôn ngữ tự nhiên đến ngôn ngữ rất hình thức và hạn chế ngữ nghĩa như ngôn ngữ mô tả kiến trúc Rapide[2]. Mô hình với ngôn ngữ UML có thể xem là sự kết hợp giữa mô tả bằng biểu đồ với diễn đạt bằng ngôn ngữ tự nhiên và sẽ được sử dụng cho mô hình kiến trúc trong tài liệu này. Nội dung của chương bao gồm:

- Các khái niệm cơ bản về mô hình kiến trúc
- Các kỹ thuật mô hình
- Biểu diễn kiến trúc

3.1 KHÁI NIỆM VỀ MÔ HÌNH KIẾN TRÚC

Kiến trúc là một tập các quyết định thiết kế chính của hệ thống. Một khi các quyết định thiết kế được đưa ra, chúng sẽ được thể hiện bởi một *mô hình* và quá trình tạo ra mô hình đó được gọi là *mô hình hóa*. Mô hình thể hiện những quyết định thiết kế kiến trúc với những mức độ nghiêm ngặt và hình thức hóa khác nhau. Phần này sẽ bàn đến một số khái niệm được sử dụng để mô hình kiến trúc và xem xét cách chọn lựa các ký hiệu khác nhau cho mô hình.

3.1.1 Yêu cầu của mô hình hóa kiến trúc

Một trong những quyết định then chốt nhất mà những bên liên quan đến phát triển kiến trúc phải chọn:

- Các quyết định và các khái niệm cần phải mô hình
- Mức độ chi tiết của mô hình những khái niệm
- Mức độ hình thức cho mô hình

Kiến trúc sư phải quyết định lựa chọn cái gì để mô hình và mức độ chi tiết thế nào tùy theo chi phí và lợi ích của bên đối tác. Nguyên tắc ưu tiên là cái gì quan trọng nhất sẽ được mô hình chi tiết và được hình thức hóa nghiêm ngặt. Việc chọn đặc trưng nào quan trọng phụ thuộc vào dự án đang tiến hành. Các hoạt động cơ bản liên quan đến mô hình hóa bao gồm:

- Xác định các khía cạnh liên quan đến phần mềm cần mô hình và phân loại các khía cạnh này theo độ quan trọng.
- Xác định mục đích để mô hình cho mỗi khía cạnh.
- Lựa chọn các ký hiệu sẽ được sử dụng để mô hình các khía cạnh với mức chi tiết phù hợp.
- Tạo ra một mô hình bằng cách sử dụng các ngôn ngữ mô hình phù hợp.

[2] http://www.mrtc.mdh.se/han/FoPlan/ass2-bjornander.pdf

Một số khái niệm kiến trúc cơ bản

- *Thành phần:* Thành phần là khối kiến trúc đóng gói một tập chức năng hay dữ liệu của hệ thống và bên ngoài chỉ có thể truy nhập qua một giao diện xác định.
- *Kết nối:* Kết nối là khối kiến trúc quy định tương tác giữa các thành phần.
- *Giao diện:* Giao diện là những điểm mà các thành phần và kết nối tương tác với thế giới bên ngoài như các thành phần và kết nối khác.
- *Cấu hình:* Cấu hình là tập các liên kết giữa các thành phần và kết nối của kiến trúc hệ phần mềm.

Những khái niệm này tạo thành điểm khởi đầu cho mô hình hóa kiến trúc. Ở mức cơ bản nhất, mô hình những khái niệm này đòi hỏi một tập ký hiệu có thể biểu diễn một đồ thị các thành phần và kết nối. Mô hình ở mức cơ bản này ít có giá trị vì nó không đủ để biểu diễn các dự án phức tạp. Do đó, mô hình này cần phải được mở rộng bằng nhiều khái niệm khác. Một số câu hỏi sau đây được xem là trọng tâm cuả mô hình hóa kiến trúc. Các chức năng giữa các thành phần được phân rã như thế nào? Các giao diện có bản chất và các kiểu gì? Ý nghĩa liên kết của các thành phần và kết nối như thế nào? Các tính chất của hệ thống thay đổi như thế nào qua thời gian?

Tùy theo bản chất và miền ứng dụng của hệ thống định phát triển, các phần tử cơ bản có thể biểu diễn theo các cách khác nhau. Ví dụ, với phần mềm ứng dụng máy tính để bàn có thể có 500 đến 1000 thành phần và kết nối, thì dễ dàng đánh số và mô tả các thành phần này cũng như các kết nối giữa chúng. Tuy nhiên, các ứng dụng phức tạp lớn và phân tán thì khó để mô hình hóa hơn. Ví dụ, khó có thể đánh số các thành phần trong hệ phân tán trên mạng vì số lượng quá nhiều và thường xuyên thay đổi. Với các hệ như vậy, cách tốt nhất là chỉ mô hình hóa những phần nào của hệ thống được biểu diễn bởi các ca sử dụng hoặc chỉ mô hình hóa kiểu kiến trúc chi phối các phần tử trong kiến trúc.

3.1.2 Kiểu kiến trúc và mô hình hóa

Như trình bày trong Chương 2, một kiểu kiến trúc là một loạt các quyết định thiết kế kiến trúc áp dụng trong một ngữ cảnh phát triển nào đó. Nó thể hiện các ràng buộc của những quyết định thiết kế đối với hệ thống dự định phát triển. Cùng với mô hình hóa các phần tử kiến trúc cơ bản, chúng ta cũng cần mô hình kiểu kiến trúc chi phối cách sử dụng các phần tử này. Giống như kiến trúc, các kiểu kiến trúc được tạo nên bởi các quyết định thiết kế và nó có một số ích lợi sau đây:

- Giảm thiểu được nhầm lẫn cái gì được phép và cái gì không được phép trong kiến trúc và do đó giảm thiểu được phác thảo và hủy bỏ kiến trúc sau này.
- Giúp dễ dàng chỉ ra được quyết định trong kiến trúc có phù hợp với ràng buộc được đưa ra hay không và đồng thời hỗ trợ định hướng tiến hóa kiến trúc sau này.
- Kiểu kiến trúc linh hoạt và hữu ích là mô hình kiến trúc với thành phần và kết nối cho các hệ thống lớn hay động. Nó là nơi duy nhất thể hiện được những quan tâm nhiều mặt và tính hợp lý của kiến trúc.

- Do kiểu được áp dụng cho nhiều dự án khác nhau nên các mô hình kiểu có thể sử dụng lại cho các dự án khác nhau sau này.

Những loại quyết định thiết kế tìm thấy trong một kiểu kiến trúc thông thường là trừu tượng và tổng quát hơn những quyết định trong một kiến trúc. Một số loại quyết định thiết kế sau đây có thể nằm trong mô hình kiểu:

- Kiểu có thể quy định các thành phần, kết nối hay giao diện sẽ sử dụng trong kiến trúc hay trong các tình huống đặc biệt.
- Thành phần, kết nối và kiểu giao diện. Một số loại phần tử có thể được phép, phải hay cấm sử dụng trong kiến trúc. Nhiều cách tiếp cận mô hình với ngữ nghĩa khác nhau có thể kết hợp trong cùng kiểu hệ thống.
- Ràng buộc về mặt tương tác: Có những ràng buộc về mặt tương tác giữa các thành phần và kết nối với nhiều dạng khác nhau: ràng buộc thời gian (thành phần phải gọi *init()* trước bất kỳ phương thức nào khác); ràng buộc về cấu trúc (chỉ những thành phần trong tầng client mới được phép triệu gọi các thành phần trong tầng server); phải sử dụng các giao thức đã chỉ ra như FTP hay HTTP; các cách tiếp cận mô hình hỗ trợ các ràng buộc về logic như logic vị từ cấp 1, logic thời gian…
- Ràng buộc hành vi: ràng buộc có thể biểu diễn với những quy luật đơn giản hay đặc tả hành vi đầy đủ cho các thành phần theo biểu đồ máy trạng thái hữu hạn. Các cách tiếp cận mô hình hỗ trợ các ràng buộc hành vi có thể sử dụng logic hay mô hình như máy trạng thái hữu hạn.
- Ràng buộc đồng bộ: Các phần tử nào thực hiện các chức năng một cách đồng thời và cách đồng bộ hóa truy nhập tài nguyên chung được thể hiện trong kiểu kiến trúc. Nhiều cách tiếp cận hỗ trợ mô hình đồng bộ sử dụng mô hình hành vi theo thời gian như biểu đồ tuần tự hay biểu đồ trạng thái.

3.1.3 Một số đặc trưng của mô hình hóa kiến trúc

Tính chất động và tĩnh của hệ thống

Những quyết định thiết kế kiến trúc có thể đề cập đến cả hai khía cạnh động và tĩnh của hệ thống. Các khía cạnh tĩnh không liên quan đến hành vi của hệ thống khi nó thực thi. Khía cạnh động liên quan đến hành vi của hệ thống trong thời gian chạy.

Các khía cạnh tĩnh dễ dàng để mô hình vì không liên quan đến thay đổi qua thời gian. Những mô hình của khía cạnh tĩnh bao gồm cấu trúc của các thành phần và kết nối, những phép gán của các thành phần và kết nối cho các phần tử hay máy chủ xử lý và cấu hình mạng hay ánh xạ các phần tử kiến trúc thành mã nguồn hay mã nhị phân.

Các khía cạnh động của hệ thống khó để mô hình hơn vì có liên quan đến hành vi của hệ thống theo thời gian. Những mô hình của các khía cạnh động có thể là các mô hình hành vi (mô tả hành vi của thành phần hay kết nối theo thời gian), mô hình tương tác (mô tả tương tác giữa tập các thành phần và kết nối theo thời gian) hay các mô hình luồng dữ liệu (mô tả cách dữ liệu truyền đi theo thời gian qua kiến trúc). Việc phân biệt tĩnh và động có tính chất tương

đối, ví dụ cấu trúc của hệ thống thì có tính chất tĩnh nhưng đôi khi có thể thay đổi do thành phần nào đó bị hỏng hóc.

Khía cạnh chức năng và phi chức năng

Kiến trúc có khả năng thể hiện cả khía cạnh chức năng và phi chức năng của hệ thống. Khía cạnh chức năng liên quan đến *những gì* hệ thống thực hiện; khía cạnh phi chức năng liên quan đến *cách thức* hệ thống thực hiện các chức năng của nó.

Để phân biệt hai khía cạnh này, chúng ta hãy xem ví dụ Hệ quản lý tuyển sinh sau đây. Khía cạnh chức năng có thể được mô tả bằng cách sử dụng câu có dạng chủ ngữ - vị ngữ: *Hệ thống in danh sách sinh viên trúng tuyển kỳ thi tuyển sinh*. Khía cạnh phi chức năng của hệ thống có thể được mô tả bằng cách thêm trạng từ vào câu: *Hệ thống in danh sách sinh viên trúng tuyển kỳ thi tuyển sinh nhanh chóng và đáng tin cậy.*

Vì khía cạnh chức năng thì cụ thể hơn nên chúng dễ dàng mô hình chặc chẽ và hình thức với hệ thống ký hiệu được mô tả với ngữ nghĩa rõ ràng. Phần lớn các ký hiệu mô hình tập trung vào thể hiện khía cạnh chức năng của hệ thống. Ngược lại, khía cạnh phi chức năng của hệ thống có xu hướng định tính và chủ quan. Vì vậy, mô hình khía cạnh này thường là phi hình thức và ít nghiêm ngặt hơn nên ngôn ngữ tự nhiên thường được sử dụng.

Tính nhập nhằng, đúng đắn và rõ ràng

Kiến trúc là sự trừu tượng hóa của hệ thống nên chúng thể hiện thông tin về một số khía cạnh của hệ thống và bỏ qua một số khía cạnh khác. Lý tưởng nhất là các khía cạnh chủ yếu, quan trọng nhất của hệ thống được xác định rõ bởi kiến trúc. Các bộ phận được đặc tả có thể mô tả trạng thái thông thường của hệ thống và bỏ qua những trạng thái bất thường. Ba khái niệm sau đây thường được sử dụng để đặc trưng mô hình một kiến trúc:

- *Tính nhập nhằng:* Một mô hình được gọi là nhập nhằng nếu nó có nhiều hơn một thể hiện. Do kiến trúc chỉ bàn về những quyết định thiết kế chính nên thường là không đầy đủ. Vì vậy, khi một khía cạnh nào đó không được đặc tả thì các bên có thể có các cách hiểu khác nhau để bổ sung nó vào kiến trúc. Nhiều nghiên cứu cho rằng để tránh việc hiểu nhập nhằng như vậy tài liệu nên xác định rõ phần nào là đầy đủ và phần nào dành cho các quyết định phát triển trong tương lai.
- *Tính đúng đắn và rõ ràng:* Một mô hình là đúng đắn nếu nó thể hiện trúng, phù hợp với thực tế hay sai khác với thực tế một giới hạn cho phép. Một mô hình là rõ ràng nếu nó cụ thể, chi tiết và chính xác. Tính đúng đắn liên quan đến tính trúng đích và tính rõ ràng liên quan đến tính chính xác. Theo thuật ngữ kiến trúc, một mô hình được xem là đúng đắn nếu nó thể hiện thông tin đúng về hệ thống được mô hình. Trong khi đó, một mô hình là rõ ràng nếu nó thể hiện nhiều thông tin chi tiết về hệ thống.

3.1.4 Tính phức tạp của mô hình hóa

Các mô hình kiến trúc là những sản phẩm phức tạp. Mô hình kiến trúc cố gắng thể hiện tất cả các quyết định thiết kế về hệ thống mà tỏ ra quan trọng với nhiều bên khác nhau. Hơn nữa, các thành viên khác nhau lại có những cái nhìn khác nhau về hệ thống. Tuy nhiên, không có một cách tiếp cận duy nhất nào có thể nắm bắt cùng một lúc tất cả các khía cạnh của một kiến

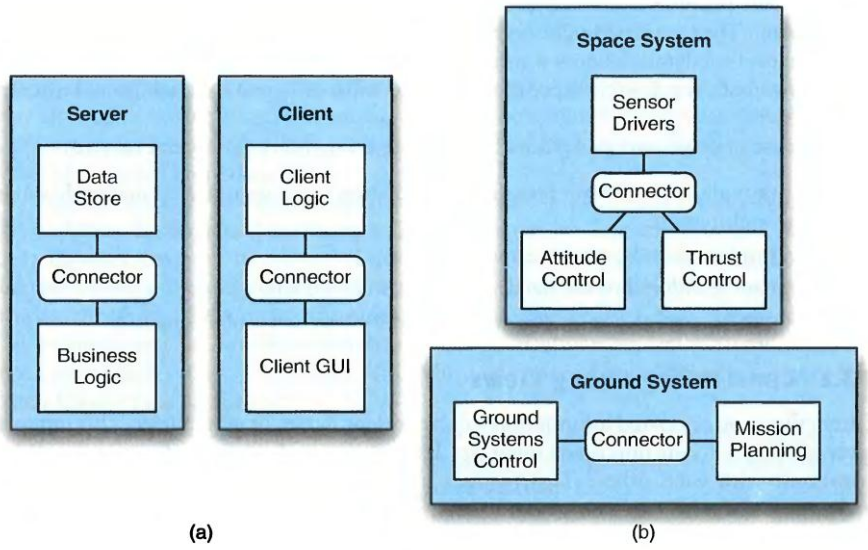
trúc. Do đó, những bộ phận khác nhau của hệ thống phải được mô hình hóa bằng những cách tiếp cận khác nhau. Ví dụ, những khía cạnh phi chức năng có thể biểu diễn bằng cách sử dụng ngôn ngữ tự nhiên hay cấu trúc đồ thị và biểu diễn các hành vi của thành phần bằng biểu đồ trạng thái.

Khung nhìn

Khung nhìn (view) là tập quyết định thiết kế liên quan đến một hay một tập quan tâm chung. *Quan điểm* (viewpoint) là cơ sở cho cái nhìn. Chúng ta có thể hiểu *quan điểm xem như một bộ lọc thông tin và khung nhìn là những gì ta nhìn thấy qua bộ lọc đó*.

Ví dụ, một hệ thống với các thành phần phần mềm phân tán trên nhiều máy chủ. Quan điểm triển khai liên quan đến những quyết định thiết kế là cách gán các thành phần cho các máy chủ như thế nào. Khung nhìn triển khai thể hiện cách gán các thành phần cho các máy chủ trong hệ cụ thể này như thế nào. Ví dụ Hình 3.1a là khung nhìn triển khai của hệ Client-Server và 3.1b là khung nhìn triển khai của hệ phân tán điều khiển hệ không gian Lunar. Cả hai khung nhìn này thể hiện quan điểm triển khai các thành phần trên máy chủ. Các quan điểm thường liên quan đến những quan tâm nhất định nào đó. Sau đây là một số quan điểm về hệ thống:

- *Quan điểm logic* (logical viewpoint): Thể hiện các thực thể logic trong hệ phần mềm và cách kết nối của chúng.
- *Quan điểm vật lý* (physical viewpoint): Thể hiện các thực thể vật lý trong hệ thống và cách kết nối của chúng.
- *Quan điểm triển khai* (deployment viewpoint): Cách các thực thể logic được ánh xạ vào các thực thể vật lý.
- *Quan điểm tiến trình* (process viewpoint): Thể hiện cách quản lý tương tranh và luồng trong hệ thống.
- *Quan điểm hành vi* (behavioral viewpoint): Thể hiện các hành vi của hệ thống.



Hình 3.1: Khung nhìn triển khai:

(a) Hệ client-server; (b) Hệ quan sát trạm không gian [3]

Quan điểm và khung nhìn trong mô hình được xem là quan trọng vì những lý do sau đây:

- Chúng nó cung cấp cách để hạn chế thông tin trong một bộ phận của kiến trúc hệ thống.
- Chúng hiển thị cùng một lúc các khái niệm liên quan và thể hiện nhu cầu của các bên liên quan.
- Chúng có thể được sử dụng để hiển thị cùng dữ liệu ở những mức độ trừu tượng khác nhau.

3.1.5 Tính chất của các khung nhìn

Với nhiều khung nhìn khác nhau như vậy, cho nên những thông tin liên quan hệ thống cần phát triển có thể được thể hiện theo nhiều cách khác nhau. Vấn đề làm thế nào để biết hai khung nhìn là không mâu thuẫn với nhau? Chúng ta cho rằng hai khung nhìn A và B là *mâu thuẫn nhau* nếu một bên A khẳng định những quyết định thiết kế là đúng trong khi bên B khẳng định là sai. Sau đây là một số loại mâu thuẫn:

- *Mâu thuẫn trực tiếp:* Mâu thuẫn này xảy ra khi hai khung nhìn khẳng định trực tiếp những mệnh đề mâu thuẫn. Ví dụ khẳng định "hệ thống chạy trên 2 máy chủ" và "hệ thống chạy trên 3 máy chủ" được xem là mâu thuẫn nhau.
- *Mâu thuẫn mịn hóa:* Mâu thuẫn này xảy ra khi hai khung nhìn của cùng hệ thống ở các mức chi tiết khác nhau khẳng định những mệnh đề mâu thuẫn nhau. Ví dụ, khung nhìn cấu trúc ở mức đỉnh cao nhất có một thành phần mà không có mặt trong khung nhìn kiến trúc ở mức thấp hơn.

3.2 KỸ THUẬT MÔ HÌNH

3.2.1 Một số kỹ thuật mô hình

Mỗi kiến trúc sư phần mềm đều có những ký hiệu và kỹ thuật để mô hình các khía cạnh khác nhau của kiến trúc. Các kỹ thuật này thay đổi theo những chiều khác nhau:

- Các kỹ thuật này có thể dùng để mô hình những đối tượng nào?
- Nó có thể hiện được ngữ nghĩa của kiến trúc và công cụ hỗ trợ không?
- Phương pháp luận hay tiến trình phát triển nào được sử dụng và nhằm mục đích gì? Điều này phụ thuộc vào các bên liên quan và kiến trúc sư hệ thống.

Sử dụng các cách tiếp cận khác nhau để mô hình hóa kiến trúc hệ thống sẽ giúp chúng ta có một cái nhìn đa dạng, đầy đủ hơn về hệ thống cần phát triển. Do đó, chúng ta không nên quá thiên vị và định kiến với một cách tiếp cận nào. Trong phần tiếp theo sẽ trình bày các cách tiếp cận khác nhau để mô hình các kiến trúc phần mềm[3]:

- Ngôn ngữ phân tích và thiết kế kiến trúc (AADL: Architecture Analysis & Design Language) là ngôn ngữ đầy đủ để thiết kế cả phần cứng và phần mềm của hệ thống. Nó hỗ trợ bộ xử lý, thiết bị và cổng cũng như tiến trình, luồng và dữ liệu.
- Môi trường và ngôn ngữ dựa trên kiến trúc (ACME: The Architecture Based Language and Environment) là ngôn ngữ nhỏ và hơi đơn giản. Nó thể hiện các khái niệm hệ thống, thành phần, kết nối, cổng, vai trò, biểu diễn. Một hệ thống được cấu thành bởi các thành phần được liên kết bởi các kết nối, cổng là các điểm cuối của các kết nối. ACME có thể xem là tập con của AADL.
- Darwin: hỗ trợ hợp phân cấp. Nghĩa là một thành phần là hợp của các thành phần sơ cấp. Darwin cũng hỗ trợ cấu trúc các chương trình song song và mô hình cấu hình mạng.
- Wright: được xây dựng dựa trên các thành phần trừu tượng, các kết nối và cấu hình. Các cấu hình có thể chia thành các thể hiện (một kiểu đặc tả thành phần), gắn kết (mô tả cấu hình của hệ thống) và phân cấp (một thành phần có thể thể hiện các thành phần khác).
- Aesop (Aesop: The Software Architecture Design Environment Generator) là tập các công cụ được thiết kế để phát triển mô hình hệ thống. Nó dựa trên môi trường UNIX, có mở rộng kiểu ống và lọc để mô hình những đặc điểm này. Nhờ có nhân tổng quát, nó thích hợp cho mọi môi trường.
- UML là ngôn ngữ mô hình hóa được sử dụng rộng rãi. Mặc dù UML không phải là ngôn ngữ mô tả kiến trúc nhưng nó lại phù hợp để mô hình hệ thống. Phần tiếp theo trong chương này sẽ dành chủ yếu trình bày UML cho biểu diễn kiến trúc.

[3] http://www.mrtc.mdh.se/han/FoPlan/ass2-bjornander.pdf

- TASM (TASM: Timed Abstract State Machine) thực sự không phải là ngôn ngữ mô tả kiến trúc nhưng nó có thể sử dụng để mô hình những hệ khá phức tạp. Nó dựa trên máy trạng thái trừu tượng với mở rộng thời gian. Một mô hình hệ thống được cấu thành bởi tập biến theo dõi "đọc", tập biến điều khiển "viết" và tập luật.

3.2.2 Đánh giá các kỹ thuật mô hình

Trong phần này chúng ta đã xem xét các chiều được sử dụng để đặc trưng các kỹ thuật mô hình khác nhau. Các chiều này sẽ được sử dụng để đánh giá các kỹ thuật mô hình [18].

Bảng 3.1: Số chiều đánh giá kỹ thuật mô hình

Số TT	Chiều	Ý nghĩa
1	Phạm vi và mục đích	Phạm vi của kỹ thuật là gì? Cái gì được mô hình và cái gì không được mô hình?
2	Các phần tử cơ bản	Các phần tử cơ bản là gì, nó được mô hình như thế nào?
3	Các khía cạnh tĩnh và động	Mức độ mà kỹ thuật hỗ trợ mô hình khía cạnh tĩnh của kiến trúc.
4	Mô hình động	Mức độ mà mô hình cần thay đổi để phản ánh thay đổi khi hệ thống thực thi
5	Khía cạnh phi chức năng	Kỹ thuật hỗ trợ mô hình khía cạnh phi chức năng của kiến trúc ở mức độ nào.
6	Nhập nhằng	Cách tiếp cận này hạn chế hay cho phép tính nhập nhằng thế nào
7	Đúng đắn	Cách tiếp cận này hỗ trợ xác định tính đúng đắn của mô hình thế nào
8	Rõ ràng	Các khía cạnh khác nhau của kiến trúc được mô hình ở mức độ chi tiết thế nào
9	Quan điểm	Mô hình hỗ trợ quan điểm nào
10	Phi mâu thuẫn	Cách tiếp cận hỗ trợ xác định tính phi mâu thuẫn của những khung nhìn khác nhau được diễn đạt ở mức độ nào trong mô hình.

3.3 MÔ HÌNH HÓA KIẾN TRÚC VỚI UML

3.3.1 Ngôn ngữ mô hình thống nhất UML

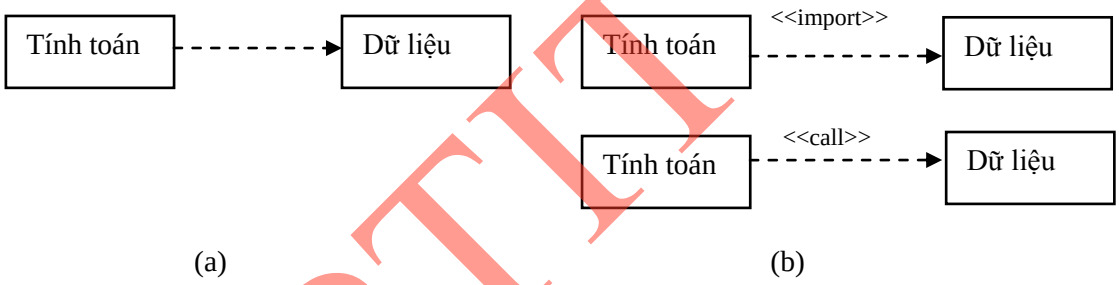
Ngôn ngữ mô hình hóa thống nhất UML (Unified Modeling Language) là bộ ký hiệu với biểu đồ quen thuộc nhất để tạo ra bản thiết kế phần mềm. UML đã kết hợp những khái niệm từ những ký hiệu đã có trước đó như biểu đồ của Booch, OMT của Rumbaugh, OOSE của Jacobson và biểu đồ trạng thái của Harel. Những điểm mạnh chủ yếu của UML đã được thừa nhận rộng rãi với nhiều mô hình hóa cấu trúc, quan điểm và công cụ hỗ trợ. Nó cung cấp tính nhiều chiều cho kỹ thuật mô hình với mười bốn biểu đồ để mô hình hóa các khía cạnh tĩnh và động của phần mềm [7].

Đã có nhiều cuộc tranh luận về khả năng sử dụng UML cho mô hình hóa kiến trúc. Các phiên bản trước đây UML 1.* tập trung nhiều vào thiết kế chi tiết cho các lớp, thuộc tính và phương thức. UML 2.0 đã mở rộng thực sự UML để hỗ trợ tốt hơn các cấu trúc mức cao trong kiến trúc. Việc hiểu đầy đủ ý nghĩa của các biểu đồ UML được xem là then chốt để sử dụng nó cho việc mô tả các quyết định thiết kế. UML được xem là có biểu diễn rõ ràng hơn các

ngôn ngữ khác nhưng lại chú trọng thiết kế sao cho tổng quát nhất nên có thể dẫn đến cách hiểu nhập nhằng.

Ví dụ, Hình 3.2 (a) biểu diễn sự phụ thuộc của hai thành phần trong biểu đồ UML bởi mũi tên. Nó chỉ rằng phần tử ở đuôi *Tính toán* phụ thuộc phần tử ở đầu mũi tên *Dữ liệu*. Tuy nhiên, điều này có thể hiểu theo một số nghĩa sau đây:

- Phần tử nào đó trong tính toán gọi Dữ liệu.
- Những thể hiện của Tính toán chứa con trỏ hay tham chiếu đến một thể hiện của Dữ liệu.
- Tính toán đòi hỏi Dữ liệu biên dịch.
- Cài đặt của Tính toán có một phương thức lấy tham số là thể hiện của cài đặt của dữ liệu.
- Tính toán gửi thông điệp đến Dữ liệu
- Tính toán và Dữ liệu có thể là thành phần, lớp, module trong C hay dịch vụ web



Hình 3.2: (a) Biểu diễn phụ thuộc (b) Sử dụng nhãn để thể hiện ngữ nghĩa

May mắn là UML đã đưa ra một cách để cho phép người sử dụng định nghĩa những thuộc tính mới và những ràng buộc như một nhãn để áp dụng vào những phần tử của mô hình. Những ràng buộc như nhãn này có thể là ngôn ngữ tự nhiên.

Ví dụ, Hình 3.2 (b) chỉ ra hai cách hiểu khái niệm phụ thuộc với hai nghĩa khác nhau được gán cho nhãn. Điều này không có nghĩa là tính nhập nhằng đã được khắc phục hoàn toàn vì còn phải hiểu thế nào là "import" và thế nào là "call".

Như vậy, một mình biểu đồ UML thì không thể chứa đủ thông tin để thể hiện mô hình hệ thống. Các bên tham gia có thể đưa ra những thỏa thuận về cách thể hiện các khía cạnh của biểu đồ UML và viết thành tài liệu các thỏa thuận này bằng ngôn ngữ tự nhiên.

3.3.2 Biểu diễn kiến trúc với UML

Như đã trình bày ở trên, kiến trúc được thể hiện bởi những khung nhìn khác nhau và mỗi khung nhìn tương ứng với một khía cạnh đặc biệt của hệ thống. Bức tranh đầy đủ của hệ thống chỉ có thể có được qua tất cả các khung nhìn. Trong UML, có năm khung nhìn được xác định như sau:

- *Khung nhìn ca sử dụng* (Use case view): kiến trúc được mô tả bởi *biểu đồ ca sử dụng* gồm một tập *ca sử dụng* và các quan hệ giữa chúng. Các kịch bản là mô tả chi tiết các

ca sử dụng nhằm mô tả tương tác giữa các đối tượng và giữa các tiến trình. Chúng được sử dụng để xác định các phần tử kiến trúc cũng như để minh họa và xác thực thiết kế kiến trúc. Chúng cũng được sử dụng để kiểm chứng ban đầu bản mẫu kiến trúc.

- *Khung nhìn logic* (Logical view): Cung cấp các chức năng cho người sử dụng đầu cuối. Trong UML có thể sử dụng các biểu đồ các *biểu đồ lớp, biểu đồ hoạt động, biểu đồ trạng thái* để thể hiện khung nhìn này.
- *Khung nhìn tiến trình* (Process view): Khung nhìn tiến trình liên quan đến khía cạnh động của hệ thống, giải thích cách xử lý và giao tiếp của các tiến trình trong hệ thống. Khung nhìn này tập trung xem xét các khía cạnh vào thời gian chạy của hệ thống như tương tranh, phân bố tài nguyên, tích hợp…Trung UML có thể sử dụng *biểu đồ hoạt động* để thể hiện khung nhìn này.
- *Khung nhìn phát triển* (Development view): Khung nhìn này thể hiện quan điểm của người lập trình về hệ thống. Nó cũng được gọi là khung nhìn cài đặt hay thành phần và sử dụng biểu đồ thành phần để biểu diễn thành phần của hệ thống. Trong UML có thể sử dụng *biểu đồ gói* để thể hiện khung nhìn này.
- *Khung nhìn vật lý* (Physical view): Thể hiện quan điểm của người kỹ sư hệ thống. Nó liên quan đến cấu hình của các thành phần phần mềm trên các tầng vật lý cũng như cách giao tiếp về mặt vật lý giữa các thành phần này. Nó cũng còn được gọi là khung nhìn triển khai và trong UML có thể sử dụng *biểu đồ triển khai* để thể hiện khung nhìn này.

Một cách khái quát, kiến trúc phần mềm có thể hiểu gồm hai phần:

- Kiến trúc logic (Logical architecture): Liên quan đến cấu trúc phần mềm và quan hệ của các thành phần.
- Kiến trúc vật lý (Physical architecture): Liên quan đến cấu trúc vật lý cho triển khai phần mềm.

Như vậy, nhìn từ góc độ này, chúng ta có thể xem xét những yêu tố nào tạo nên một kiến trúc tốt. Sau đây là một số hướng dẫn có thể sử dụng để xem xét, đánh giá một kiến trúc thế nào được cho là tốt [18]:

- Mô tả đúng những thành phần xác định hệ thống cả về mặt kiến trúc logic lẫn mặt kiến trúc vật lý.
- Có được cách để người phát triển dễ dàng xác định được chức năng hay khái niệm nào đó được cài đặt. Chức năng hay khái niệm có thể theo hướng ứng dụng (theo mô hình miền nào đó) hay theo hướng thiết kế (giải pháp cài đặt kỹ thuật). Điều này cũng ám chỉ những yêu cầu của hệ thống cần phải theo dõi được cho đến khi xây dựng được mã xử lý nó.
- Dễ dàng thay đổi và mở rộng tại một vị trí mà không ảnh hưởng đến phần còn lại của hệ thống.

- Các giao diện và các phụ thuộc giữa các bộ phận được xác định rõ để một người phát triển một bộ phận nào đó không cần phải hiểu đầy đủ các chi tiết của toàn bộ hệ thống.
- Việc sử dụng lại cho phép tích hợp những bộ phận có sẵn vào trong thiết kế và tạo nên những thành phần mới để sử dụng cho hệ thống khác.

Kiến trúc thỏa mãn tất cả những tính chất này không phải dễ dàng thiết kế nên nhiều khi chúng ta phải có một thỏa hiệp nào đó. Tuy nhiên, việc xác định kiến trúc cơ bản tốt là một trong những bước quan trọng nhất trong phát triển thành công hệ thống. Nếu không nhận thức được đầy đủ như vậy và cho rằng kiến trúc có thể xác định được từ mã nguồn thì nhất định hệ thống sau này sẽ khó hiểu, khó thay đổi cũng như khó dễ dàng mở rộng và bảo trì. Trong phần còn lại của chương này, chúng ta sẽ khảo sát chi tiết hơn các quan điểm kiến trúc về mặt logic và vật lý.

3.3.3 Kiến trúc logic

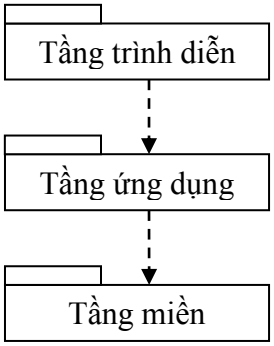
Kiến trúc logic bao hàm logic của ứng dụng và không quan tâm về phân bố vật lý của kiến trúc trong môi trường gồm những máy tính khác nhau. Kiến trúc logic cho ta cách hiểu rõ ràng về cách xây dựng của hệ thống và điều này giúp dễ dàng quản trị và phối hợp công việc giữa các nhóm. Không phải tất cả các thành phần của kiến trúc logic đều được phát triển trong dự án mà có thể sử dụng lại các thư viện lớp, thành phần nhị phân và mẫu thiết kế có sẵn. Sau đây là một số câu hỏi mà kiến trúc logic cần phải đáp ứng:

- Cấu trúc chung của hệ thống là như thế nào?
- Hệ thống cung cấp các chức năng nào?
- Các lớp chính là gì và chúng nó liên kết với nhau như thế nào?
- Các lớp và đối tượng phối hợp với nhau như thế nào để cung cấp các chức năng?
- Các cơ chế được áp dụng như thế nào để không gây tranh chấp nhau trong thiết kế?
- Kế hoạch nào phù hợp cho nhóm người phát triển hệ thống này?

Kiến trúc logic không phải là một thiết kế logic đầy đủ mà chỉ tập trung trả lời những câu hỏi trên khi được mô tả bằng biểu đồ UML. Kiến trúc đưa ra cách tổ chức, ràng buộc và một tập các công cụ cho thiết kế. Trong UML, các biểu đồ được sử dụng để mô tả kiến trúc logic là *biểu đồ lớp, biểu đồ trạng thái, biểu đồ hoạt động và biểu đồ tuần tự.*

Cấu trúc trong kiến trúc logic

Biểu đồ lớp mô tả kiến trúc có thể tập trung vào cấu trúc của hệ thống bằng cách chỉ ra các gói, thành phần, phụ thuộc và các giao diện. Một kiến trúc quen thuộc là kiến trúc ba tầng trong đó hệ thống được chia thành tầng trình diễn, tầng ứng dụng và tầng miền và được biểu diễn với UML như trong Hình 3.3.



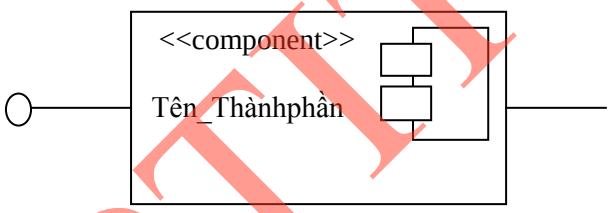
Hình 3.3: Kiến trúc 3 tầng với phụ thuộc biểu diễn như các các gói UML

Thành phần

Ngoài khái niệm gói mà UML cung cấp như là một cơ chế để nhóm các lớp phụ thuộc nhau, chúng ta có thể phân rã hệ thống thành các thành phần. Cách phân rã theo thành phần được xem là cơ chế để nhóm các lớp có ngữ nghĩa phong phú hơn là nhóm theo gói. *Thành phần* (component) là một đơn vị đóng gói trạng thái và hành vi của một tập các lớp. Ngược với khái niệm gói, các nội dung của thành phần có phạm vi truy nhập là *private* và hơn nữa thành phần thực hiện hành vi phải thông qua các giao diện.

Tính chất đóng gói đầy đủ và tách biệt giao diện với cài đặt làm cho thành phần trở nên một đơn vị có thể thay thế được bởi một thành phần có chức năng tương đương tại thời gian thiết kế hay thời gian chạy. Một khiá cạnh quan trọng của phát triển phần mềm dựa trên thành phần là khả năng sử dụng lại các thành phần có sẵn. Các thành phần luôn được xây dựng sao cho nó có thể xử lý độc lập hết sức có thể để khi thay thế chúng có thể kết nối với nhau qua giao diện.

Mỗi thành phần được biểu diễn bởi hình chữ nhật (Xem Hình 3.4) với nhãn <<component>>, trên góc trên bên phải có thể có icon thành phần và hai giao diện (một giao diện yêu cầu và một giao diện cung cấp). Thành phần có thể chứa nhiều thành phần bên trong nó. Mô hình thành phần và phát triển dựa vào thành phần sẽ được trình bày đầy đủ hơn trong Chương 4.



Hình 3.4: Biểu diễn của thành phần

Mẫu thiết kế kiến trúc

Mẫu thiết kế đã nhận được nhiều quan tâm của cộng đồng hướng đối tượng vì chúng thể hiện sự đột phá trong phát triển phần mềm. Một số đặc trưng sau đây thể hiện được sự quan trọng của việc sử dụng mẫu thiết kế trong thiết kế kiến trúc phần mềm:

- Giải pháp hợp lý: Các mẫu thiết kế là những giải pháp hợp lý đã được đề xuất bởi những nhà thiết kế kinh nghiệm.
- Tổng quát: Các mẫu thiết kế không phụ thuộc vào kiểu hệ thống, ngôn ngữ lập trình hay miền ứng dụng.
- Đã trải nghiệm: Các mẫu thiết kế thường được phát hiện từ các hệ thống đã được xây dựng. Nó không chỉ là sản phẩm tư duy hàn lâm mà đã được kiểm định thành công trong nhiều hệ thống.
- Đơn giản: Các mẫu thiết kế thường rất nhỏ vì nó chỉ liên quan đến một số lớp. Để xây dựng các giải pháp phức tạp hơn, các mẫu thiết kế sẽ được kết hợp và pha trộn với nhau.

- Sử dụng lại được: Các mẫu thiết kế được mô tả sao cho dễ dàng sử dụng. Tuy nhiên, cần chú ý rằng nó được sử dụng lại ở mức thiết kế mà không phải ở mức mã lập trình.

Chi tiết về mô hình mẫu thiết kế trong UML cũng như các mẫu thiết kế thông dụng sẽ được trình bày đầy đủ hơn trong các Chương 8-10.

3.3.4 Kiến trúc vật lý

Kiến trúc vật lý nhằm mô tả chi tiết cách gán sản phẩm phần mềm của hệ thống vào các máy tính cụ thể. Nó thể hiện cấu trúc đồ thị của phần cứng gồm những đỉnh và cạnh là cách chúng kết nối với nhau. Nó cũng minh họa sự phụ thuộc của các thành phần phần mềm trong kiến trúc logic và sự phân bố phần mềm theo biểu đồ triển khai. Kiến trúc vật lý trả lời các câu hỏi:

- Hệ thống gồm có những máy tính và những thiết bị phần cứng nào và chúng được kết nối với nhau như thế nào?
- Môi trường trong đó các bộ phận khác nhau của hệ thống chạy là gì?
- Những thành phần được triển khai trên những máy tính nào?
- Các file mã nguồn phụ thuộc với nhau như thế nào? Nếu một file thay đổi thì file nào sẽ phải biên dịch lại?

Có sự tương ứng giữa kiến trúc vật lý và kiến trúc logic. Kiến trúc vật lý liên quan đến việc cài đặt và do đó cũng được mô hình bởi các biểu đồ thành phần và biểu đồ triển khai trong UML. Biểu đồ thành phần chỉ cách cài đặt các thành phần, trong khi đó biểu đồ triển khai chỉ ra kiến trúc chạy của hệ thống.

Phần cứng

Khái niệm phần cứng trong kiến trúc vật lý có thể chia thành:

- Thiết bị: Tài nguyên tính toán về mặt vật lý để cho các thành phần phần mềm được triển khai thực thi.
- Giao tiếp: Kết nối giữa các bộ xử lý được mô tả bởi các phương tiện vật lý như dây cáp và giao thức phần mềm như TCP/IP.
- Môi trường thực thi: Các thiết bị được mô hình như các đỉnh con với các thành phần phần mềm được triển khai.

Phần mềm

Thông thường phần mềm trong kiến trúc hệ thống được hiểu một cách chung nhất là gồm các bộ phận. Tên gọi để chỉ cho một đơn vị môdun là gọi hệ thống con. Nó có giao diện và gồm nhiều hệ thống con hoặc các lớp. Các hệ thống con có thể có môi trường thực thi đi đôi với nó. Trong UML một hệ thống con được mô hình như một thành phần với nhãn <<subsystem>>. Hệ thống con khác với gói vì gói gộp một số lớp nhưng thường thiếu ngữ nghĩa. Trong thiết kế thường rõ ràng hơn nếu định nghĩa một hay nhiều giao diện như mẫu thiết kế thích ứng façade (sẽ được trình bày chi tiết trong Chương 10) cho hệ thống con. Bằng cách sử dụng façade, hệ thống con trở thành một đơn vị đóng gói với thiết kế bên trong được

ẩn giấu và chỉ có giao diện façade là phụ thuộc với những phần tử khác của hệ thống. Các khái niệm chính sau đây được sử dụng để mô tả phần mềm trong kiến trúc vật lý:

- Thành phần: Thành phần trong UML được định nghĩa là bộ phận cài đặt các phần tử mô hình như được xác định trong biểu đồ lớp hay biểu đồ tương tác.
- Sản phẩm: Biểu diễn thông tin về mặt vật lý được sử dụng bởi tiến trình phát triển phần mềm như file mô hình, file nguồn, file nhị phân…
- Đặc tả triển khai: Là tập các tính chất xác định các tham số thực thi của thành phần được triển khai trên một máy hay cụm máy.

3.4 KẾT LUẬN

Chương này đã trình bày một số khái niệm cơ bản về kiến trúc, các kiểu kiến trúc và các biểu đồ UML liên quan đến biểu diễn kiến trúc phần mềm. Đặc biệt nội dung chú trọng đề cập đến mô hình với UML cho biểu diễn kiến trúc dựa trên các khung nhìn vì nó thể hiện các quan điểm khác nhau cần có khi tiến hành mô hình kiến trúc của hệ cần xây dựng. Các kiến thức chương này được xem là nền tảng để bạn đọc có thể theo dõi các chương tiếp theo.

BÀI TẬP

1. Phân biệt các khái niệm kiến trúc, thành phần, mẫu thiết kế, khung nhìn, khung làm việc, cơ sở hạ tầng.
2. Sử dụng Tool VP để thể hiện các biểu đồ với các khung nhìn khác nhau.
3. Biểu diễn thành phần và kết nối của chúng trong UML và thể hiện với Tool VP
4. Khảo sát mô hình khung nhìn kiến trúc 4+1 của Philippe Kruchten. Tham khảo: https://www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf
5. Khảo sát các kiểu kiến trúc và ứng dụng hiện nay của các hệ thống game online.
6. Trình bày các đặc trưng và mối liên quan giữa kiến trúc vật lý và kiến trúc logic.
7. Khảo sát kiến trúc của các hệ thống thương mại điện tử như eBay và chỉ ra số lượng dòng mã, số lớp…và các công nghệ đã sử dụng. Tham khảo:

   http://www.slideshare.net/tcng3716/ebay-architecture
8. Sử dụng Tool như VP để biểu diễn các thành phần, biểu đồ triển khai và gói của hệ Quản lý đăng ký học theo tín chỉ
9. Sử dụng Tool như VP để biểu diễn các thành phần, biểu đồ triển khai và gói của hệ Quản lý thư viện
10. Sử dụng Tool như VP để biểu diễn các thành phần, biểu đồ triển khai của hệ Quản lý bán sách online

CHƯƠNG 4: KIẾN TRÚC PHẦN MỀM DỰA TRÊN THÀNH PHẦN

Chương này tập trung trình bày một số nội dung sau đây:

- Khái niệm thành phần và lập trình hướng thành phần
- Mô hình thành phần phần mềm
- Kiến trúc dựa trên thành phần
- Biểu diễn UML cho kiến trúc dựa trên thành phần

4.1 GIỚI THIỆU

4.1.1 Từ lập trình hướng đối tượng đến lập trình hướng thành phần

Trong lập trình hướng đối tượng (OOP: Object-Oriented Programming), các chương trình được xây dựng dựa trên các lớp và các đối tượng. Mỗi đối tượng được lưu trữ trong một đoạn của bộ nhớ máy tính và liên kết với dữ liệu để thực hiện hành vi được gán cho nó. Các ngôn ngữ lập trình đều có các kiểu dữ liệu đã xác định sẵn như số nguyên *int*, ký tự *char*…và người lập trình có thể tự định nghĩa kiểu dữ liệu cho bài toán của mình. Như vậy, họ có thể mô tả các thực thể độc lập để tăng tính tự nhiên trong các ứng dụng của họ.

Lập trình hướng thành phần (COP: Component-Oriented Programming) cho phép các chương trình được xây dựng từ các khối mã máy là các thành phần phần mềm đã được xây dựng sẵn theo các tiêu chuẩn được xác định trước bao gồm giao diện, kết nối, phiên bản và triển khai. Các thành phần này có thể được mua trực tuyến hay mở rộng các thành phần có sẵn với logic nghiệp vụ đã xác định. Về nguyên tắc, mỗi thành phần có thể được tái sử dụng với ngữ cảnh xác định.

Trong khi OOP nhấn mạnh các lớp và các đối tượng, COP phát triển phần mềm bằng cách lắp ghép các thành phần và chú trọng đến các giao diện và thành phần. Có thể nói rằng COP là lập trình dựa trên giao diện; các thành phần trong COP không cần biết thành phần đối tác thực thi chức năng của nó như thế nào miễn là có thể giao tiếp qua giao diện. Xây dựng các hệ thống từ các thành phần có sẵn là cách làm tự nhiên trong công nghệ chế tạo và xây dựng.

Ví dụ, công nghiệp ô tô sử dụng các thành phần với mọi kích cỡ từ một ốc vít nhỏ đến cả một hệ phức tạp như động cơ và bộ phận truyền động. Như vậy, có thể nói nhà máy ô tô hiện đại là một hệ thống tích hợp hơn là sản xuất. Hàng trăm năm nay, ngành công nghiệp này đã thừa nhận những chuẩn để tích hợp và lắp ráp các bộ phận có sẵn nhằm tăng tốc độ phát triển các sản phẩm có độ phức tạp cao.

Tuy nhiên, trong công nghiệp phần mềm, các sản phẩm chủ yếu vẫn là thủ công và do đó năng suất thấp, chất lượng không đảm bảo, các dự án phần mềm hầu hết đều vượt thời gian và chi phí. Hiện tượng này thường được gọi tên là *khủng hoảng phần mềm*. Với sự phát triển như vũ bão của công nghệ phần cứng, giá thành của việc phát triển một ứng dụng máy tính chủ yếu phụ thuộc vào phần mềm. Vấn đề chính của kỹ thuật phần mềm là làm thế nào để tạo ra phần mềm có chất lượng và hiệu quả? Đa số kỹ sư phần mềm xem thành phần như cách tiếp cận công nghệ quan trọng để giải quyết vấn đề khủng hoảng phần mềm. Một số nguyên nhân

lý giải tại sao COP lại quan trọng là nó cung cấp một mức trừu tượng cao và càng ngày càng có nhiều thư viện thành phần tái sử dụng hỗ trợ cho việc phát triển các ứng dụng trong nhiều lĩnh vực khác nhau. Ba đặc trưng chính COP: giảm thiểu phức tạp, quản lý thay đổi và tái sử dụng.

Giảm thiểu phức tạp

Chúng ta đang sống trong một thế giới phức tạp với sự bùng nổ thông tin từ 1 đến 2 exabytes (1 exabyte = 1 tỷ gigabytes) thông tin mỗi năm và kích cỡ cũng như độ phức tạp của phần mềm đã được tăng lên đáng kể cùng với sự bùng nổ đó.

COP được cho là cách giải quyết hiệu quả các phần mềm phức tạp dựa trên phương cách "chia để trị". Do yêu cầu người dùng, các thông số kỹ thuật, nhân sự cho đến ngân sách cũng như công nghệ thường xuyên thay đổi...nên "thay đổi" được xem là bản chất của kỹ thuật phần mềm. Do đó, một trong những nguyên lý cơ bản của kỹ nghệ phần mềm là chú trọng quản lý sự thay đổi nghĩa là *trọng tâm chính trong kiến trúc và thiết kế là quản lý các phụ thuộc và thay đổi*.

COP đem lại một phương pháp hiệu quả để giải quyết vấn đề thay đổi thường xuyên trong kỹ thuật phần mềm từ lập kế hoạch đến thiết kế và cài đặt. Lý do là các thành phần dễ dàng thích ứng với những yêu cầu mới và sự thay đổi. Các kỹ sư phần mềm đều cho rằng cách tốt nhất để giải quyết các thay đổi thường xuyên là xây dựng các hệ thống từ các thành phần có thể tái sử dụng theo tiêu chuẩn thành phần và kiến trúc kiểu lắp ghép.

Sử dụng lại

Sử dụng lại một số bộ phận phần mềm như hàm, lớp…trong phát triển thường được xem là làm tăng hiệu suất và cải tiến được chất lượng. Nhiều mức độ khác nhau về sử dụng lại phần mềm:

- Sao chép lại mã nguồn là mức sử dụng lại thấp nhất. Các thư viện hướng thủ tục trong C là dạng sử dụng lại dựa trên mã nguồn nhưng khó mở rộng, trong khi các thư viện lớp là kiểu sử dụng lại tốt hơn và có thể mở rộng được. Tuy nhiên, mức sử dụng lại này đòi hỏi phải hiểu biết nhiều về nó trước khi các hàm/lớp có thể được sử dụng lại. Hơn nữa, nó chỉ hỗ trợ sử dụng lại theo kiểu "hộp trắng", nghĩa là sẽ bị ảnh hưởng nếu bên trong các lớp bị thay đổi. Ví dụ, trong ngôn ngữ C++ hoặc Java, các lớp được kết hợp với cài đặt các lớp cơ sở nên thay đổi bất kì lớp cơ sở nào trong cây kế thừa cũng sẽ phá vỡ các lớp sử dụng chúng.
- COP hỗ trợ mức sử dụng lại cao nhất bởi vì nó cho phép nhiều kiểu sử dụng lại: sử dụng lại kiểu hộp trắng, sử dụng lại kiểu hộp xám, sử dụng lại kiểu hộp đen. Sử dụng lại kiểu hộp trắng nghĩa là mã nguồn của một thành phần có thể sử dụng lại, tích hợp hoặc cập nhật. Sử dụng lại kiểu hộp đen dựa trên nguyên lý ẩn dấu thông tin nghĩa là giao diện chỉ ra các dịch vụ mà một thành phần có thể cung cấp được. Giao diện không thay đổi nhưng thành phần có thể thay đổi bên trong mà không ảnh hưởng đến đối tác sử dụng nó. Sử dụng lại kiểu hộp xám xem là trung gian giữa sử dụng lại kiểu hộp trắng và đen.

4.1.2 Khái niệm thành phần

Kỹ thuật phần mềm có quá trình phát triển đã gần năm mươi năm với nhiều công nghệ được phát minh và áp dụng bao gồm lập trình cấu trúc, công nghệ CASE, công nghệ hướng đối tượng...Tuy nhiên, về bản chất các cách phát triển phần mềm như vậy cũng không khác mấy với cách lập trình truyền thống. Kiểu sản xuất như vậy là nhân tố chính gây lên sự không phù hợp giữa năng suất phần cứng và năng suất phần mềm.

Do đó, cần có cuộc cách mạng trong việc phát triển phần mềm giống như cuộc cách mạng công nghiệp. Lập trình bằng cách viết mã từng dòng và dựa trên cú pháp nên được thay thế bằng việc kết hợp các thành phần dựa trên giao diện và ngữ nghĩa. Nghĩa là, cách phát triển phần mềm thủ công cần được thay thế bằng phương pháp công nghệ dựa trên "lắp ráp" thành phần. Có nhiều định nghĩa khác nhau về khái niệm thành phần phần mềm [8]:

(1) Một thành phần là một phần của một hệ thống gần như độc lập và có khả năng thay thế. Nó thoả mãn một số chức năng cụ thể trong ngữ cảnh của kiến trúc đã xác định.

(2) Một thành phần phần mềm là một gói ràng buộc động của một hoặc nhiều đoạn chương trình. Nó được quản lý như một đơn vị và được truy nhập thông qua các giao diện được khám phá tại thời gian chạy.

(3) Một thành phần phần mềm là một đơn vị kết hợp các giao diện cụ thể có tính ràng buộc và các phụ thuộc ngữ cảnh cụ thể. Một thành phần phần mềm có thể được triển khai độc lập và ràng buộc với bên thứ ba.

(4) Một thành phần thể hiện việc thực thi một khái niệm nghiệp vụ hoặc tiến trình nghiệp vụ. Nó bao gồm các tài liệu cần thiết để biểu diễn, cài đặt và triển khai để có thể sử dụng lại cho phát triển các hệ thống lớn hơn.

Mặc dù có những quan điểm khác nhau, nhưng nhìn chung có thể xem *thành phần phần mềm là một khối mã đóng gói một tập các chức năng liên quan và có thể kết hợp với các thành phần khác thông qua giao diện*.

Như vậy, một thành phần có thể được cài đặt và thực thi trong môi trường phần mềm xác định. Việc cài đặt bên trong của một thành phần thường ẩn với người dùng và có thể được sử dụng lại như một thực thể trọn vẹn trong các ngữ cảnh khác nhau.

Các công nghệ thành phần phù hợp với cách hiểu như trên bao gồm JavaBeans, EJB (Enterprise Java Beans) của Sun Microsystems, COM (Component Object Model), DCOM (Distributed Component Object Model), các thành phần .Net của Microsoft và các thành phần CORBA (Common Object Request Broker Architecture) của nhóm Object Management Group.

4.2 PHÁT TRIỂN PHẦN MỀM DỰA TRÊN THÀNH PHẦN

Cách tiếp cận thành phần cho phép gộp một tập chức năng thành một bộ phận tách biệt với các chức năng khác để dễ cài đặt và triển khai. Phần này trình bày một số hướng dẫn để xây dựng thành phần và mô tả các kiểu thành phần tương ứng với mỗi tầng ứng dụng trong thiết kế phân tầng như trình bày trong Chương 2.

4.2.1 Hướng dẫn thiết kế thành phần

Khi thiết kế thành phần trong một ứng dụng cụ thể, chúng ta cần tham khảo một số hướng dẫn sau đây [8]:

- Áp dụng các nguyên lý thiết kế SOLID cho thiết kế lớp vào trong thành phần. Nguyên lý SOLID cho thiết kế lớp có thể tóm lược như sau:
    - Nguyên lý đơn nhiệm: Một lớp chỉ nên gán một trách nhiệm
    - Nguyên lý đóng/mở: Các lớp phải mở rộng được mà không đòi hỏi sửa đổi
    - Nguyên lý thay thế Liskov: Các kiểu con phải thay thế được cho các kiểu cơ bản
    - Nguyên lý giao tiếp tách biệt: Các lớp nên có những giao tiếp riêng cho những đối tác có yêu cầu khác nhau
    - Nguyên lý đảo phụ thuộc: Phụ thuộc giữa các lớp nên được thay thế bởi các lớp trừu tượng để cho phép thiết kế kiểu top-down.
- Các thành phần thiết kế phải kết hợp chặt chẽ: Không nên gộp các chức năng không liên quan vào trong một thành phần. Ví dụ, nên tránh gộp logic truy nhập dữ liệu và logic nghiệp vụ vào trong thành phần nghiệp vụ.
- Thành phần không nên dựa vào chi tiết bên trong các thành phần khác. Mỗi thành phần hay đối tượng khi gọi một phương thức của thành phần hay đối tượng khác thì phương thức đó cần có thông tin về cách xử lý yêu cầu đó hay biết cách chuyển yêu cầu này đến thành phần xử lý phù hợp.
- Hiểu cách giao tiếp giữa các thành phần với nhau. Điều này yêu cầu hiểu được kịch bản triển khai trong ứng dụng nghĩa là phải xác định được giao tiếp qua biên các tầng vật lý hay biên tiến trình hay tất cả chạy trong cùng một tiến trình.
- Đảm bảo mã xuyên tầng như an toàn, giao tiếp hay quản lý login…là trừu tượng với logic ứng dụng. Việc gộp mã cài đặt các chức năng này với logic thành phần có thể làm khó việc mở rộng và bảo trì sau này.
- Áp dụng các nguyên lý của kiến trúc dựa trên thành phần như tái sử dụng được, thay thế được, mở rộng được…như đã trình bày trong Chương 2.

4.2.2 Phân bố thành phần theo tầng

Mỗi tầng ứng dụng tương ứng với các thành phần cài đặt chức năng của tầng đó. Để dễ sử dụng lại và bảo trì sau này những thành phần trên mỗi tầng cần phải cấu trúc sao cho có kết hợp chặt chẽ và kết nối lỏng lẻo.

Các thành phần tầng trình diễn

Các thành phần tầng trình diễn cài đặt chức năng cho phép người dùng tương tác với ứng dụng. Một số kiểu thành phần sau đây thường được sử dụng trong tầng trình diễn:

- Thành phần giao diện người dùng: Giao diện người dùng cho một ứng dụng được đóng gói trong các thành phần giao diện dùng (UI: user interface). UI là những phần trực quan để hiển thị và nhận thông tin từ người dùng. Thành phần này nên chứa càng ít logic ứng dụng càng tốt vì nó có thể ảnh hưởng đến bảo trì và sử dụng lại cũng như kiểm chứng cơ bản.
- Thành phần logic trình diễn: Logic trình diễn là mã ứng dụng xác định hành vi logic và cấu trúc ứng dụng độc lập với cài đặt giao diện người dùng. Các thành phần logic trình diễn có thể chia thành hai loại sau:
    - Các thành phần mô hình trình diễn, khung nhìn, điều khiển
    - Các thành phần thực thể trình diễn: Các thành phần này đóng gói logic nghiệp vụ và dữ liệu để làm dễ dàng cho hiển thị UI và thành phần logic trình diễn. Ví dụ, để tích hợp dữ liệu từ nhiều nguồn.

Các thành phần tầng dịch vụ

Một ứng dụng có thể có tầng dịch vụ để tương tác với khách hàng hay hệ thống khác. Vai trò tầng dịch vụ là cung cấp cho các client/ứng dụng khác cách truy nhập logic nghiệp vụ trong ứng dụng và sử dụng chức năng của ứng dụng bằng cách gửi thông điệp đi hay nhận qua kênh giao tiếp. Các kiểu thành phần sau đây thường được cài đặt trong tầng dịch vụ:

- Giao diện dịch vụ: Dịch vụ thể hiện một giao diện qua đó gửi các thông điệp đi. Giao diện này có thể xem như một giao diện một cửa (façade) thể hiện logic nghiệp vụ được cài đặt trong ứng dụng.
- Kiểu thông điệp: Khi trao đổi dữ liệu qua tầng dịch vụ, các cấu trúc dữ liệu được bao phủ bởi các cấu trúc thông điệp với hỗ trợ các kiểu thao tác khác nhau.

Các thành phần tầng nghiệp vụ

Các thành phần tầng nghiệp vụ cài đặt các chức năng chủ yếu của hệ thống và đóng gói logic nghiệp vụ liên quan. Các kiểu thành phần sau đây thường được cài đặt trong tầng nghiệp vụ:

- Façade ứng dụng: Thành phần này cung cấp giao diện cho các thành phần logic nghiệp vụ và thường là kết hợp nhiều thao tác nghiệp vụ thành một thao tác đơn. Điều này khiến cho nó dễ dàng sử dụng logic nghiệp vụ và giảm được phụ thuộc vì các cuộc gọi bên ngoài không cần biết chi tiết của các thành phần nghiệp vụ và quan hệ của chúng.
- Các thành phần logic nghiệp vụ: Logic nghiệp vụ là một logic ứng dụng liên quan đến truy xuất, xử lý, biến đổi và quản lý dữ liệu ứng dụng cũng như chính sách, quy tắc nghiệp vụ và bảo đảm tính phi mâu thuẫn và đúng đắn của dữ liệu. Để có thể sử dụng lại sau này, các thành phần logic nghiệp vụ không nên chứa bất kỳ hành vi hay logic ứng dụng hơi cá biệt với ca sử dụng hay kịch bản nào đó. Các thành phần logic nghiệp vụ có thể chia thành hai loại sau đây:

- Các thành phần luồng nghiệp vụ: Sau khi thành phần UI nhận đầu vào từ người dùng và chuyển cho tầng nghiệp vụ, ứng dụng có thể sử dụng đầu vào này để thực thi tiến trình nghiệp vụ. Mỗi tiến trình nghiệp vụ thường có nhiều bước phải được tiến hành theo thứ tự nhất định và có thể tương tác với nhau qua bộ phận điều phối. Các thành phần luồng nghiệp vụ xác định các tiến trình có liên quan và phối hợp các tiến trình này để thực thi các chức năng yêu cầu. Hiện nay có nhiều công cụ hỗ trợ quản lý tiến trình công việc, ví dụ Eclipse: https://eclipse.org/jwt/
- Các thành phần thực thể nghiệp vụ: Các thực thể nghiệp vụ hay đối tượng nghiệp vụ đóng gói logic nghiệp vụ và dữ liệu cần thiết để biểu diễn các phần tử thế giới thực như Khách hàng hay Đơn đặt hàng. Có nhiều trường hợp các thực thể nghiệp vụ phải truy nhập được vào những thành phần và dịch vụ trong cả hai tầng nghiệp vụ và tầng dữ liệu. Ví dụ, các thực thể nghiệp vụ có thể ánh xạ thành lược đồ cơ sở dữ liệu và truy nhập được bởi các thành phần nghiệp vụ.

Các thành phần tầng dữ liệu

Các thành phần tầng dữ liệu cung cấp cách truy nhập dữ liệu nằm trên các biên của hệ thống và dữ liệu được đưa ra bởi các hệ thống mạng khác. Các kiểu thành phần sau đây thường được cài đặt trong các tầng dữ liệu:

- Thành phần truy nhập dữ liệu: Các thành phần này trừu tượng hóa logic của yêu cầu truy nhập cơ sở dữ liệu. Phần lớn việc truy nhập dữ liệu đòi hỏi một logic chung mà có thể được tách và cài đặt thành thành phần hỗ trợ riêng biệt hay trong khung hỗ trợ phù hợp. Điều này có thể làm giảm độ phức tạp các thành phần truy nhập dữ liệu và làm giản đơn việc bảo trì sau này.
- Tác nhân dịch vụ: Khi một thành phần nghiệp vụ phải sử dụng chức năng cung cấp bởi dịch vụ bên ngoài, chúng ta cần cài đặt phần mã gọi là tác nhân dịch vụ để quản lý ngữ nghĩa truyền thông với dịch vụ đó.

Các thành phần xử lý chung

Nhiều công việc được thực thi bởi mã chương trình ứng dụng trên nhiều tầng khác nhau. Thành phần xử lý chung cài đặt các chức năng có thể truy nhập những thành phần trong bất kỳ tầng nào. Những kiểu quen thuộc của thành phần này bao gồm:

- Thành phần cài đặt an toàn: Bao gồm những thành phần thực hiện xác thực, định danh…
- Thành phần cài đặt tác vụ quản lý: Những thành phần như chính sách xử lý ngoại lệ, đăng nhập, theo dõi…
- Thành phần cài đặt giao tiếp: Bao gồm những thành phần giao tiếp với những dịch vụ và ứng dụng khác.

4.3 CƠ SỞ HẠ TẦNG CỦA PHÁT TRIỂN PHẦN MỀM HƯỚNG THÀNH PHẦN

Không thể có được một định nghĩa thống nhất về thành phần phù hợp với những ngữ cảnh khác nhau do các thành phần luôn được kết nối với cơ sở hạ tầng tương ứng. Các công nghệ thành phần khác nhau có cơ sợ hạ tầng thành phần khác nhau và do đó có những định nghĩa về thành phần khác nhau.

Cơ sở hạ tầng thành phần là gì? Cơ sở hạ tầng thông thường được hiểu là những cấu trúc và phương tiện cơ bản cần thiết để một quốc gia hoặc một tổ chức hoạt động một cách hiệu quả. Ví dụ như các tòa nhà, giao thông, nước, các nguồn năng lượng và các hệ thống quản lý. Một cơ sở hạ tầng thành phần cũng được hiểu là khung làm việc (framework) hay những công cụ cơ bản cho việc xây dựng và quản lý thành phần. Nó bao gồm ba mô hình:

- *Mô hình thành phần:* Xác định một thành phần khi nào là hợp lệ và làm thế nào để tạo ra một thành phần mới trong cơ sở hạ tầng của thành phần. Các kỹ sư xây dựng các thành phần để có thể tái sử dụng theo mô hình thành phần. Mỗi cơ sở hạ tầng thành phần có một thư viện gồm các thành phần có thể tái sử dụng.
- *Mô hình kết nối:* Mô hình kết nối xác định một tập các kết nối và các công cụ hỗ trợ cho việc kết nối thành phần. Như vậy, mô hình kết nối đưa ra cách để xây dựng một ứng dựng hay một thành phần lớn hơn dựa trên các thành phần có sẵn.
- *Mô hình triển khai:* Mô tả cách đặt các thành phần vào một môi trường làm việc.

Cơ sở hạ tầng thành phần đôi khi còn được gọi là *công nghệ thành phần* hay *kiến trúc thành phần*. Thuật ngữ cơ sở hạ tầng thành phần phù hợp với khái niệm lập trình hướng thành phần. Các công nghệ thành phần phù hợp với định nghĩa trên gồm JavaBeans, EJB của Sun Microsystems, COM (Component Object Model) và DCOM (Distributed COM) của Microsoft và CORBA (Common Object Request Broker Architecture) của OMG (Object Management Group).

COM đưa ra một khung làm việc để tạo và sử dụng các thành phần trên nền tảng Windows. Nó hỗ trợ khả năng tương tác và tái sử dụng các đối tượng phân tán bởi cho phép người phát triển xây dựng hệ thống thông qua việc tập hợp các thành phần có thể tái sử dụng từ những nhà cung cấp khác nhau với giao tiếp thông qua COM. Nó định nghĩa một giao diện lập trình ứng dụng (API) để tạo ra các thành phần dùng trong việc tích hợp các ứng dụng truyền thống hoặc cho phép các thành phần khác nhau tương tác với nhau. Các thành phần này chỉ cần gắn với cấu trúc nhị phân đặc tả bởi Microsoft. DCOM mở rộng COM để các thành phần có thể tương tác với nhau trên môi trường mạng.

CORBA là một đặc tả cấu trúc chuẩn để các nhà cung cấp phát triển các sản phẩm ORB (Object Request Broker) nhằm hỗ trợ tính khả chuyển và tính tương tác của ứng dụng với các ngôn ngữ lập trình, nền tảng phần cứng, hệ điều hành và các cài đặt ORB khác nhau. Các cài của CORBA có sẵn trên thị trường, bao gồm các cài đặt từ các nhà sản xuất máy tính lớn và các nhà cung cấp phần mềm độc lập.

Mỗi thành phần cơ sở hạ tầng được hỗ trợ bởi một công cụ xây dựng thành phần hoặc một môi trường phát triển tích hợp (IDE). Ví dụ, BDK (Bean Development Kit) phát triển bởi Sun

đã thiết kế môi trường cho JavaBeans hỗ trợ thành phần cơ sở hạ tầng của Javabeans. Microsoft .NET hỗ trợ thành phần cơ sở hạ tầng của .NET. JES (Java Embedded Server) 2.0 hỗ trợ thành phần cơ sở hạ tầng OSGi.

Hầu hết các công cụ xây dựng thành phần đều bao gồm việc "thiết kế để sử dụng lại" và "thiết kế với thành phần có sẵn" trong phát triển các thành phần cơ sở. Một số được thiết kế chỉ để cho "thiết kế với tái sử dụng". Ví dụ, BDK1.1 không có chương trình soạn thảo mã nguồn, không có trình biên dịch cũng như các ứng dụng JAR, nên BDK 1.1 không tốt cho việc phát triển thành phần, nhưng lại hỗ trợ cho việc sử dụng lại. Người phát triển phải sử dụng IDE khác để tạo ra một gói Java bean trong một tệp JAR và tải nó vào "Tool Box" của BDK. Ngoài ra, BDK cũng không cung cấp các phương thức chỉnh sửa hoặc các sự kiện chỉnh sửa. Vì vậy, khi sử dụng BDK khó thay đổi được một Java bean hơn là thay đổi các thuộc tính public. Trong điều kiện của toán tử kết nối, BDK cung cấp 2 toán tử để tích hợp các thành phần: thay đổi các thuộc tính và điều khiển sự kiện. Với ý tưởng của một công cụ thực thi, người sử dụng có thể tạo ra một thành phần mới bằng cách mã hóa mã nguồn hoặc bằng cách sửa đổi một thành phần lấy từ một thành phần thư viện.

4.4 BIỂU DIỄN KIẾN TRÚC VÀ THÀNH PHẦN VỚI UML

Mỗi thành phần được thể hiện bởi ba đặc trưng: Mô hình thành phần, mô hình kết nối, mô hình triển khai. Phần này sẽ trình bày biểu diễn các đặc trưng này trong UML để làm cơ sở cho phát triển các nền tảng phần mềm dựa trên thành phần như J2EE và .NET.

4.4.1 Mô hình thành phần

Các biểu đồ thành phần thể hiện cấu trúc và phụ thuộc giữa các thành phần với một số chức năng được nhóm lại và đóng gói theo cấu trúc logic nhất định. Các thành phần thường là các file trong môi trường phát triển và mô hình hóa như sản phẩm theo các loại sau đây:

- Sản phẩm nguồn: là một sản phẩm có ý nghĩa ở thời gian biên dịch. Nó thường là một file mã nguồn chứa một hoặc nhiều lớp.
- Sản phẩm thực thi: là một chương trình thực thi được, nó liên kết các thành phần nhị phân trong thời gian chạy ở dạng tĩnh hoặc động. Một thành phần thực thi đại diện cho các đơn vị thực thi được điều hành bởi một bộ xử lý hay máy tính.

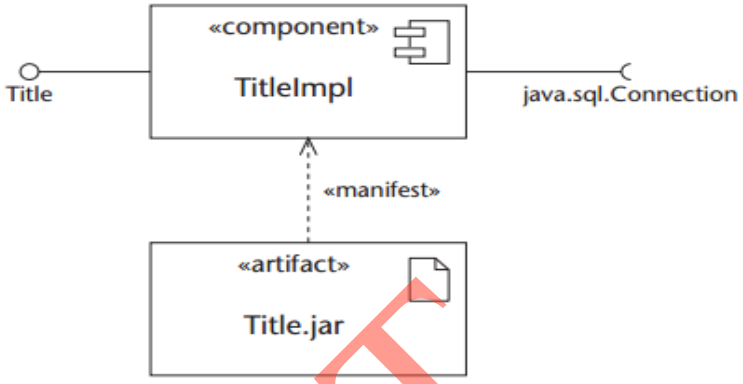
Các nhãn thường được sử dụng để mở rộng nghĩa của sản phẩm tương ứng với công nghệ. Ví dụ với một profile EJB trong J2EE có thể được định nghĩa như một loại <<executable>> với <<EJB>>.

Trong UML một thành phần được biểu diễn bởi một hình chữ nhật với các nhãn <<component>> hoặc một biểu tượng thành phần đặc biệt trong các hình vuông (đây là biểu tượng đã được sử dụng trong các phiên bản trước đó của UML cho các thành phần). Một sản phẩm được thể hiện như một hình chữ nhật với kí hiệu <<artifact>> hoặc một "file biểu tượng" trên góc. Một sản phẩm là một biểu thị vật lý của một thành phần (hoặc một phần tử gói như là các lớp đơn giản).

Một quan hệ phụ thuộc cũng có thể được biểu thị giữa các artifact với ký hiệu là các đường gạch ngang với mũi tên mở, sự phụ thuộc này có nghĩa là một artifact phải cần đến

artifact khác mới có thể hoạt động được. Sự biểu thị này được thể hiện như đường gạch ngang với mũi tên và có ký hiệu <<manifest>>.

Sự phụ thuộc từ một mã nguồn artifact A tới artifact B có ý nghĩa như có sự phụ thuộc ngôn ngữ cụ thể từ A đến B. Trong ngôn ngữ biên dịch, nếu có sự thay đổi trong B sẽ dẫn đến việc A bị biên dịch lại bởi vì ta đang sử dụng B như một thành phần khi biên dịch A. Nếu các artifact có thể chạy được thì các kết nối phụ thuộc có thể được sử dụng để thư viện động của chương trình chạy. Hình 4.7 dưới đây minh họa sự thực hiện của thành phần và artifact tại thời điểm thực thi.



Hình 4.1: Kết hợp khi thành phần thực thi [21]

Trong Hình 4.7 thành phần được kí hiệu là <<component>> và có tên là Titlempl. Điều này có nghĩa là trong hệ thống có một thành phần tên là Titlempl. <<artifact>> là file tên Title.jar. File có đuôi jar thường là file chứa các thư viện.

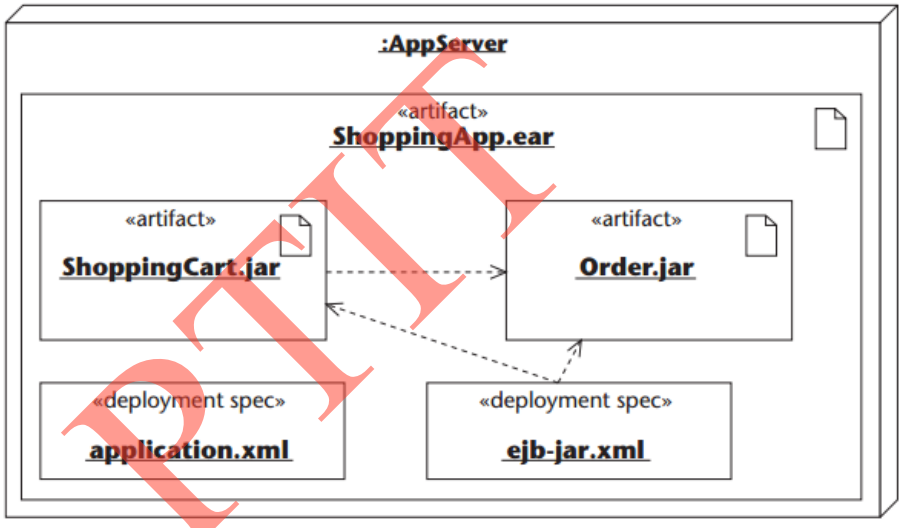
4.4.2 Biểu đồ triển khai

Biểu đồ triển khai minh họa kiến trúc theo thời gian chạy của các thiết bị, môi trường thực thi và các sản phẩm trong kiến trúc. Đó là một mô tả vật lý của cấu hình hệ thống dựa trên mô tả cấu trúc của các đơn vị phần cứng và phần mềm thực thi trong mỗi đơn vị. Nghĩa là trong một cấu trúc có thể xem một đỉnh nào đó có thành phần nào đang thực thi và phần tử logic nào (lớp, đối tượng..) được cài đặt trong thành phần đó. Hơn nữa, có thể duyệt những phần tử nào tương ứng với phân tích qua ca sử dụng của hệ thống.

- Nút: là những tài nguyên tính toán mà những sản phẩm được triển khai để thực hiện. Các tài nguyên này bao gồm các thiết bị như máy tính, thiết bị di động…và cũng bao gồm các nút con trong những thiết bị này như thùng chứa (container) của J2EE, cơ sở dữ liệu…Một nút có kiểu và thể hiện của nó và được biểu diễn bởi hình lập phương ba chiều với tên bên trong nó. Giống như đối tượng, một thể hiện được gạch chân. Khi đỉnh được sử dụng để biểu diễn tài nguyên tính toán, chúng nó được gán nhãn.
- Sản phẩm triển khai: Các sản phẩm có thể triển khai trên các nút. Bằng cách sử dụng cách phân loại hoặc thuật ngữ, tên của các sản phẩm được gạch chân khi hiển thị với người dùng. Một sản phẩm được triển khai vào một nút có thể được thể hiện với một tập hợp các thuộc tính mô tả các thông số thực hiện cho từng artifact trên mỗi nút cụ thể. Tập hợp của các thuộc tính này được mô hình hóa trực tiếp như các artifact được

triển khai hoặc chia ra như các đặc điểm kỹ thuật triển khai. Khi một đặc điểm kỹ thuật triển khai được hiển thị, nó được mô phỏng như một hình chữ nhật phân loại đơn giản với khuôn mẫu <<deployment spec>>. Mối quan hệ phụ thuộc có thể được mô hình hóa giữa các artifact đã được triển khai. Một đặc điểm kỹ thuật triển khai có liên quan đến một artifact được triển khai với một liên kết một chiều. Một đặc điểm kỹ thuật triển khai cũng có thể tồn tại trong một artifact; trong trường hợp này, các biểu tượng cho các yếu tố mô hình được thể hiện trong các biểu tượng đại diện cho artifact được triển khai.

Ví dụ, biểu diễn triển khai một phần hệ bán hàng online được cho trên Hình 4.8. J2EEServer <<execution environment>>, DBServer <<execution environment>>. Artifact là file *ShoppingCartApp.ear* và có các artifact con là các file *ShoppingCart.jar* và *Order.jar* được sử dụng như các file thư viện, các file *application.xml* và file *ejb-jar.xml* là các file biểu hiện cho các đặc điểm kĩ thuật triển khai được viết dưới kí hiệu <<deployment spec>>.



Hình 4.2: Biểu đồ triên khai

4.5 KẾT LUẬN

Chương này đã trình bày chi tiết khái niệm thành phần, những nguyên lý phát triển thành phần và mô hình thành phần. Sau đó cũng đã xem xét những vấn đề liên quan như cách mô hình thành phần, triển khai và kết nối. Nội dung cũng tập trung xem xét việc biểu diễn thành phần và triển khai của thành phần với UML. Trong hai chương tiếp theo, chúng ta sẽ xem xét hai cơ sở hạ tầng của phát triển phần mềm: một là dựa trên J2EE và hai là dựa trên .NET.

BÀI TẬP

1. Phân biệt các khái niệm lớp, thành phần và thể hiện trong biểu đồ UML
2. Khảo sát hai nguyên lý thể hiện kết hợp và kết nối của thành phần. Tham khảo: http://www.jasoncoffin.com/cohesion-and-coupling-principles-of-orthogonal-object-oriented-programming
3. Thể hiện kết hợp và kết nối trong thiết kế phần mềm hướng thành phần. Tham khảo: http://www.codeproject.com/Articles/845616/Cohesion-and-coupling-Principles-of-orthogonal-sca
4. Đo khả năng sử dụng lại của thành phần dựa trên độ kết hợp và kết nối. Tham khảo: http://www.academypublisher.com/jcp/vol04/no09/jcp0409797805.pdf http://www.irisa.fr/lande/lande/icse-proceedings/msr/p18.pdf
5. Trình bày nguyên lý SOLID trong thiết kế phần mềm. Cho ví dụ minh họa. Tham khảo https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design http://www.davesquared.net/2009/01/introduction-to-solid-principles-of-oo.html http://www.c-sharpcorner.com/UploadFile/damubetha/solid-principles-in-C-Sharp/
6. Khảo sát và so sánh những nguyên lý phát triển phần mềm hướng đối tượng và thành phần.
7. Sử dụng Tool như VP để biểu diễn các thành phần của hệ quản lý khách hàng thuê ôtô tự lái.
8. Sử dụng Tool như VP để biểu diễn các thành phần của hệ quản lý khách hàng mua sách online
9. Sử dụng Tool như VP để biểu diễn các thành phần của hệ quản lý đăng ký học theo tín chỉ
10. Sử dụng Tool như VP để biểu diễn các thành phần của hệ quản lý thư viện
11. Sử dụng Tool như VP để biểu diễn các thành phần của hệ quản lý đăng ký mua vé máy bay

CHƯƠNG 5: MÔ HÌNH THÀNH PHẦN VỚI EJB

Mục tiêu của chương này là trình bày:

- J2EE và kiến trúc EJB
- Các khái niệm về thành phần EJB và môi trường chạy của nó
- Các kiểu thành phần, kết nối và việc triển khai EJB
- Giới thiệu các tính năng của EJB 2.x và các tính năng mới của EJB 3.0
- Hướng dẫn xây dựng, triển khai và sử dụng thành phần EJB

5.1. KIẾN TRÚC EJB

5.1.1. Tổng quan về kiến trúc EJB và J2EE platform

Đặc tả Enterprise Java Bean (EJB) do Sun Microsystems Inc công bố vào năm 1998. EJB 2.1 được đề xuất vào năm 2002 là kiến trúc thành phần dành cho việc phát triển và triển khai các ứng dụng phân tán hướng thành phần.

Một thành phần EJB có khả năng sử dụng lại, viết một lần và chạy mọi nơi thường được gọi là WORA (Writing Once, Run Anywhere); có khả năng di động và được biên dịch để có thể triển khai trên bất kỳ server EJB nào như J2EE, Struts và môi trường WebLogic Enterprise…Là một bộ phận của J2EE, công nghệ EJB cung cấp một tập API và các dịch vụ khác nhau để người phát triển tập trung vào logic nghiệp vụ. Nó được thiết kế để hỗ trợ các ứng dụng lớn như các giao dịch thương mại điện tử và các thành phần Web như là công nghệ JSP, JSF và Servlets. Kiến trúc EJB giúp cho việc phát triển các ứng dụng trong doanh nghiệp dễ dàng hơn vì chúng không cần quan tâm đến các dịch vụ mức hệ thống như quản lý giao dịch, quản lý bảo mật, quản lý đa luồng, và các vấn đề quản lý chung khác. Sau đây là một số đặc trưng của EJB:

- Một thành phần EJB có thể được phát triển một lần và sau đó có thể sử dụng lại trong nhiều ứng dụng và triển khai trên nhiều nền tảng khác nhau mà không phải biên dịch và sửa lại mã nguồn.
- Một thành phần EJB là một thành phần phía server cung cấp các dịch vụ cho điều khiển từ xa hoặc client cục bộ, trong khi một java bean là một thành phần phía client được cài đặt và chạy hầu hết ở phía client. Chúng ta có thể có một Java bean phía server nhưng khó có thể cung cấp các dịch vụ cho các client từ xa.
- Một thành phần EJB được chứa bởi container của nó và container được hỗ trợ bởi J2EE hoặc bất kỳ công cụ tuân theo đặc tả J2EE.

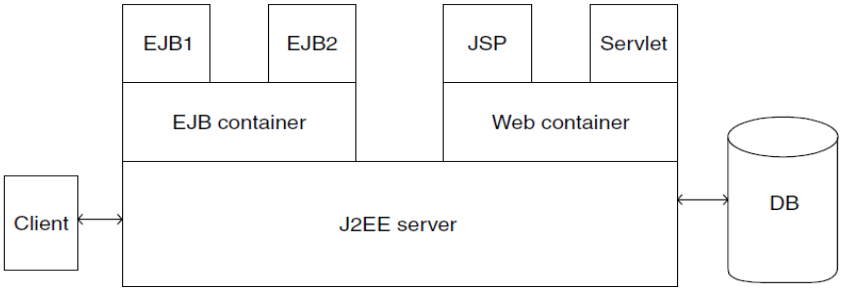
EJB có những đặc trưng khác nhau tùy theo phiên bản phát triển. Để làm sáng tỏ những phát triển về mặt lịch sử, các ví dụ trong chương này sẽ sử dụng cả phiên bản EJB 3.0 và EJB 2.x.

5.1.2. J2EE Server

J2EE server có kiến trúc trong Hình 5.1 và cung cấp một số dịch vụ sau:

- JNDI (Java Naming and Directory In terface) API cho phép các client tìm kiếm và xác định vị trí container chứa các thành phần EJB đã đăng ký.

- J2EE hỗ trợ việc xác thực và an toàn.
- J2EE hỗ trợ dịch vụ HTTP và các dịch vụ liên quan đến EJB, các dịch vụ đa luồng…

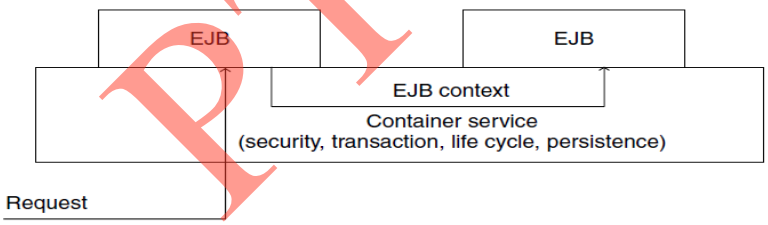


Hình 5.1: Kiến trúc J2EE

5.1.3. Container

Một thể hiện của EJB chạy trên một EJB container. *Container* là môi trường chạy (tập các file class được sinh ra trong quá trình phát triển) điều khiển một thể hiện của thành phần EJB và cung cấp tất cả các dịch vụ quản lý cần thiết cho toàn bộ vòng đời của nó:

- Quản lý giao dịch và an toàn (transaction, security): đảm bảo các đặc tính giao dịch và an toàn của các thực thi giao dịch phân tán.
- Quản lý lưu trữ (persistence): đảm bảo trạng thái của một bean thực thể (entity bean) được sao lưu bởi cơ sở dữ liệu.
- Quản lý vòng đời (life cycle): đảm bảo sự dịch chuyển trạng thái của EJB trong vòng đời của nó.



Hình 5.2: EJB container [21]

EJB container cung cấp một giao diện để thành phần EJB giao tiếp với thế giới bên ngoài. Tất cả các yêu cầu gửi tới thành phần EJB hay các đáp ứng từ thành phần EJB đều phải thông qua container. Container cô lập thành phần EJB để nó không bị truy nhập trực tiếp từ các client bằng cách chặn lời gọi từ client để đảm bảo tính bền vững, tính giao dịch và an toàn của các hoạt động của client trên EJB.

Hình 5.2 chỉ ra rằng container hỗ trợ thành phần EJB và một thành phần cần container để giao tiếp với bên ngoài và để nhận các thông tin cần thiết từ giao diện của nó. EJB container có trách nhiệm tạo ra các đối tượng EJB home, giúp cho việc xác định, tạo và xóa các đối tượng EJB. Giao diện ngữ cảnh EJB cung cấp bởi EJB container đóng gói các thông tin liên quan về môi trường của container như là định danh của một thành phần EJB, các trạng thái của giao dịch và tham chiếu từ xa tới EJB.

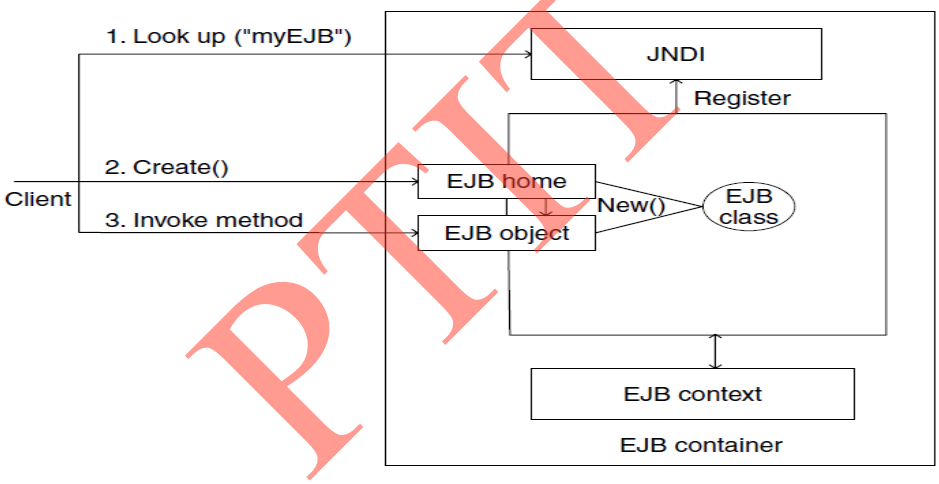
5.1.4. Thành phần EJB

Một enterprise bean là một thành phần phân tán nằm trong một EJB container và được truy cập bởi client từ trên mạng thông qua giao diện từ xa hoặc được truy nhập thông qua enterprise bean khác trên cùng server thông qua giao diện cục bộ. Thành phần EJB là một thành phần có khả năng thực thi từ xa được triển khai trên server của nó và có khả năng tự mô tả thể hiện bởi mô tả triển khai (DD: Deployment Descriptor) có định dạng XML.

5.2. MÔ HÌNH THÀNH PHẦN CỦA EJB

5.2.1. Tổng quan về EJB 2.x

Mỗi thành phần EJB có một giao diện logic nghiệp vụ được tạo ra bởi thành phần đó nên các client có thể truy cập vào các thao tác logic nghiệp vụ thông qua giao diện này mà không cần phải biết cài đặt chi tiết đằng sau giao diện đó. Một thể hiện của thành phần EJB được tạo ra và quản lý thông qua giao diện cục bộ của nó (home interface) với EJB container. Mỗi enterprise bean phải có một giao diện cục bộ và một giao diện từ xa. Thành phần EJB có thể được cấu hình tại thời gian triển khai bằng đặc tả DD của nó. Hình 5.3 mô tả cấu trúc của một thành phần EJB và quá trình tương tác giữa một client với nó.



Hình 5.3: Tương tác giữa client và thành phần EJB

Lớp EJB (EJB class) đằng sau các giao diện cục bộ và từ xa được thiết kế để thực thi hai giao diện. Một thành phần EJB là một hộp đen nghĩa là client của nó chỉ biết tương tác với thành phần nào chứ không biết cách hoạt động của nó. Client tạo một yêu cầu tới thành phần EJB với tên được triển khai của nó bằng việc tìm kiếm trong JNDI để lấy tham chiếu đối tượng của thành phần EJB. Client sau đó có thể tạo một thể hiện của thành phần EJB này trên server theo tham chiếu đối tượng. Cuối cùng, client gọi các phương thức của thể hiện EJB này. Tất nhiên một EJB phải đăng kí với JNDI để các client có thể tìm được nó.

Các Enterprise bean là các thành phần có thể được nhúng trong nhiều ứng dụng khác nhau mà không cần phải dịch lại hoặc thay đổi mã nguồn của chúng. Chúng có thể được triển khai trong nhiều máy chủ theo chuẩn EJB. Mô hình EJB hỗ trợ những loại bean sau:

- Bean phiên phi trạng thái (Stateless session bean) thực thi nhiều logic nghiệp vụ khác nhau, như phiên dịch ngôn ngữ, đăng nhập, tính toán thuế và chuyển đổi tiền tệ. Nó được đóng gói trong một dịch vụ Web. Bất kì một bean đang tồn tại nào cũng có thể

đóng gói trong một dịch vụ Web bên ngoài bằng một tài liệu dịch vụ web WSDL mô tả điểm cuối dịch vụ Web mà bean đó thực thi. Những bean đặc biệt như vậy dùng cho các dịch vụ Web không cung cấp các giao diện mà một EJB thành phần thường cung cấp.

- Bean phiên trạng thái (Stateful session bean) cũng đảm trách cùng một vai trò như các bean phiên phi trạng thái ngoại trừ chúng theo vết các trạng thái của việc giao tiếp giữa các client tới các EJB thành phần. Ví dụ, một shopping cart bean có thể là một bean phiên giao dịch có trạng thái điển hình.

Bất cứ một bean phiên nào cho dù là có trạng thái hay phi trạng thái cũng không hỗ trợ các yêu cầu lưu trữ cho một bean thực thể trên nền tảng kiến trúc thành phần EJB:

- MDB (message – driven bean) là bean theo hướng thông điệp nhằm biểu diễn một loại thành phần EJB mới làm việc trong chế độ giao tiếp bất đồng bộ giống như mô hình ủy quyền hướng sự kiện trong Java.
- BMP (Bean Managed Persistence) là các bean thực thể (entity bean) trong việc quản trị lưu trữ dữ liệu.
- CMP (Container Managed Persistence) là các bean thực thể trong đó việc quản trị lưu trữ lâu dài của chúng và được đặc tả bởi công cụ triển khai và được quản lý bởi container. Tuy nhiên, các bean thực thể CMP không cần xử lý bất kì việc truy xuất cơ sở dữ liệu SQL nào mà do một bean thực thể được phục hồi bởi cơ sở dữ liệu quan hệ.

Giao diện từ xa của một thành phần EJB thực thi giao diện *javax.ejb.EJBObject* sau là giao diện *java.rmi.Remote*. Giao diện cục bộ của một thành phần EJB thực thi giao diện *javax.ejb.EJBHome*, sau đó lại thực thi giao diện *java.rmi.remote*. Giao diện cục bộ thực thi giao diện *javax.ejb.EJBLocalObject* và giao diện *javax.ejb.EJBLocalHome.* Giao diện cục bộ được sử dụng bởi thành phần EJB khác chạy trên cùng một server để truy nhập nên nó có thể giảm chi phí gây ra bởi việc truy cập từ xa. Giao diện từ xa đem lại sự độc lập vị trí nhưng đắt hơn còn các giao diện cục bộ tạo lời gọi có hiệu quả hơn. Điểm khác biệt quan trọng giữa các giao diện cục bộ và từ xa đó là việc gọi phương thức trong giao diện cục bộ sử dụng việc truyền bằng tham chiếu còn việc gọi giao diện ở xa sử dụng việc truyền bằng giá trị.

5.2.2. EJB 3.0

Một số đặc trưng mới của EJB 3.0:

- Định nghĩa các đánh dấu siêu dữ liệu để chú giải các ứng dụng EJB. Các đánh dấu siêu dữ liêu này làm đơn giản hoá công việc của những nhà phát triển, giảm số lớp và giao diện cần phải cài đặt và người phát triển không cần cung cấp một file mô tả triển khai.
- Đóng gói các phụ thuộc môi trường và truy cập JNDI thông qua việc sử dụng các đánh dấu, các cơ chế kích hoạt các phụ thuộc và các cơ chế tìm kiếm đơn giản.
- Không cần thiết các giao diện cho bean phiên. Giao diện nghiệp vụ cho một bean phiên có thể là một java interface đơn thuần không cần phải là một interface *EJBObject, EJBLocalObject,* hoặc *java.rmi.Remote.*

- Loại bỏ các yêu cầu cho giao tiếp cục bộ của bean phiên. Đơn giản hoá các bean thực thể. Hỗ trợ mô hình hoá miền, cung cấp kế thừa và đa hình.
- Loại bỏ tất cả các giao diện cho thực thể lưu trữ. Thể hiện đặc tả các đánh dấu siêu dữ liệu và mô tả triển khai với XML cho việc ánh xạ các quan hệ của thực thể lưu trữ.

5.2.3 Bean phiên

Một bean phiên (sesion bean) biểu diễn một client đơn bên trong Server ứng dụng. Để truy xuất một ứng dụng được triển khai trên server, client gọi các phương thức của bean phiên. Nghĩa là nhằm tránh cho client phải thực hiện các tác vụ phức tạp, các bean phiên sẽ thực hiện tác vụ này trên server. Bean phiên giống như một phiên giao dịch, nó chỉ đại diện cho client và khi client ngắt, bean phiên tương ứng cũng bị ngắt. Dữ liệu không được lưu vào cơ sở dữ liệu. Có ba kiểu bean phiên đó là bean trạng thái và bean phiên phi trạng thái:

- **Bean phiên trạng thái:** Trạng thái của một đối tượng là bộ giá trị của các biến. Trong một bean phiên trạng thái, các giá trị biểu diễn trạng thái của một bean tương ứng duy nhất một client. Vì client tương tác với bean tương ứng với nó, nên trạng thái này thường được gọi là trạng thái đàm thoại. Trạng thái này được duy trì trong suốt phiên làm việc giữa client và bean. Nếu client loại bỏ bean hoặc ngắt, phiên kết thúc và các trạng thái sẽ mất.
- **Bean phiên phi trạng thái:** Bean này không duy trì trạng thái đàm thoại với client. Khi một client gọi một phương thức của một bean phi trạng thái, các biến của bean có thể chứa một trạng thái cụ thể cho một client, nhưng chỉ trong thời gian thực hiện cuộc gọi. Khi phương thức hoàn thành, trạng thái này sẽ bị mất. Tuy nhiên, các client có thể thay đổi trạng thái của các bean và giữ cho lời gọi tiếp theo. Vì bean phi trạng thái có thể hỗ trợ cho nhiều client, nên chúng linh hoạt hơn cho các ứng dụng đòi hỏi số client lớn. Bean hiên phi trạng thái có thể cài đặt dịch vụ web nhưng bean phiên trạng thái thì không.
- **Bean phiên đơn nhất:** Bean này được khởi tạo một lần và tồn tại suốt vòng đời một ứng dụng. Bean này được thiết kế để dùng chung và truy nhập bởi nhiều client. Khác với bean phi trạng thái, chỉ có một bean phiên đơn nhất cho mỗi ứng dụng và bean này cũng có thể cài đặt dịch vụ web.

Ví dụ 5.1: Sử dụng bean phiên phi trạng thái để thực hiện việc chuyển đổi độ F sang độ C và ngược lại.

```
//remote interface: Converter
package converter.ejb;
import javax.ejb.Remote;
@Remote
public interface Converter {
    public double cToF(double c);
    public double fToC(double f);
```

```
}
//stateless session bean: ConverterBean
package converter.ejb;
import javax.ejb.Stateless;
@Stateless
public class ConverterBean implements Converter {
    @Override
    public double cToF(double c) {
        return c * 9/5 + 32;
    }
    @Override
    public double fToC(double f) {
        return (f - 32) * 5/9;
    }
}
```

5.2.4. Thành phần dịch vụ EJB

Một dịch vụ client có thể truy xuất một ứng dụng J2EE theo 2 cách. Một là, client có thể truy xuất một web service được tạo với JAX-WS. Hai là, một web service client có thể gọi phương thức của một bean phiên phi trạng thái. Web service sử dụng các giao thức SOAP, HTTP, WSDL nên bất kỳ client (viết bằng ngôn ngữ Java hoặc một ngôn ngữ khác) của dịch vụ web nào cũng có thể truy cập một bean phiên phi trạng thái. Client không cần biết công nghệ cài đặt các service là bean phiên phi trạng thái, JAX – WS hoặc các công nghệ khác. Tính linh hoạt này cho phép người phát triển tích hợp các ứng dụng J2EE với dịch vụ web.

Một client của dịch vụ web có thể truy xuất một bean phiên phi trạng thái thông qua lớp cài đặt dịch vụ web của bean. Tất cả các phương thức public trong bean có thể được truy cập bởi các client dịch vụ web. Ví dụ HelloService sau đây là một bean phiên phi trạng thái cho dịch vụ.

Ví dụ 5.2: Bean phiên phi trạng thái với dịch vụ web

```
Package ejb.service;
import javax.ejb.Stateless;
import javax.jws.WebMethod;
import javax.jws.WebService;

@Stateless
@WebService
public class HelloServiceBean {
    private String message = "Hello, ";
    public void HelloServiceBean() {
    }
    @WebMethod
```

```
    public String sayHello(String name) {
        return message + name + ".";
    }
}
```

5.2.5 Bean thực thể

Một bean thực thể (entity bean) thể hiện dữ liệu được sao lưu bởi cơ sở dữ liệu. Ví dụ, trong Hệ Quản lý Học tập theo tín chỉ thì Sinh viên, Giảng viên và các Khóa học là những bean thực thể. Mỗi bean thực thể tương ứng với một bảng trong cơ sở dữ liệu và mỗi thể hiện của bean được lưu giữ trong một hàng của bảng. Mỗi thực thể cung cấp cách truy cập và được hỗ trợ bởi dịch vụ trong EJB qua container của nó. Nó có một khóa chính duy nhất để cho các client xác định được vị trí mình. Sau đây chúng ta sẽ xem xét cách tạo dựng bean thực thể qua hai phiên bản của Java.

Ví dụ 5.3: Với EJB 2.x, có hai loại entity bean: BMP (entity bean Bean Managed Persistence) và CMP (Container Managed Persistence). Sau đây là một ví dụ tạo bean thực thể BMP sinh viên Student (trích từ [21]). Một bảng cơ sở dữ liệu được tạo bằng SQL :

```
CREATE TABLE student(id VARCHAR(3) CONSTRAINT pk_student PRIMARY
KEY,gpa Number(2,1));
//The following is the home interface for this student BMP
//entity bean.
import javax.ejb.*;
import java.util.*;

public interface StudentHome extends EJBHome {
    public Student create(String id, double gpa) throws
RemoteException, CreateException;
    public Account findByPrimaryKey(String id) throws
FinderException, RemoteException;
}
//The following code is the remote interface for this student //BMP
entity bean.
import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Student extends EJBObject {
    public double getGpa() throws RemoteException;
}
//The following is the BMP implementation entity bean where SQL
//statements are explicitly included.
import java.sql.*;
import javax.sql.*;
import java.util.*;
import javax.ejb.*;
import javax.naming.*;

public class StudentEJB implements EntityBean {
    private String id;
    private double gpa;
    private EntityContext context;
```

```
    private Connection con;
    private String dbName = "java:comp/env/jdbc/StudentDB";
    public double getGpa() {
        return gpa;
    }
    //The following ejb callback methods are executed by EJB
    //container. The Detailed references are in this chapter
    //reference.
    //When a new bean instance is created the method ejbCreate()
    // is automatically called by the container to insert a row in a
    //corresponding table in the database.
public String ejbCreate(String id, double gpa) throws
CreateException {
        try {
            insertRow(id, gpa);
        } catch (Exception ex) {
            throw new EJBException("ejbCreate: ");
        }
        this.id = id;
        this.gpa = gpa;
        return id;
    }
public String ejbFindByPrimaryKey(String primaryKey) throws
FinderException {
        boolean result;
        try {
            result = selectByPrimaryKey(primaryKey);
        } catch (Exception ex) {
            throw new EJBException("ejbFindByPrimaryKey: ");
        }
        if (result) {
            return primaryKey;
        }
        else {
            throw new ObjectNotFoundException
            ("Row for id " + primaryKey + " not found.");
        }
    }

public void ejbRemove() {
        try {
            deleteRow(id);
        } catch (Exception ex) {
            throw new EJBException("ejbRemove: ");
        }
    }

public void setEntityContext(EntityContext context) {
        this.context = context;
        try {
           makeConnection();
```

```
        } catch (Exception ex) {
           throw new EJBException("Failed to connect todatabase.”);
        }
    }

public void ejbActivate() {
        id = (String)context.getPrimaryKey();
    }

    public void ejbPassivate() {
        id = null;
    }

public void ejbLoad() {
        try {
            loadRow();
        } catch (Exception ex) {
            throw new EJBException("ejbLoad: ");
        }
    }

public void ejbStore() {
        try {
            storeRow();
        } catch (Exception ex) {
            throw new EJBException("ejbLoad: " );
        }
    }

public void ejbPostCreate(String id, double gpa) { }
    void makeConnection() throws NamingException, SQLException {
        InitialContext ic = new InitialContext();
        DataSource ds = (DataSource) ic.lookup(dbName);
        con = ds.getConnection();
    }
//The following methods are callback methods to invoke SQL
//statements to access database
void insertRow (String id, double gpa) throws SQLException {
    String insertStatement = "insert into student values(?,? )";
        PreparedStatement                    prepStmt                    =
con.prepareStatement(insertStatement);
        prepStmt.setString(1, id);
        prepStmt.setDouble(2, gpa);
        prepStmt.executeUpdate();
        prepStmt.close();
    }
void deleteRow(String id) throws SQLException {
    String deleteStatement = "delete from student where id = ?";
    PreparedStatement
    prepStmt=con.prepareStatement(deleteStatement);
    prepStmt.setString(1, id);
```

```
    prepStmt.executeUpdate();
    prepStmt.close();
    }

boolean selectByPrimaryKey(String primaryKey) throws SQLException {
        String selectStatement="select id "+"from student where id
=? ";
        PreparedStatement                    prepStmt                =
con.prepareStatement(selectStatement);
        prepStmt.setString(1, primaryKey);
        ResultSet rs = prepStmt.executeQuery();
        boolean result = rs.next();
        prepStmt.close();
        return result;
    }

void loadRow() throws SQLException {
        String selectStatement = "select gpa " + "from student where
id = ? ";
        PreparedStatement                    prepStmt                =
con.prepareStatement(selectStatement);
        prepStmt.setString(1, this.id);
        ResultSet rs = prepStmt.executeQuery();
        if (rs.next()) {
            this.gpa = rs.getDouble(2);
            prepStmt.close();
        }
        else {
            prepStmt.close();
            throw new NoSuchEntityException(id + " not found.");
        }
    }

void storeRow() throws SQLException {
    String updateStatement = "update student set gpa = ? " + "where
id = ?";
        PreparedStatement                    prepStmt                =
con.prepareStatement(updateStatement);
        prepStmt.setDouble(1, gpa);
        prepStmt.setString(2, id);
        int rowCount = prepStmt.executeUpdate();
        prepStmt.close();
        if (rowCount == 0) {
            throw new EJBException("Store id " + id + " failed.");
        }
    }
}
```

Ví dụ 5.4: Với JavaEE 5 cũng như các phiên bản sau này có thể sử dụng JPA (Java Persistent API) để tạo ra các thực thể để ánh xạ vào các bảng trong cở sở dữ liệu. Ví dụ dưới đây nêu lên cách tạo một bean thực thể BookEntity và một bean phiên phi trạng thái BookBean có các phương thức tạo, xoá, sửa, tìm kiếm thông tin sách trong cơ sở dữ liệu.

```
//EntityBean: BookEntity
package ejb.book;

import java.io.Serializable;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.NamedQuery;

@Entity
@NamedQuery(name = "findAllBooks", query = "SELECT b FROM Book b")
public class BookEntity implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Column(nullable  = false)
    private String title;
    private float price;
    @Column(length = 2000)
    private String description;
    private String isbn;
    private int nbOfPage;
    private boolean illustrations;

    public BookEntity() {
    }

    public BookEntity(Long id, String title, float price, String
description, String isbn, int nbOfPage, boolean illustrations) {
        this.id = id;
        this.title = title;
        this.price = price;
        this.description = description;
        this.isbn = isbn;
        this.nbOfPage = nbOfPage;
        this.illustrations = illustrations;
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getDescription() {
```

```
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public boolean isIllustrations() {
        return illustrations;
    }

    public void setIllustrations(boolean illustrations) {
        this.illustrations = illustrations;
    }

    public String getIsbn() {
        return isbn;
    }

    public void setIsbn(String isbn) {
        this.isbn = isbn;
    }

    public int getNbOfPage() {
        return nbOfPage;
    }

    public void setNbOfPage(int nbOfPage) {
        this.nbOfPage = nbOfPage;
    }

    public float getPrice() {
        return price;
    }

    public void setPrice(float price) {
        this.price = price;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    @Override
    public int hashCode() {
        int hash = 0;
        hash += (id != null ? id.hashCode() : 0);
```

```
        return hash;
    }

    @Override
    public boolean equals(Object object) {
        // TODO: Warning - this method won't work in the case the id
fields are not set
        if (!(object instanceof BookEntity)) {
            return false;
        }
        BookEntity other = (BookEntity) object;
        if ((this.id == null && other.id != null) || (this.id !=
null && !this.id.equals(other.id))) {
            return false;
        }
        return true;
    }

    @Override
    public String toString() {
        return "com.me.bookws.Book[id=" + id + "]";
    }
}

//Remote Interface: BookRemote
package ejb.book;

import java.util.List;
import javax.ejb.Remote;

@Remote
public interface BookRemote {

    void create(BookEntity book);

    void edit(BookEntity book);

    void remove(BookEntity book);

    BookEntity find(Object id);

    List<BookEntity> findAll();

    List<BookEntity> findRange(int[] range);

    int count();

}

//Stateless session bean: BookBean
package ejb.book;
```

```
import java.util.List;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;
import javax.persistence.criteria.CriteriaQuery;
import javax.persistence.criteria.Root;

@Stateless
public class BookBean implements BookRemote {
    @PersistenceContext(unitName = "BookWSPU")
    private EntityManager em;

    public void create(BookEntity book) {
        em.persist(book);
    }

    public void edit(BookEntity book) {
        em.merge(book);
    }

    public void remove(BookEntity book) {
        em.remove(em.merge(book));
    }

    public BookEntity find(Object id) {
        return em.find(BookEntity.class, id);
    }

    public List<BookEntity> findAll() {
        CriteriaQuery cq = em.getCriteriaBuilder().createQuery();
        cq.select(cq.from(BookEntity.class));
        return em.createQuery(cq).getResultList();
    }

    public List<BookEntity> findRange(int[] range) {
        CriteriaQuery cq = em.getCriteriaBuilder().createQuery();
        cq.select(cq.from(BookEntity.class));
        Query q = em.createQuery(cq);
        q.setMaxResults(range[1] - range[0]);
        q.setFirstResult(range[0]);
        return q.getResultList();
    }

    public int count() {
        CriteriaQuery cq = em.getCriteriaBuilder().createQuery();
        Root<BookEntity> rt = cq.from(BookEntity.class);
        cq.select(em.getCriteriaBuilder().count(rt));
        Query q = em.createQuery(cq);
        return ((Long) q.getSingleResult()).intValue();
    }

}
```

5.2.6. Bean hướng thông điệp MDB (Message-Driven Beans)

Một bean hướng thông điệp MDB là một enterprise bean cho phép các ứng dụng Java EE xử lý các thông điệp theo cách không đồng bộ. Nó hoạt động như một bộ lắng nghe thông điệp JMS giống bộ lắng nghe sự kiện trừ khi nó nhận các thông điệp JMS theo các sự kiện. Các thông điệp có thể được gửi bởi bất kì thành phần Java EE nào (các ứng dụng client, một enterprise bean khác, thành phần web) hoặc bởi một ứng dụng JMS hoặc hệ thống không sử dụng công nghệ Java. Nghĩa là MDB có thể xử lý các thông điệp JMS hoặc các kiểu thông điệp khác.

MDB có thể làm việc theo chế độ một-một hay quảng bá kiểu một-nhiều. MDB có thể hoạt động theo kiểu không đồng bộ trong đó một thông báo có thể được nhận bởi một MDB thành phần và các đáp ứng của nó có thể ngay lập tức hoặc lâu hơn. Một thành phần MDB hoạt động như sau:

- Container đăng ký thành phần MDB này với JMS
- JMS đăng kí tất cả các đích JMS với JNDI
- EJB container thể hiện thành phần MDB này
- Client tìm kiếm đích với MDB
- Client gửi thông điệp đến đích
- EJB container chọn MDB tương ứng để xử lý gói tin

MDB hoạt động theo chế độ một-nhiều không đồng bộ và các loại thông điệp có thể là tin nhắn văn bản, các tin nhắn đối tượng, tin nhắn dòng, hoặc tin nhắn byte. Các gói tin được truyền và xử lý theo phương thức *MessageListener* được gọi là *onMessage()*. Ví dụ dưới đây xây dựng một bean MDB xử lý các thông điệp văn bản (TextMessage).

Ví dụ 5.5: Bean MDB xử lý các thông điệp văn bản đến

```
//MessageDrivenBean: SimpleMessageBean
package ejb.mdb;
import javax.ejb.MessageDriven;
import javax.ejb.MessageDrivenContext;
import javax.jms.MessageListener;
import javax.jms.Message;
import javax.jms.TextMessage;
import javax.jms.JMSException;
import javax.annotation.Resource;

@MessageDriven(mappedName = "jms/Queue")
public class SimpleMessageBean implements MessageListener {
    @Resource
    private MessageDrivenContext mdc;
    public SimpleMessageBean() {
    }
    @Override
    public void onMessage(Message inMessage) {
        TextMessage msg = null;
        try {
            if (inMessage instanceof TextMessage) {
```

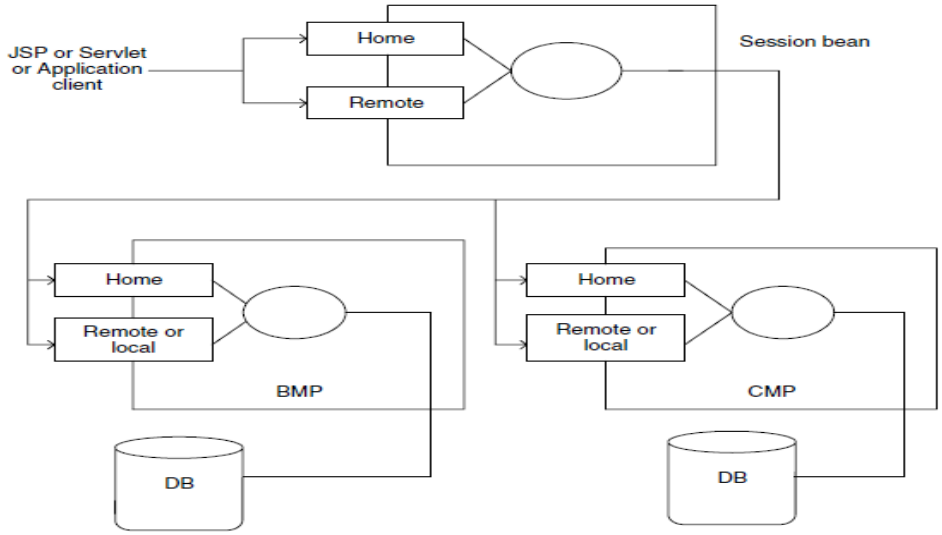
```
                msg = (TextMessage) inMessage;
                System.out.println("MESSAGE      BEAN:      Message
received: " + msg.getText());
            } else {
                System.out.println(
                        "Message of wrong type: "
                        + inMessage.getClass().getName());
            }
        } catch (JMSException e) {
            e.printStackTrace();
            mdc.setRollbackOnly();
        } catch (Throwable te) {
            te.printStackTrace();
        }
    }
}
```

5.3 MÔ HÌNH KẾT NỐI CỦA EJB

Trong phần này, chúng ta sẽ trình bày các cơ chế kết nối giữa các thành phần EJB nghĩa là xem xét cách liên kết các thành phần cơ bản thành khối thành phần mới hoặc các ứng dụng mới. Mô hình kết nối của EJB sẽ hướng dẫn thiết kế các thành phần EJB, các kết nối, và các giao tiếp giữa các thành phần.

Để cho một đối tượng thành phần client EJB trao đổi được với các đối tượng EJB khác, client phải lấy được một đối tượng tham chiếu tới thành phần đích và một thể hiện từ xa (RMI). Điều này có thể được thấy từ trong ví dụ chuyển đổi nhiệt độ.

Có thể có các kết nối đồng bộ hoặc không đồng bộ giữa hai thành phần EJB thông qua giao diện từ xa hoặc cục bộ của các thành phần. Một thành phần EJB cũng có thể được truy nhập bằng thành phần Web, thành phần JavaBean, một thành phần Servlet, hoặc một thành phần JSP. Một thành phần EJB cũng có thể truy cập các đối tượng dữ liệu khác bằng JDBC. Hình 5.4 biểu thị một bean phiên kết nối với hai bean thực thể bởi cơ sở dữ liệu.



Hình 5.4. Kết nối giữa bean phiên và bean thực thể [8]

5.3.1 Kết hợp các kết nối đồng bộ

Một lời gọi đồng bộ thực thi theo mô hình tương tác yêu cầu – đáp ứng. Một client gọi một phương thức từ xa qua giao diện của đối tượng thành phần đích EJB. Giao diện che dấu tất cả việc cài đặt chi tiết các phương thức logic nghiệp vụ của các client. Sau khi đối tượng thành phần EJB đích kết thúc công việc, nó sẽ đáp ứng các yêu cầu bằng việc gửi trả kết quả hoặc có thể yêu cầu các dịch vụ từ các thành phần khác và chờ phản hồi để chuyển tới client.

Một ví dụ điển hình cho kiểu truyền thông này là tạo giỏ hàng. Bean phiên sẽ gửi một yêu cầu tới một bean thực thể và chờ xử lý. Khi một đối tượng thành phần client khởi tạo một yêu cầu, luồng yêu cầu được đóng lại cho đến khi nó nhận được phản hồi từ thành phần EJB đích. Hầu hết các ví dụ chúng ta đã thấy cho đến nay đều hoạt động ở chế độ đồng bộ.

5.3.2 Kết nối lỏng lẻo không đồng bộ

Một truyền thông không đồng bộ gồm truyền thông dựa trên thông điệp giữa một thành phần EJB và client của nó. Một client gửi yêu cầu tới một thành phần EJB nhưng không tự đóng mà là tiếp tục duy trì tiến trình của mình. Có hai hình thức truyền thông không đồng bộ: Quảng bá thông điệp dựa vào hàng đợi và quảng bá dựa vào đăng ký/công bố.

Truyền thông không đồng bộ dựa trên một hàng đợi yêu cầu, một hàng đợi thông điệp với hỗ trợ bởi JMS giữa client và người nhận thông điệp hoặc người dùng thông điệp. Trong khi đó, với phương pháp đăng ký/công bố, người dùng đăng ký chủ đề để cho người cung cấp gửi các thông điệp vào và khi đó một hoặc nhiều người dùng có thể nhận thông điệp từ chủ đề này. Trong truyền thông không đồng bộ, các client và server đều kết nối lỏng lẻo nên dẫn đến thông lượng tốt hơn.

5.3.3 Truyền thông từ xa và cục bộ

Một thành phần client và server của nó có thể nằm trên một máy hoặc trên các máy ảo Java khác nhau. Nếu hai thành phần EJB chạy trên cùng một máy, sẽ có chi phí thấp hơn nhiều bởi sử dụng giao diện cục bộ so với sử dụng giao diện từ xa. Đó là lý do tại sao EJB 2.x thêm các giao diện *localHome* và *localObject* vào giao diện từ xa. Một vài thực thể bean có thể cung cấp cả hai giao diện trong EJB 2.x . Một bean phiên có thể đóng vai trò của bean thực thể và một bean thực thể có thể tương tác trực tiếp với client hoặc thông qua một bean phiên hoặc được kết nối tới bean thực thể khác. Một giao diện cục bộ cho phép tham chiếu tới một phương thức trong mô hình truyền tham chiếu, trong khi một giao diện từ xa thực hiện nó trong mô hình truyền tham trị.

5.3.4 Các tham chiếu đối tượng đến các bean thực thể

Để đạt đến bean thực thể đích, một bean phiên phi trạng thái phải tìm các thành phần được triển khai theo tên đăng kí tại JNDI để lấy tham chiếu đối tượng và có thể qua đó tham chiếu tới các bean khác. Một bean phiên trạng thái có thể gửi tham chiếu tới một bean thực thể. Một bean phiên hoặc một thành phần MDB có thể giữ việc tham chiếu tới thành phần EJB khác. Một bean thực thể BMP có thể giữ một tham chiếu tới bean thực thể khác.

5.3.5 Kết hợp các kết nối giữa các bean thực thể

Với EJB 2.x, mỗi bean thực thể được lưu trữ bởi một bảng quan hệ. Mỗi thể hiện của nó được lưu trong một hàng trong bảng bất kể BMP hay CMP. Các quan hệ giữa các bean thực thể có

thể là một-một, một-nhiều hay nhiều-nhiều, giống như các quan hệ dữ liệu trong cơ sở dữ liệu được thể hiện trong biểu đồ quan hệ-thực thể.

Ví dụ 5.6: Quan hệ giữa bean thực thể sinh viên *Student* và bean thực thể khóa học *Course* là quan hệ nhiều–nhiều. Một sinh viên có thể đăng ký nhiều khóa học và một khóa học có thể có nhiều sinh viên. Thể hiện các bean thực thể này bằng CMP trong kiến trúc EJB 2.x . Giả sử có hai thực thể bean CMP được triển khai tại một container như vậy và sử dụng giao diện cục bộ.

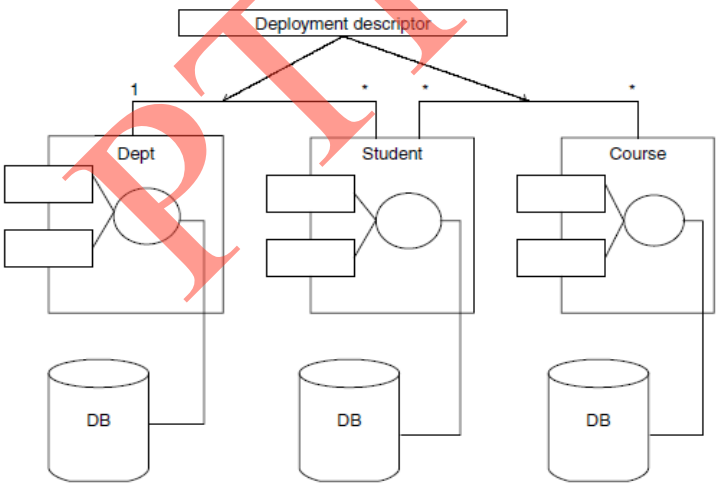
```
package student;

import java.util.*;
import javax.ejb.CreateException;
import javax.ejb.EntityBean;
import javax.ejb.EntityContext;
import javax.naming.Context;
import javax.naming.InitialContext;

public abstract class StudentBean implements EntityBean {
    private EntityContext context;
    //StudentID and StudentName are CMP fields
    public abstract String getStudentID(); //primary key
    public abstract void setStudentID(String id);
    public abstract String getName();
    public abstract void setName(String lastName);
    //CMR(container-managed relationship) fields to course bean
    public abstract Collection getCourses();
    public abstract void setCourses (Collection courses);
    //StudentBean next defines its business logic such that
    //getCourseList()
    //returns all corresponding courses this student has taken and
    //addCourse() will add a new course for this student.
    public ArrayList getCourseList() {
        ArrayList list = new ArrayList();
        Iterator c = getCourses().iterator();
        while (c.hasNext()) {
        list.add(
            (LocalCourse)c.next());
        }
        return list;
    }

    public void addCourse (LocalCourse course) {
        getCourses().add(course);
    }
}

//Student Local Home Interface Here's the LocalStudentHome home
interface for the StudentBean:

import javax.ejb.CreateException;
import javax.ejb.EJBLocalHome;
import javax.ejb.FinderException;

public interface LocalStudentHome extends EJBLocalHome {
    public LocalStudent create (String studentID, String Name)
```

```
throws CreateException;

    public LocalStudent findByPrimaryKey (
        String studentID)throws FinderException;
    }
}
    //The Student bean also defines a local interface. The bean's
    //LocalStudent interface extends the EJBLocalObject interface
    //not the EJBObject interface.

import java.util.ArrayList;
import javax.ejb.EJBLocalObject;
public interface LocalStudent extends EJBLocalObject {
    public String getStudentID();
    public String getName();
    public ArrayList getCourseList();
}
```

Ở đây chúng ta có danh sách thực thể CMP sinh viên và thực thể CMP lớp học có cấu trúc rất giống nhau. Thực thể lớp học bao gồm các phương thức tương đương *get* và *set* tất cả các sinh viên được liên kết tới môn học cụ thể.

```
public abstract Collection getStudents();
public abstract void setStudents(Collection students);
```

Ví dụ đưa ra có thể được sử dụng cho cả hai kiểu bean CMP và CMR.



Hình 5.5. Quan hệ kết nối giữa các entity bean

Trong Hình 5.5, có 3 thực thể bean CMP. Bean Dept có quan hệ một-nhiều với bean *Student* và bean *Student* có quan hệ nhiều–nhiều với bean *Course*. Tất cả quan hệ giữa các bean, CMP và BMP, và việc thực thi các phương thức trong SQL được xác định bởi người triển khai các công cụ tương ứng. Rõ ràng BMP đòi hỏi người triển khai phải làm nhiều việc, trong khi CMP không đòi hỏi người lập trình cung cấp bất cứ mã SQL nào.

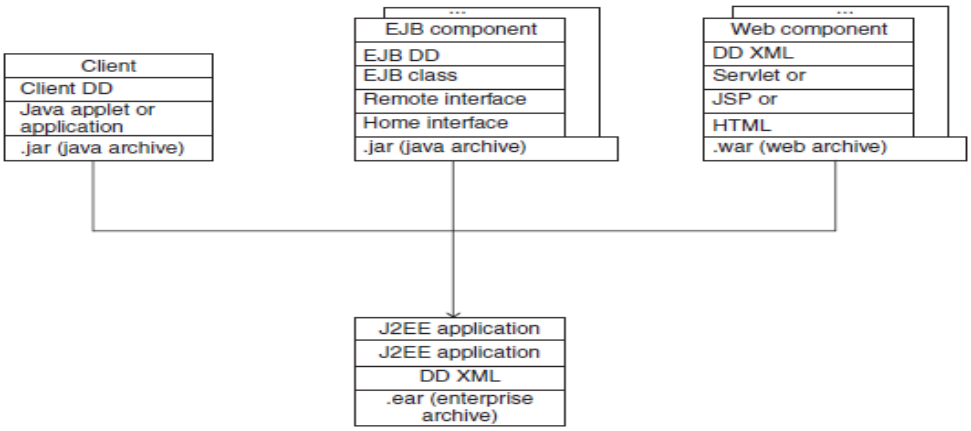
5.4. MÔ HÌNH TRIỂN KHAI EJB

Một thành phần EJB được đóng gói thành flie *.jar, sau đó sẽ được tổ hợp với các gói Web thành phần *.war và các gói ứng dụng client trong file ứng dụng *.ear.

Sau khi lập trình cho một bean và client của nó, chúng được dịch thành các file *.class. Sau đó, chúng ta gói các thành phần EJB, thành phần Web, hoặc client thành các file nén java (.jar) hoặc Web archive file (.war) với file DD XML. Cuối cùng gộp tất cả các phần này và nén thành một file enterprise (.ear) để được triển khai trên một server. Hình 5.6 cho minh họa về mô hình triển khai EJB.

DD tạo một file triển khai thuộc định dạng XML. Nó thể hiện các loại EJB, tên các class cho giao diện từ xa, giao diện home, cài đặt bean, đặc tả quản lý giao dịch, an toàn truy nhập, và các đặc trưng lưu trữ của bean thực thể. DD được tạo tự động bởi các công cụ triển khai sau khi khung triển khai được hòan thành. Dưới đây là một phần nội dung của một DD:

```
<?xml version="1.0">
<ejb-jar>
   <enterprise-beans>
       <entity>
           <ejb-name>studentBean</ejb-name>
           <home>studenthome</home>
      <remote>student</remote>
            <ejb-class>Student</ejb-class>
            <persistence-type>Container</persistence-type>
            <pri-key-class>Integer</pri-key-class>
            ...
            <cmp-field><field-name>id</field-name></cmp-field>
            ...
            <cmp-field><field-name>name</field-name></cmp-
field>
      </enterprise-beans>
      <assembly-descriptor>
          <security-role>
          ...
          </security-role>
          ...
      </assembly-descriptor>
</ejb-jar>
```

Hình 5.6. Hợp và triển khai trong J2EE

5.5 CASE STUDY

Phần này trình bày hai ví dụ với chỉ dẫn từng bước giải thích cách xây dựng các thành phần EJB, làm thế nào để kết hợp và triển khai các thành phần EJB trong các môi trường khác nhau.

5.5.1. Tạo giỏ hàng

Bean phiên cart biểu diễn một shopping cart trong Bookstore trực tuyến. Client của bean có thể thêm, xoá sách ra khỏi giỏ, lấy lại nội dung của giỏ sách. Chúng ta cần xây dựng hai file sau đây:

- Lớp bean phiên (CartBean)
- Interface nghiệp vụ từ xa (Cart)

Bước 1: Xây dựng Interface nghiệp vụ

Interface nghiệp vụ Cart định nghĩa tất cả các phương thức được cài đặt trong lớp bean. Nếu lớp bean cài đặt một interface, thì interface được gọi là interface nghiệp vụ. Interface nghiệp vụ là một interface cục bộ trừ khi nó được đánh dấu với *javax.ejb.Remote*.

```
package cart.ejb;
import cart.util.BookException;
import java.util.List;
import javax.ejb.Remote;

@Remote
public interface Cart {
    public void initialize(String person) throws
BookException;
    public void initialize(String person, String id)
            throws BookException;
    public void addBook(String title);
    public void removeBook(String title) throws BookException;
    public List<String> getContents();
    public void remove();
}
```

Tạo lớp bắt ngoại lệ BookException

```
package cart.util;

public class BookException extends Exception {
    public BookException() {
    }
    public BookException(String msg) {
        super(msg);
    }
}
```

Bước 2: Xây dựng lớp Bean

Giống như bất kỳ bean phiên trạng thái, CartBean phải đạt được các yêu cầu sau:

- Lớp phải được chú thích @Stateful
- Lớp cài đặt các phương thức định nghĩa trong interface nghiệp vụ

Bean phiên trạng thái cũng có thể:

- Cài đặt implements interface
- Cài đặt các phương thức life cycle callback, chú thích @PostConstruct, @PreDestroy, @PostActivate, @PrePassivate
- Cài đặt các phương thức với tuỳ chọn chú thích @Remote

```
Package cart.ejb;

import cart.util.BookException;
import cart.util.IdVerifier;
import java.util.ArrayList;
import java.util.List;
import javax.ejb.Remove;
import javax.ejb.Stateful;

@Stateful()
public class CartBean implements Cart {
    List<String> contents;
    String customerId;
    String customerName;

    public void initialize(String person) throws BookException
{
        if (person == null) {
            throw    new    BookException("Null    person    not
allowed.");
        } else {
            customerName = person;
        }
        customerId = "0";
        contents = new ArrayList<String>();
    }

    public void initialize(
        String person,
        String id) throws BookException {
        if (person == null) {
            throw    new    BookException("Null    person    not
allowed.");
        } else {
            customerName = person;
        }

        IdVerifier idChecker = new IdVerifier();

        if (idChecker.validate(id)) {
            customerId = id;
        } else {
            throw new BookException("Invalid id: " + id);
        }
```

```
        contents = new ArrayList<String>();
    }

    public void addBook(String title) {
        contents.add(title);
    }

    public void removeBook(String title) throws BookException
{
        boolean result = contents.remove(title);

        if (result == false) {
            throw new BookException("\"" + title + "\" not in
cart.");
        }
    }

    public List<String> getContents() {
        return contents;
    }

    @Remove()
    public void remove() {
        contents = null;
    }
}
```

Bước 3: Xây dựng client

```
package cart.client;

import java.util.Iterator;
import java.util.List;
import javax.ejb.EJB;
import cart.ejb.Cart;
import cart.util.BookException;

/**
 *
 * The client class for the CartBean example. Client adds
books to the cart,
 * prints the contents of the cart, and then removes a book
which hasn't been
 * added yet, causing a BookException.
 * @author cnpm
 */
public class CartClient {
    @EJB
    private static Cart cart;

    public CartClient(String[] args) {
    }
```

```
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        CartClient client = new CartClient(args);
        client.doTest();
    }

    public void doTest() {
        try {
            cart.initialize("Duke d'Url", "123");
            cart.addBook("Infinite Jest");
            cart.addBook("Bel Canto");
            cart.addBook("Kafka on the Shore");

            List<String> bookList = cart.getContents();

            bookList = cart.getContents();

            Iterator<String> iterator = bookList.iterator();

            while (iterator.hasNext()) {
                String title = iterator.next();
                System.out.println
                 ("Retrieving book title from cart: " +
title);
            }

            System.out.println
                ("Removing \"Gravity's Rainbow\" from
cart.");
            cart.removeBook("Gravity's Rainbow");
            cart.remove();

            System.exit(0);
        } catch (BookException ex) {
            System.err.println
                 ("Caught a BookException: " +
ex.getMessage());
            System.exit(1);
        } catch (Exception ex) {
            System.err.println("Caught an unexpected
exception!");
            ex.printStackTrace();
            System.exit(1);
        }
    }
}
```


Bước 4: Build và đóng gói ứng dụng >*cnpm*

Bước 5: Triển khai ứng dụng > *cnpm deploy*

Như vậy cart.ear được triển khai trên server

Bước 6: Chạy Cart Client

> cnpm run

Kết quả:

[echo] running application client container.

[exec] Retrieving book title from cart: Infinite Jest

[exec] Retrieving book title from cart: Bel Canto

[exec] Retrieving book title from cart: Kafka on the Shore

[exec] Removing "Gravity's Rainbow" from cart.

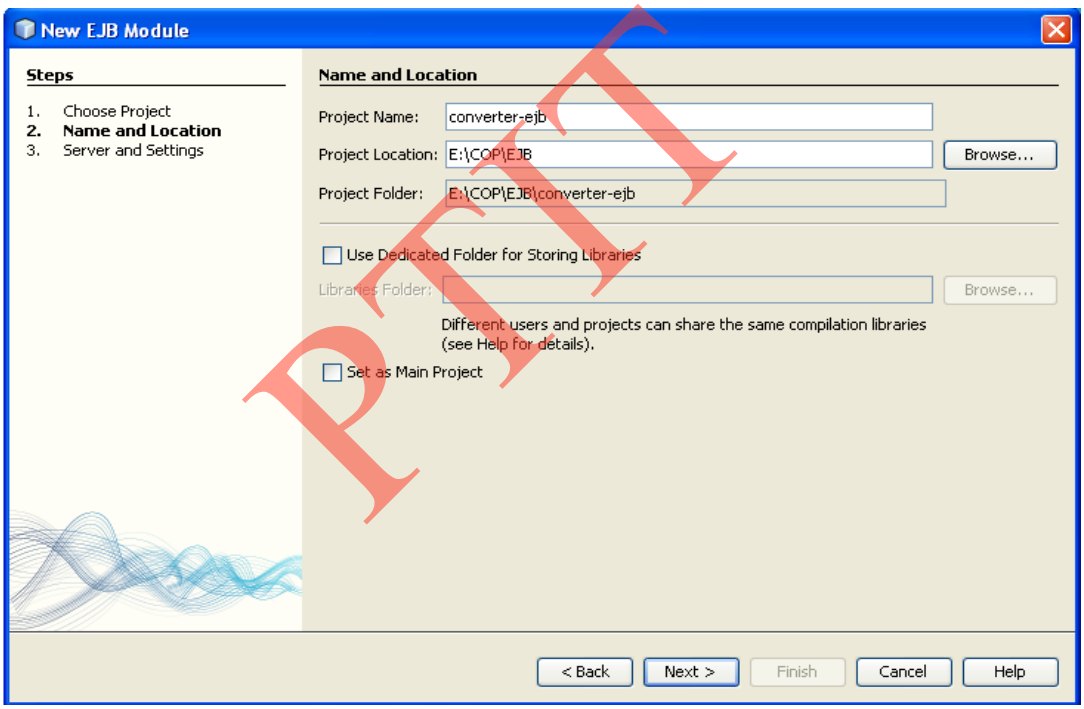
[exec] Caught a BookException: "Gravity's Rainbow" not in cart.

[exec] Result: 1

5.5.2. Xây dựng Converter với Netbean

Ví dụ Converter chuyển đổi từ độ F sang độ C và ngược lại dựa trên netbean.

Bước 1: Tạo ejb module converter-ejb



Bước 2: Tạo một Interface nghiệp vụ Converter

```
package converter.ejb;

import javax.ejb.Remote;

/**
 *
 * @author CNPM
 */
@Remote
public interface Converter {
    public double cToF(double c);
    public double fToC(double f);
}
```

Bước 3: Tạo ConverterBean

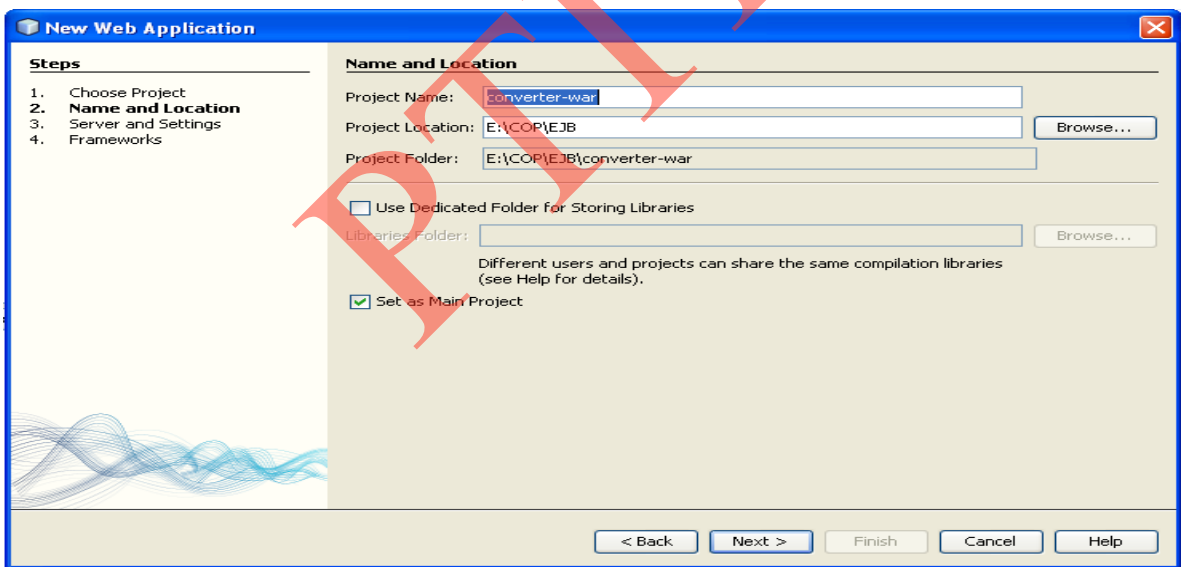
ConverterBean là bean hiên phi trạng thái thực thi Interface Converter

```
package converter.ejb;

import javax.ejb.Stateless;

@Stateless
public class ConverterBean implements Converter {

    @Override
    public double cToF(double c) {
        return c * 9/5 + 32;
    }

    @Override
    public double fToC(double f) {
        return (f - 32) * 5/9;
    }

}
```

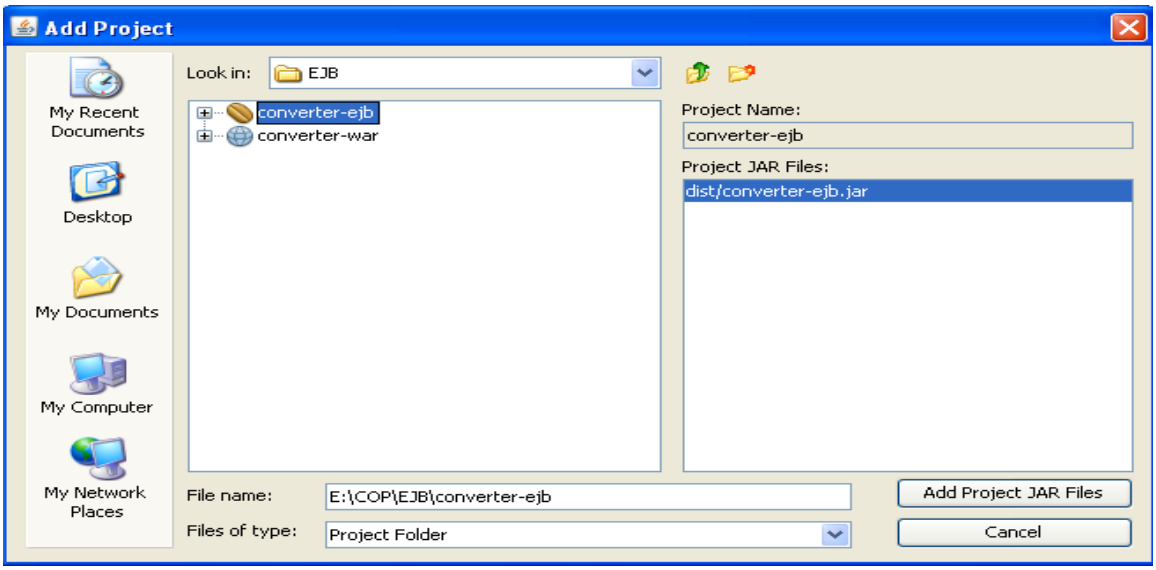
Bước 4: Build converter-ejb

Click chuột phải vào converter-ejb chọn build để build project

Bước 5: Tạo một ứng dụng web sử dụng converter-ejb: converter-war



Bước 5: Add converter-ejb vào library của converter-war



Bước 5: Tạo file index.jsp trong converter-war

```
<%@ page import="converter.ejb.Converter, javax.naming.*"%>
<%@ page import="java.math.*"%>
<%@ page import="javax.naming.*"%>
<%@ page import="javax.ejb.*" %>
<%@ page import="java.text.DecimalFormat" %>
<%@ page import="java.rmi.RemoteException" %>
<%!
    private Converter converter = null;
    public void jspInit() {
        try {
            InitialContext ic = new InitialContext();
            converter = (Converter)
    ic.lookup(Converter.class.getName());
        } catch (Exception ex) {
            System.out.println
             ("Couldn't create converter bean."+ ex.getMessage());
        }
    }

    public void jspDestroy() {
        converter = null;
    }
%>

<html>
<head>
<title>Temperature Converter</title>
</head>
    <body bgcolor="white">
    <h1><b><center> Temperature Converter </center></b></h1>
    <hr>
    <p>Enter a degree to convert:</p>
    <form method="get">
        <input type="text" name="degree" size="25">
        <br>
        <p>
        <input type="submit" name="fToC"
                value="Fahrenheit to Centigrade">
```

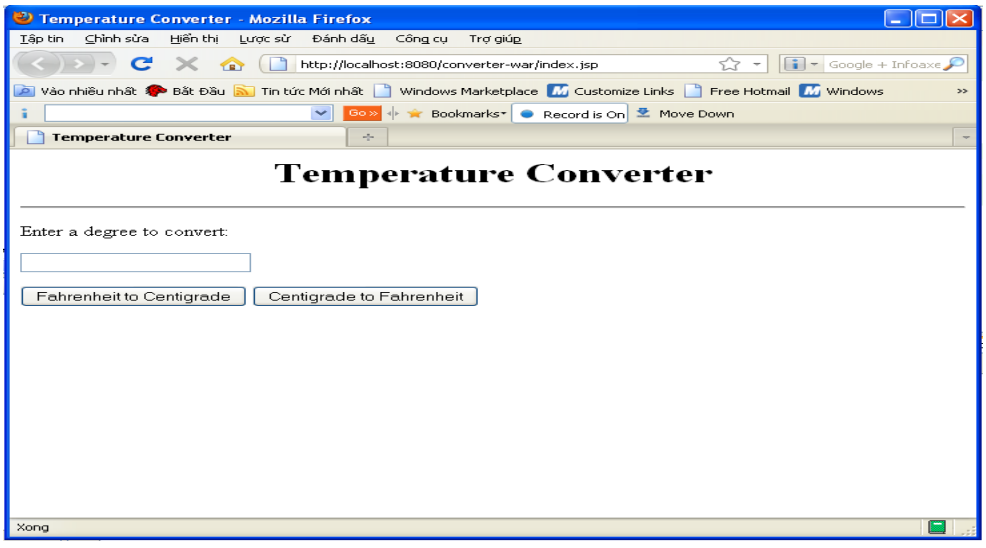
```
        <input type="submit" name="cToF"
                value="Centigrade to Fahrenheit">
    </form>
    <%
        DecimalFormat twoDigits = new DecimalFormat ("0.00");
        String degree = request.getParameter("degree");
        if ( degree != null && degree.length() > 0 ) {
        double d = Double.parseDouble(degree);
    %>
    <%
        if (request.getParameter("fToC") != null ) {
    %>
    <p>
    <%= degree %> fahrenheit degrees are
    <%= twoDigits.format(converter.fToC(d)) %>
    centigrade degrees.
    <% }
    %>
    <% if (request.getParameter("cToF") != null ) {
    %>
    <p>
    <%= degree %> centigrade degrees are
    <%= twoDigits.format (converter.cToF(d)) %>
    fahrenheit degrees .
    <% }
    %>
    <% }
    %>
    </body>
</html>
```

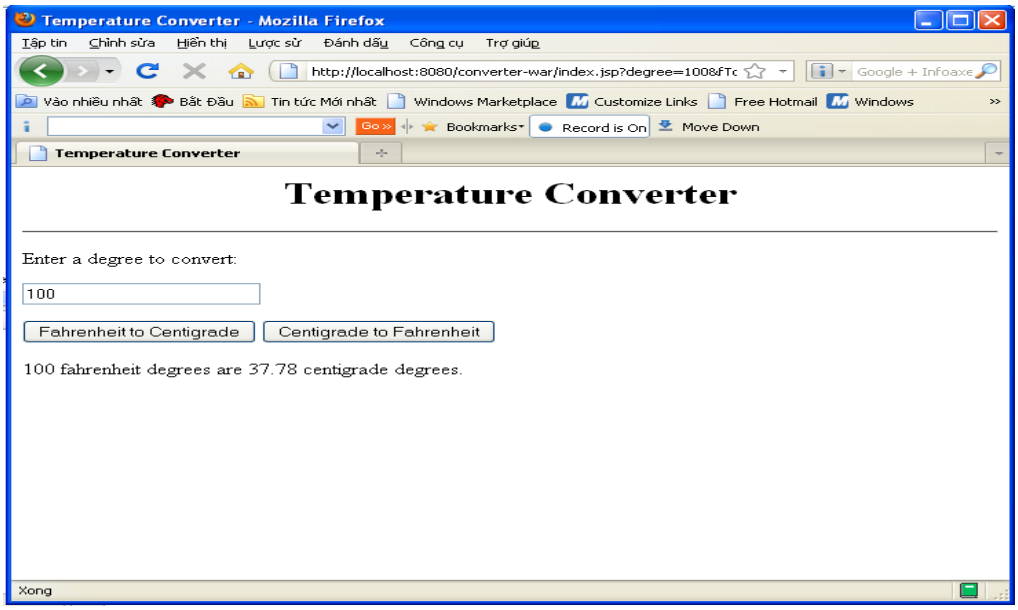
Bước 6: Built project converter-war

Click chuột phải vào converter chọn build

Bước 7: Chạy project converter-war

Kết quả:





5.6. KẾT LUẬN

EJB cung cấp một cơ sở hạ tầng thành phần tương tự như CORBA 3.0. Mô hình thành phần của công nghệ EJB được sử dụng trong việc xây dựng ứng dụng Java enterprise trong môi trường phân tán. Giao diện EJB cung cấp và thể hiện tất cả phương thức logic nghiệp vụ để được sử dụng bởi client của nó. Thực thi lớp bean mở rộng một lớp bean phiên hay lớp bean thực thể, triển khai dựa trên kiểu của thành phần EJB. Có hai kiểu bean phiên là phi trạng thái và trạng thái và cả hai đều hoạt động dựa trên các hành vi của client của nó. Thành phần EJB phi trạng thái không lưu giữ các thông tin trạng thái trong suốt phiên của chúng như một máy tính điện tử online. Mặt khác bean phiên trạng thái lưu giữ thông tin nào đó về trạng thái trong suốt phiên của chúng. Chẳng hạn, một bean phiên giỏ hàng Cart cần theo dõi việc thu thập các mặt hàng của client. Có hai kiểu bean thực thể là BMP và CMP, cả hai đều sử dụng tầng cơ sở để kết nối đến cơ sở dữ liệu và hỗ trợ việc lưu trữ. Mỗi bean thực thể có một bảng dữ liệu quan hệ tương ứng với nó. Mỗi thể hiện của thành phần thực thể EJB được lưu trữ trong bảng như là một cột. Bean CMP được tự do truy cập SQLvà ánh xạ đến cơ sở dữ liệu và được hoàn thành bởi người phát triển trong thời gian triển khai.

BÀI TẬP

1. Trình bày nguyên lý SOLID và thể hiện trong một ví dụ lập trình. Tham khảo: https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design
2. Trình bày mô hình thành phần của của J2EE
3. Phân biệt hai loại bean phiên và ý nghĩa sử dụng của hai loại này
4. Hoàn thiện hai Case study trong Mục 5.5 và chạy chương trình để cho ra kết quả
5. Xây dựng module quản lý mua hàng trong Hệ quản lý bán sách online với mô hình EJB
6. Xây dựng module quản lý mượn sách trong Hệ quản lý thư viện với mô hình thành phần EJB
7. Xây dựng module quản lý cho thuê xe ôtô với mô hình thành phần EJB
8. Xây dựng module quản lý đăng ký học tín chỉ với mô hình thành phần EJB

9. Xây dựng module quản lý nghe nhạc online với mô hình thành phần EJB
10. Xây dựng module quản lý đăng ký tour du lịch online với mô hình thành phần EJB
11. Xây dựng module quản lý đăng ký mua vé máy bay online với mô hình thành phần EJB

CHƯƠNG 6: MÔ HÌNH THÀNH PHẦN .NET

Mục tiêu của chương nhằm trình bày:

- .NET framework, một số khái niệm chung của các thành phần .NET.
- Các kiểu thành phần .NET, kết nối giữa các thành phần, và cách triển khai chúng.
- Các thành phần cục bộ và phân tán, các thành phần kết hợp và hợp thành.
- Phương thức đồng bộ và không đồng bộ.
- Hướng dẫn từng bước để xây dựng, triển khai, và sử dụng các thành phần .NET.

6.1 GIOÍ THIỆU

6.1.1 Tổng quan về .NET framework

.NET là một trong những công nghệ nổi tiếng của Microsoft. Phiên bản Beta đầu tiên được giới thiệu vào năm 2000. Khung .NET là một nền tảng giúp cho việc xây dựng, triển khai, và chạy nhanh chóng các ứng dụng. Các thành phần của .NET được tích hợp an toàn trong các ứng dụng cũng như để phát triển nhanh chóng dịch vụ web và các ứng dụng. .NET cung cấp một môi trường đa ngôn ngữ, hiệu năng cao và dựa trên thành phần cho các ứng dụng hiện thời trên Internet. Khung .NET bao gồm một máy ảo để cung cấp một nền tảng mới cho việc phát triển phần mềm. Lõi của .NET bao gồm các file XML và giao thức truy nhập đối tượng đơn giản (SOAP: Simple Object Access Protocol) để cung cấp dịch vụ web thông qua Internet.

Mục đích của .NET là để thuận tiện cho việc phát triển các ứng dụng máy để bàn và các dịch vụ ứng dụng dựa trên nền Web. Môi trường này làm cho dịch vụ như luôn sẵn sàng và có thể truy nhập được không chỉ trên nền Windows mà còn trên các nền tảng khác thông qua các giao thức phổ biến như SOAP và HTTP (Hình 6.1). Sau đây là một số đặc trưng của .NET:

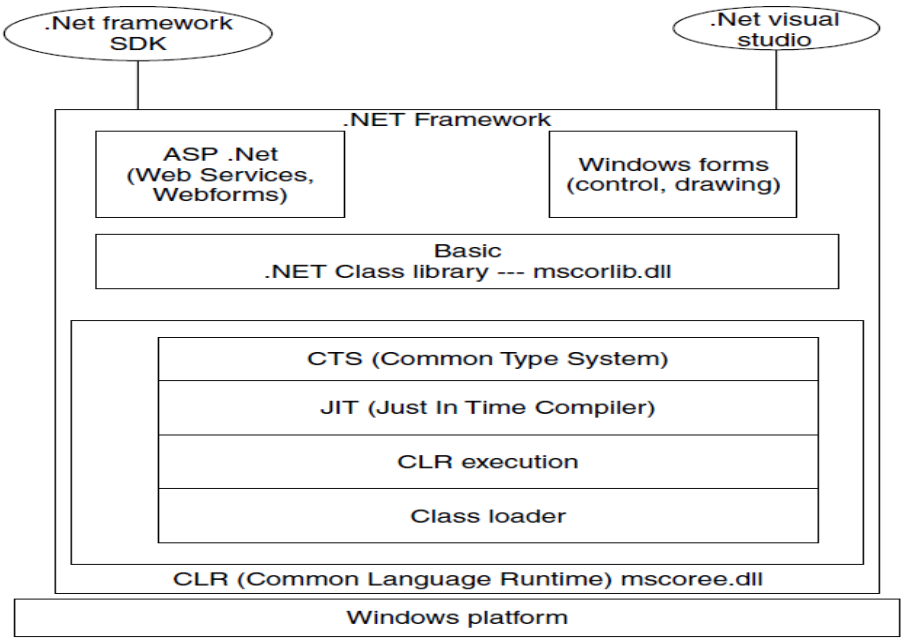
- Công nghệ này đã làm đơn giản việc thành phần hóa với công nghệ thành phần COM (Object Model) và công nghệ phân tán DCOM. Về nguyên lý, các thành phần COM có thể sử dụng lại như các thành phần phần mềm kéo thả trong việc xây dựng thành phần phần mềm và ứng dụng. Tuy nhiên, tiến trình phát triển cũng rất phức tạp và COM không hỗ trợ việc thực thi side-by-side, đây có thể là nguyên nhân gây xung đột giữa các phiên bản (vấn đề DLL Hell).
- Công nghệ .NET cho phép triển khai thành phần theo cách lắp ráp, điều này cho phép nhiều phiên bản của các thành phần cùng tên có thể cùng tồn tại mà không có bất kỳ xung đột nào. Công nghệ .NET đơn giản hóa việc tạo và triển khai các thành phần ngoài việc bảo mật các dịch vụ tin cậy và có khả năng thay đổi được cung cấp bởi các thành phần.
- .NET cũng giúp dễ dàng phát triển các thành phần phân tán bằng công nghệ truyền thông từ xa. .NET framework hỗ trợ khả năng phối hợp hoạt động giữa các phần giữa COM và các thành phần .NET. Một thành phần có thể làm việc với bất kỳ thành phần COM nào đang tồn tại. Nói cách khác, .NET có thể cung cấp các dịch

vụ tới các thành phần COM, và các thành phần COM có thể sử dụng bất kỳ các thành phần .NET nào. Việc phát triển các thành phần trong .NET dễ dàng hơn là trong COM.

- Dịch vụ web là một sự thay thế của công nghệ MS DCOM cho các ứng dụng Internet được hỗ trợ bởi các giao thức XML, SOAP, và HTML. .NET giải phóng việc viết mã của các nhà phát triển khỏi việc lập trình các chương trình dùng cho doanh nghiệp lớn như là quản lý giao dịch thông qua Enterprise Service. .NET khắc phục việc thiếu hỗ trợ tường lửa của DCOM và làm cho các dịch vụ trở nên sẵn sàng giữa các platform thông qua các giao thức gắn kết lỏng lẻo XML và SOAP.
- .NET framework có sẵn trong SDK và Visual Studio .NET IDE SDK, cả hai công nghệ này đều có thể tải về từ MS Website. .NET SDK là cơ sở của Visual Studio .NET và là một phần của Visual Studio .NET khi Visual Studio .NET được cài đặt. .NET framework bao gồm 2 phần chính: Common Languague Runtime (CLR) và một tập thống nhất các thư viện lớp cơ bản của framework bao gồm ASP.NET Web form để xây dựng các ứng dụng Web, Windows Forms để xây dựng các ứng dụng máy cá nhân, và ADO.NET để truy cập dữ liệu. SDK bao gồm tất cả các nhu cầu viết, xây dựng, kiểm tra và triển khai các ứng dụng .NET của bạn. Nó hỗ trợ tất cả các ngôn ngữ .NET như VB .NET, VC .NET. C#, và nhiều ngôn ngữ khác. .NET SDK và Visual Studio .NET có thể truy cập các dịch vụ của tất cả các tầng trong nền tảng .NET.

6.1.2. Cơ sở của .NET framework – CLR.

Giống như JVM trong Java, CLR là một môi trường máy ảo nằm trên đỉnh của hệ điều hành Windows. CLR bao gồm Common Type System (CTS), Just-In-Time IL Compiler (JIT), Execution unit (đơn vị thực thi), cùng với các dịch vụ quản lý khác như kết nối dữ liệu và quản lý bảo mật. Tất cả các thành phần phần mềm này được tập hợp lại trong một gói assembly (trong kiến trúc Java là file .jar) bao gồm mã MS Intermediate Language (MSIL) và file manifest (metadata miêu tả về gói này). Mã IL được biên dịch thành mã bản địa bởi trình biên dịch JIT. Mã IL được kiểm tra lại bởi CTS đầu tiên để kiểm tra tính hợp lệ của kiểu dữ liệu sử dụng trong mã đó. Hình 6.2 biểu diễn cách hoạt động của CLR.

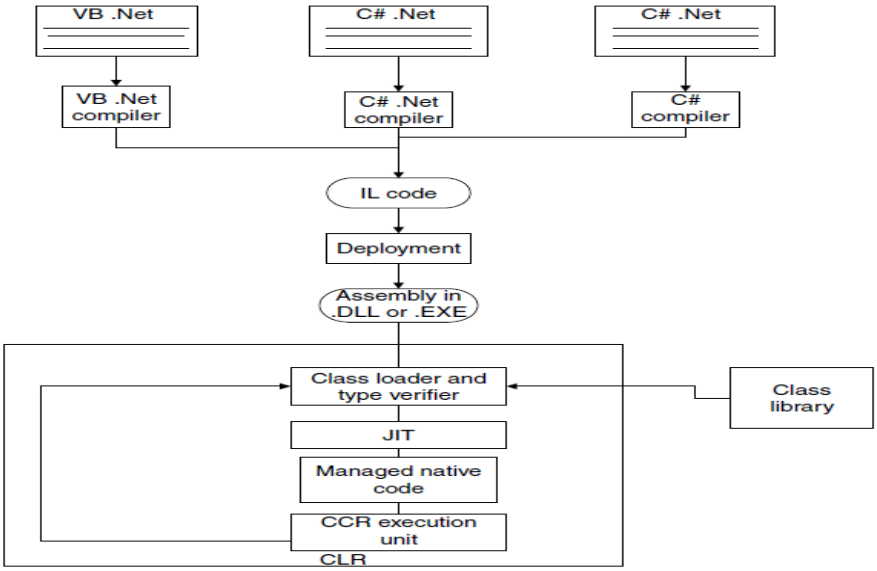


Hình 6.1. .NET framework [21]

.NET framework tích hợp nhiều ngôn ngữ lập trình (VB, VC++, C#, …) bằng cách cài đặt CLR. Không chỉ một thành phần trong một ngôn ngữ có thể truy cập tới các dịch vụ cung cấp bởi các thành phần khác trong các ngôn ngữ lập trình khác mà một lớp trong một ngôn ngữ có thể kế thừa các thuộc tính và các phương thức từ các lớp có quan hệ trong các ngôn ngữ khác. Thư viện lớp thống nhất (United Class Library) cung cấp một tập các lớp có thể sử dụng lại cho việc phát triển thành phần. CTS định nghĩa một tập chuẩn các kiểu dữ liệu và các luật cho việc tạo các kiểu mới. CLR cho biết làm thế nào để thực thi các kiểu này. Có hai loại kiểu: kiểu tham chiếu và kiểu giá trị. Mã hướng đến CLR và được thực thi bởi CLR được gọi là mã quản lý được của .NET. Tất cả các trình biên dịch ngôn ngữ của MS tạo ra mã để tương ứng với CTS.

Mã IL giống như mã byte code của Java, nó có thể giao tiếp với bất kỳ loại mã trình nào thông qua sự hỗ trợ của CLR. Mã IL có thể có định dạng là tập tin thực thi (.EXE) hoặc thư viện liên kết động kiểu (.DLL). Nếu mã IL này được tạo ra bởi trình biên dịch .NET, chúng được gọi là mã có quản lý (managed code) và chỉ được thực thi trên nền tảng .NET. Một số file DLL hoặc EXE không được tạo ra bởi trình biên dịch .NET (như phiên bản đầu của VC++) được gọi là mã không quản lý được.

Tóm lại, CLR là một máy thực thi có hiệu năng cao. Nó cung cấp một môi trường thực thi mã trong .NET. Việc quản lý mã bao gồm quản lý bộ nhớ, luồng, bảo mật, kiểm tra mã, và việc biên dịch IL.

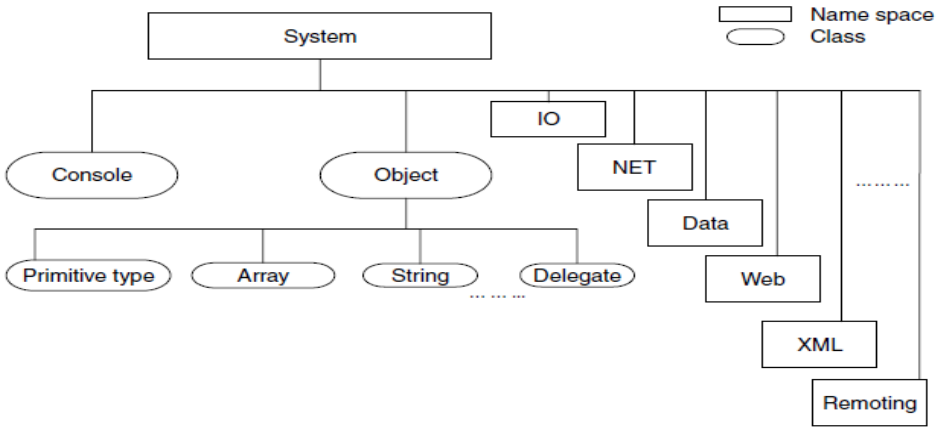


Hình 6.2. CLR của .NET [21]

6.1.3. Thư viện lớp của .NET

Thư viện lớp của .NET là một tập hợp các lớp cơ bản có thể sử dụng lại và được tổ chức bởi không gian tên (namespace). Thư viện lớp này kết nối tất cả các lớp bao gồm Windows Foundation Classes (WFC) thành một tập các lớp thống nhất. Không gian tên chỉ như một gói trong công nghệ Java và thư viện lớp giống như cấu trúc Java API. Một không gian tên bao gồm nhiều lớp và các không gian tên con. Nó được triển khai như một thư viện lớp thành phần và được tổ chức thành một cây phân cấp dựa trên thành phần. Hình 6.3 liệt kê một tập các thành phần trong kiến trúc thư viện lớp .NET. Tất cả các lớp trong thư viện có thể sử dụng bởi các lớp khác có ngôn ngữ khác nhau.

Namespace gốc trong thư viện lớp là System, nó bao gồm nhiều lớp cơ bản như Object, Console, và nhiều subnamespace như IO, Net, Data, Rmoting...Ví dụ, XML là một subnamespace của System và được triển khai thành System.XML.dll, ADO.NET có sẵn trong System.Data.dll tương ứng với System.Data, và các lớp Form-based UI có sẵn trong System.Windows.Forms.dll tương ứng với namespace System.Windows.Forms. Các nhà phát triển có thể tùy chỉnh namespace và tổ chức các lớp có liên quan trong một namespace tùy chỉnh. Một namespace có thể được triển khai như một assembly của các thành phần nhị phân. Các lớp với tên giống nhau có thể đặt trong các namespace khác nhau bởi vì chúng được tham chiếu bởi tiền tố namespacse khác nhau.



Hình 6.3. Thư viện lớp .NET [21]

Để sử dụng các lớp trong một namespace, cú pháp using <namespace> trong C# hoặc cú pháp import <namespace> trong VB phải được đưa vào ở phần đầu của code. Thư viện lớp built-in cơ bản của hệ thống đã được triển khai trong file mscorlib.dll.

6.2. MÔ HÌNH THÀNH PHẦN CỦA .NET

Công nghệ thành phần .Net tăng cường và đơn giản hoá sự tồn tại của các công nghệ MS COM, DCOM, COM+. Các thành phần MSIL DL thay thế các thành phần COM; các thành phần MSIL Remoting Channels EXE thay thế cho thành phần DCOM; Web Service là các thành phần SOAP được cho là các thành phần dựa trên web đa nền tảng và đa ngôn ngữ. Các thành phần .NET dễ dàng phát triển hơn COM và DCOM. Chúng giải quyết vấn đề xung đột version DLL Hell và vấn đề firewall trong DCOM. Công nghệ thành phần .NET là thành phần hướng ngôn ngữ hợp nhất. Bất kì thành phần .NET nào cũng có định dạng của MSIL đã được biên dịch trước, nó có thể là thành phần nhị phân được cắm vào bởi các thành phần MISL khác hoặc các client .NET tương thích khác.

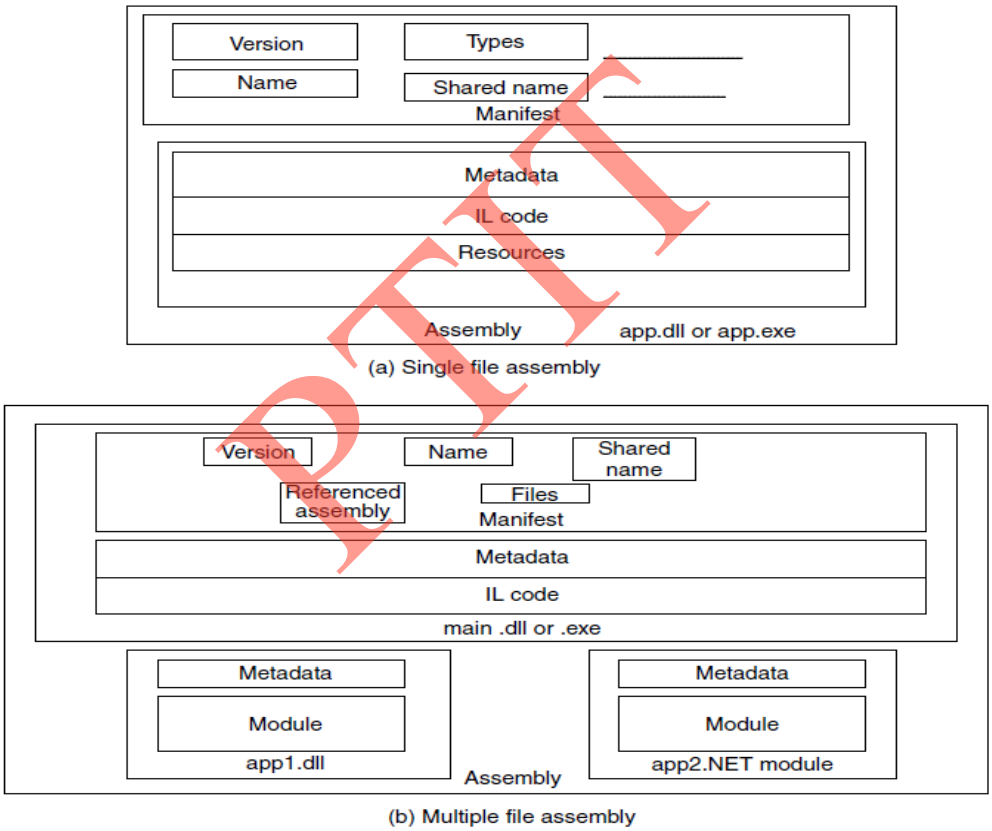
.NET framework tự nó được xây dựng theo mô hình thành phần. Ví dụ, System namespace **System.Runtime.Remoting** đã có sẵn trong **mscolib.dll** và **System.XML** namespace đã có sẵn trong **System.XML.dll**. Một file **.dll** là một thành phần đã được triển khai của .NET (Assembly). Một namespace giống một gói trong Java để tổ chức các class có quan hệ với nhau. Một assembly có thể có nhiều namespace, và một namespace có thể mở rộng trên nhiều file assembly. Chi tiết của .NET assembly sẽ được miêu tả trong các phần sau. Một thành phần .NET là một module MSIL đơn được biên dịch trước và xây dựng từ một hoặc nhiều class hoặc nhiều module được triển khai trong một assembly file DLL. Một assembly chứa bốn phần:

- Bảng ghi thông tin (Manifest) gồm tên assembly, phiên bản...
- Siêu dữ liệu (Metadata) của module
- Mã IL của module
- Các tài nguyên như các file ảnh…

Một module có mã MISL và siêu dữ liệu của nó nhưng không có manifest. Một module không thể tải động, nó được sử dụng như một khối xây dựng tại thời điểm biên dịch để xây dựng nên assembly Module file. Nó có phần mở rộng là **.netmodule**. Có thể có một hoặc

nhiều class trong một module. Một assembly được tạo ra từ một hoặc nhiều class trong một module. Mỗi module có thể được code bằng nhiều ngôn ngữ khác nhau nhưng cuối cùng phải có cùng định dạng MSIL. Assembly có một file manifest để tự mô tả thành phần. Một assembly có một file **.dll** hoặc **.exe** và có khả năng tải động. Đó là lý do tại sao người ta gọi thành phần .NET là một assembly. Một file **.dll** là một file mã nhị phân không có khả năng thực thi, giống như file **.class** trong Java.

Có nhiều kiểu thành phần khác nhau trong .NET framework. Chúng ta có thể phân loại chúng thành các loại thành phần trực quan hoặc không trực quan. Một thành phần trực quan là một điều khiển có thể được triển khai trong một hộp công cụ (toolbox) ví dụ như một icon để "kéo và thả" trong một cửa sổ dạng container. Thành phần không trực quan, được biết đến như .NET thành phần. Một .NET thành phần có thể được cài đặt ở phía client, phía server hoặc phía middleware. Kiểu của thành phần không quan trọng. Một .NET thành phần luôn cung cấp một số dịch vụ cho các client của chúng (client có thể là một thành phần hoặc ứng dụng client khác). Hình 6.4 biểu diễn nội dung của một assembly



Hình 6.4. Nội dung assembly của thành phần.NET [21]

Một thành phần .NET có thể là một thành phần cục bộ (.dll), nó chỉ có thể được truy cập một cách cục bộ (trong cùng miền ứng dụng) trong cùng một máy hoặc là một thành phần phân tán từ xa (.exe) qua nhiều miền ứng dụng. Một miền ứng dụng là một tiến trình lightweight, nó có thể được bắt đầu hoặc dừng lại một cách độc lập trong một tiến trình. Nó chỉ là một mức độ tự cô lập khác trong .NET. Một thành phần không thể trực tiếp truy

cập vào một thành phần trong miền hoặc tiến trình ứng dụng khác vì mỗi miền ứng dụng có không gian bộ nhớ riêng.

Một thành phần .NET DLL có thể được triển khai như một thành phần riêng khi nó biết client đích hoặc có thể được triển khai như một thành phần được chia sẻ công khai không biết client đích. Một thành phần DLL có thể được kéo thả vào Windowns form, Web form của DLL hoặc ứng dụng khác. Chúng ta hãy xem một thành phần rất đơn giản trong C# cung cấp các dịch vụ chuyển nhiệt độ giữa F^0 và C^0.

```
using System;

namespace TempConv
{
    public class TempConvComp
    {
        public double cToF(double c)
        {
            return (int)((c * 9) / 5.0 + 32);
        }
        public double fToc(double f)
        {
            return (int)((f - 32) * 5 / 9.0);
        }
    }
}
```

Chúng ta có thể xây dựng một module từ nó với dòng lệnh sau:

>csc /t:module TempConvComp.cs -> TempConvComp.netModule

Module này có thể được thêm vào một thành phần bằng cách:

>csc /t:library /addmodule: TempConvComp.netmodule anotherComp.dll

Chúng ta có thể xây dựng một DLL thành phần bằng dòng lệnh

>csc /t:library TempConvComp.cs -> TempConvComp.dll

Ở đây có 2 client của thành phần này. Một là **TempConvCSClient.cs** trong C# và **TempConvCppClient.cpp** trong C++. Cả 2 sử dụng lại thành phần **TempConvComp**

Dưới đây là chương trình C#:

```
using System;
using TempConv;

namespace TempConvCSClient
{
    class MainApp
    {
        public static void Main()
        {
        TempConv.TempConvComp myCSTempConvComp = new
TempConv.TempConvComp();
        double choice;
        double input;
```

```
        double output;
        bool next = true;
     while (next)
            {
         Console.WriteLine("Please enter your choice:");
         Console.WriteLine("\t\t 1 - Converter from F to
C");
         Console.WriteLine("\t\t 2 - From C to F");
         Console.WriteLine("\t\t 3 - exit");

     choice = Double.Parse(Console.ReadLine());
         if (choice == 1)
            {
  Console.WriteLine("Please tell me the temperature in F:
");
              input = Double.Parse(Console.ReadLine());
              output = myCSTempConvComp.fToC(input);
              Console.WriteLine("result = {0} ", output);
            }
         else if (choice == 2)
            {
  Console.WriteLine("Please tell me the temperature in C:
");
                    input =
Double.Parse(Console.ReadLine());
                    output = myCSTempConvComp.cToF(input);
                    Console.WriteLine("result = {0}
",output);
            }
         else
         {
         next = false;
                  Console.WriteLine("see you next time");
               }
            }
         }
     }
}
```

Trong **TempConvCSClient.cs**, client tải namespace **TempConv** bằng lệnh "**using TempConv**" và khởi tạo một thể hiện của thành phần **TempConvComp** và gọi method **fToC()** và **cToF()** được thành phần này cung cấp. Ứng dụng Client trong C# có thể được xây dựng bằng câu lệnh

>csc/t:exe /r:TempConvTemp.dll TempConvCSClient.cs -> TempConvCSClient.exe

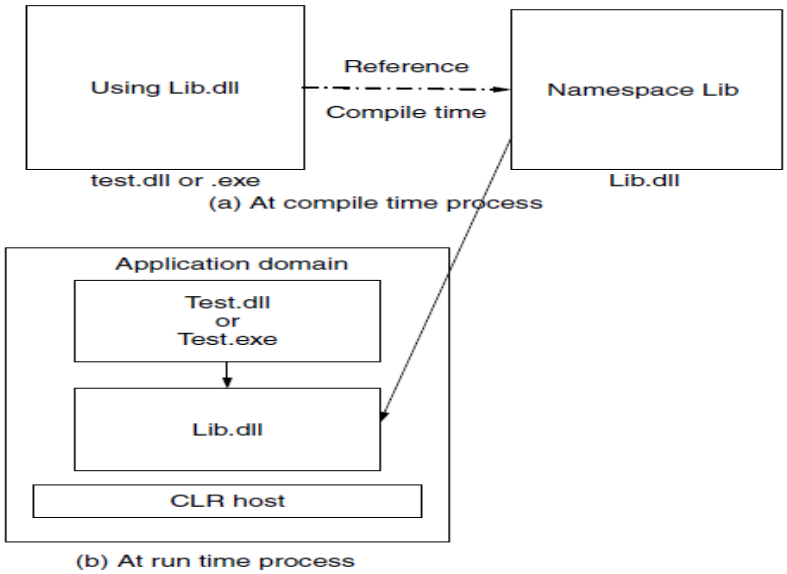
Sau đây là chương trình C++ client của thành phần **TempConvComp**:

```
#include "stdafx.h"
#include <iostream>
#using <mscorlib.dll>
#using "TempConv.dll"
```

```
using namespace System;
using namespace std;

#ifdef _UNICODE
int wmain(void)
#else
int main(void)
#endif
{
     int choice;
     double input;
     double output;
     bool next = true;

     TempConv::TempConvComp *myCSConvComp
                    = new TempConv::TempConvComp();
     while (next) {
          Console::WriteLine("Please enter your choice: ");
          Console::WriteLine("\t\t1 - Converter from F to
C");
          Console::WriteLine("\t\t2 - Converter from C to
F");
          cin >> choice;
          if (choice == 1) {
Console::WriteLine("Please input the temperature in F : ");
          cin >> input;
          output = myCSConvComp->fToC(input);
          Console::WriteLine(output);

          } else if (choice == 2) {
Console::WriteLine("Please input the temperature in C : ");
          cin >> input;
          output = myCSConvComp->cToF(input);
          Console::WriteLine(output);
          } else {
          next = false;
          Console::WriteLine("See you next time");
     }
     return 0;
}
```

Tương tự, trong **TempConvCppClient.cpp**, client tải namespace **TempConv** bằng **#using "TempConv.dll"** và lấy thể hiện của thành phần này, và sau đó lấy các dịch vụ thành phần này cung cấp.

Tóm lại, client của một thành phần tạo một tham chiếu đến thành phần server tại thời điểm biên dịch, và sau đó client tải thành phần tự động vào miền ứng dụng của nó tại thời gian chạy khi nó cần. Hình 6.5 mô tả tiến trình của một thành phần tại thời điểm biên dịch và thời điểm chạy.

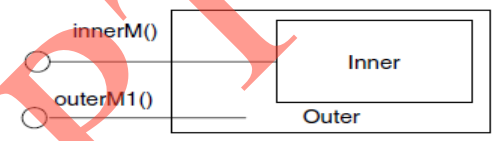


Hình 6.5. Các thành phần .NET tại thời điểm chạy và thời điểm biên dịch [21]

6.3. MÔ HÌNH KẾT NỐI

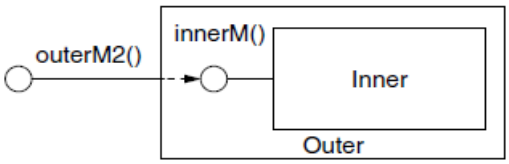
6.3.1. Thành phần kết nối .NET

Các tổ hợp thành phần cho phép thành phần tái sử dụng theo cả tổ hợp liên kết và tổ hợp bao hàm. Trong mô hình tổ hợp liên kết, dịch vụ của thành phần bên trong sẽ cung cấp dịch vụ trực tiếp cho thành phần client bên ngoài. Phương thức *innerM( )* của thành phần bên trong trở thành một phần của giao diện với thành phần bên ngoài như Hình 6.5. Ví dụ cài đặt chi tiết được thể hiện trong đoạn code dưới đây.



Hình 6.6. Tổ hợp liên kết

Trong tổ hợp bao hàm, nếu một lời gọi đến thành phần bên ngoài cần sự trợ giúp của thành phần bên trong, lời gọi đó sẽ được chuyển đến thành phần bên trong đó. Thành phần bên ngoài giấu giao diện của thành phần bên trong. Client không thấy được việc chuyển giao lời gọi. Phương thức *outerM( )* đẩy một lời gọi đến phương thức *innerM( )* của thành phần bên trong như trong Hình 6.7.

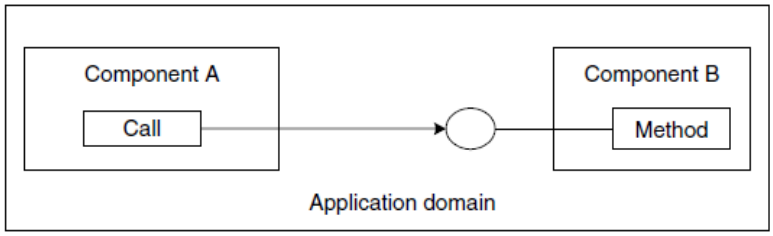


Hình 6.7. Containment

Một thành phần .NET còn có thể được tạo thành nhờ việc kết hợp tổ hợp liên kết và tổ hợp bao hàm theo kiểu cấu trúc phẳng hoặc lồng các tổ hợp ở nhiều mức sâu hơn. Dưới đây là một ví dụ giải thích việc kết hợp các tổ hợp liên kết và bao hàm.

```
using System;
namespace NS1
{
      public class Inner
      {
            public void innerM ()
            {
                  Console.WriteLine ("I am Inner.")
            }
      }
}
using System;
using NS1;
public class Outer
{
      public Inner i = new Inner ();
      //aggregation: Outer expose the Inner
      public void outerM1 ()
      {
            Console.WriteLine ("I am outer.");
      }
      public void outerM2() //delegation in containment
      {
            i.innerM();
      }
            public static void main()
      {
            outer o1 = new Outer();
            Inner i1 = o1.i;
            i1.innerM(); //interface to the aggregate
            o1.outerM();
            o1.outerM2(); // interface to the containment
            Inner i2 = new Inner();
            i2.innerM();
      }
}
```

Hình 6.8 biểu diễn một phương thức gọi trực tiếp một thành phần



Hình 6.8. Lời gọi trực tiếp

6.3.2. Kết nối các thành phần bằng Sự kiện (Event) và Ủy nhiệm (Delegate)

.NET Delegate là một kiểu phương thức tham chiếu đến một phương thức giống như con trỏ hàm trong C++ nhưng nó an toàn và đảm bảo về kiểu. Một Delegate sẽ ủy nhiệm luồng

điều khiển đến bộ điều khiển sự kiện đã đăng ký của nó khi sự kiện xảy ra. Nó hoạt động như một người quan sát, tương tự như một bộ lắng nghe sự kiện trong Java.

Một thể hiện của Delegate có thể chứa một phương thức tĩnh của một lớp, hoặc một phương thức của một thành phần, hoặc một phương thức của chính đối tượng là nó. Có hai kiểu Delegate: SingleCast và MultiCast. Một SingleCast chỉ có thể ủy nhiệm một phương thức một lần, như trong ví dụ dưới đây:

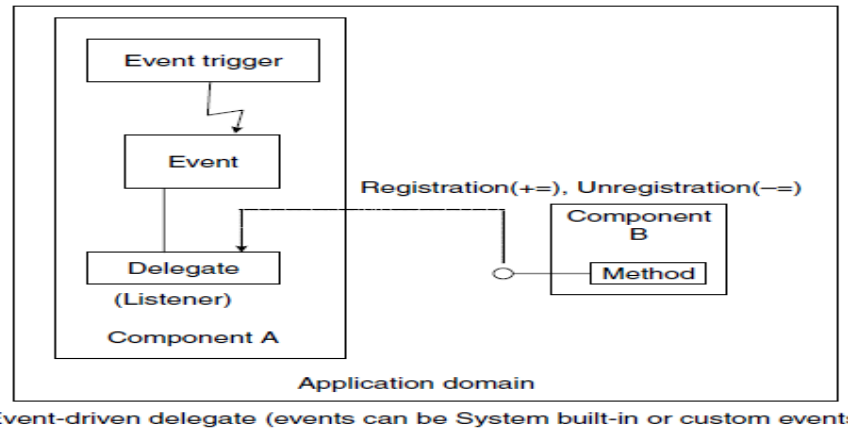
```
Delegate int Mydelegate();

public class MyClass
{
      public int ObjMethod() {- - - }

      static public int StaticMethod () {- - - }

      public class Drive { Static public void Main()
      {
            Myclass c = new MyClass();
            MyDelegate dlg = new MyDelegate(c.objMethod());
            dlg();
            dlg = new MyDelegate (MyClass.StaticMethod());
            dlg();
      }
}
```

Trong ví dụ này, MyDelegate là một Delegate tham chiếu đến bất cứ phương thức nào trả về kiểu int và không có tham biến. Các đặc trưng của *objMethod* và *StaticMethod* thỏa mãn *Delegate MyDelegate*. Lệnh *dlg( )* đầu tiên gọi đến *objMethod* và lệnh *dlg( )* thứ hai gọi đến phương thức *StaticMethod()*.

MultiCast Delegate có kiểu trả về là *void* và có thể nối với nhiều phương thức, và gọi chúng theo thứ tự đăng ký.

```
Delegate void MultiDelegate();
MultiDelegate mdlg = null;
mdlg += new MultiDelegate (Method1);
mdlg += new MultiDelegate (Method2);
```

Việc đăng ký được thực hiện bởi lệnh += *Delegate* và việc hủy bỏ đăng ký được thực hiện bởi lệnh -= *Delegate*. Delegate đóng vai trò bộ lắng nghe, và bộ xử lý sự kiện phải đăng ký với bộ lắng nghe này để xử lý sự kiện ngay khi sự kiện được kích hoạt. Quan hệ giữa sự kiện xử lý và kích hoạt sự kiện thông qua một Delegate được thể hiện trên Hình 6.9.



Hình 6.9. Quan hệ uỷ nhiệm giữa sự kiện và bộ xử lý của nó [21]

Một sự kiện là một thông điệp được một đối tượng gửi đi để gọi một hành động. Đối tượng phát ra sự kiện là nguồn sự kiện, và đối tượng nhận và xử lý sự kiện là đích sự kiện. Đây là một kiểu giao tiếp hướng sự kiện giữa các thành phần hoặc trong cùng một thành phần. Lớp Delegate là một lớp kênh giao tiếp giữa nguồn sự kiện và đích sự kiện. Sự kiện có thể là sự kiện được định nghĩa trước như một động cơ sự kiện bởi Windows Form thành phần. Người phát triển cũng có thể tự định nghĩa sự kiện. Thủ tục để tạo và sử dụng sự kiện như sau:

1. Tạo một lớp delegate.

```
public class DelegateStart;
```

2. Tạo một lớp chứa miền delegate, lớp MyClass.

```
public event DelegateStart EventStart;
- - -
```

3. Định nghĩa bộ xử lý sự kiện.

```
public void handleEvent(){ - - -}
```

4. Kết nối sự kiện ủy nhiệm với một bộ xử lý sự kiện thông qua bộ lắng nghe sự kiện, kích hoạt một sự kiện, gọi đến bộ xử lý sự kiện.

```
Public static void Main() { MyClass EventObj = new
MyClass();
EventObj.EventStart += new DelegateStart(handleEvent);
EventObj.EventStart();
...
}
```

6.3.3. Các bộ kết nối từ xa cho các thành phần phân tán .NET

Một thành phần hay một client không thể truy nhập trực tiếp tới một thành phần từ xa đang chạy trên một miền ứng dụng khác cũng như các tiến trình khác trừ phi nó sử dụng kênh kết nối từ xa. Có hai cách tạo đối tượng để nó có thể gọi một phương thức từ xa của một thành phần phân tán: Theo kiểu MBV (Marshal by Value), server truyền bản sao của đối

tượng cho client; theo kiểu MBR (Marshal by Reference), client sẽ tạo một tham chiếu của đối tượng từ xa. MBR là lựa chọn duy nhất khi thành phần từ xa chạy trên một phía ở xa.

Các client giao tiếp với các thành phần từ xa thông qua các giao thức. Ví dụ sau đây sử dụng kênh TCP (TCP channel). Tương tự như truyền thông kiểu socket trong Java, mỗi giao thức xác định bởi một cổng tương ứng. Chúng ta tạo kênh TCP trên cổng 4000 và đăng kí kênh này với lớp từ xa và tên URI mà client sẽ sử dụng để lấy đối tượng của thành phần từ xa. Chúng ta cũng phải tạo một kênh TCP và đăng kí nó ở phía client.

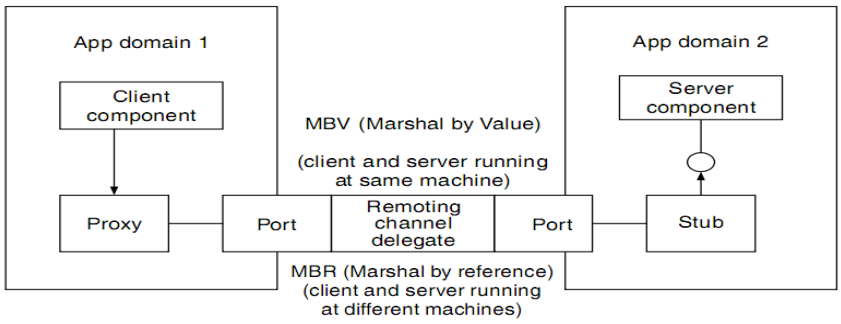
Dưới đây là ví dụ của một thành phần phân tán và client của nó. Chúng chạy ở chế độ từ xa (tức là chạy trên các miền ứng dụng hay các tiến trình khác biệt) vẫn sử dụng thành phần TempConvComp cũ nhưng ở đây được xây dựng như một thành phần phân tán được truy nhập từ xa.

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
public class CoTempConv : MarshalByRefObject
{
     public static void Main()
     {
          TcpChannel channel = new TcpChannel(4000);
          ChannelServices.RegisterChannel(channel);
          RemotingConfiguration.
          RegisterWellKnownServiceType (
                    typeof(CoTempConv),
                    "TempConvCompDotNet",
                    WellKnownObjectMode.Singleton);
                    System.Console.WriteLine("Hit <enter> to
                    exit...");
          System.Console.ReadLine();
     }
     public double cToF(double c)
     {
          return (int)((c*9/5.0+32)*100)/100.0;
     }
     public double fToC(double f)
     {
          return (int)((f-32)*5/9.0*100)/100.0;
     }
}
```

Dưới đây là client của thành phần trên.

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
class MainApp
{
     public static void Main()
```

```
      {
            try
            {
                  TcpChannel channel = new TcpChannel();
                  ChannelServices.RegisterChannel(channel);
                  CoTempConv myCSTempConvComp =
                  (CoTempConv)Activator.GetObject(
                  typeof(CoTempConv),
                  "tcp://127.0.0.1:4000/
                  TempConvCompDotNet");
                  double choice;
                  double input;
                  double output;
                  bool next = true;
                  while (next)
                  {
                        Console.WriteLine("Please enter your
                        choice:
                        1 - Converter from F to C,
                        2 - from C to F,
                        3 - exit");
                        choice=Double.Parse (Console.ReadLine());
                        if (choice == 1)
                        {
                  Console.WriteLine("Input temperature in F: ");
                        input=Double.Parse(Console.ReadLine());
                              output                              =
myCSTempConvComp.fToC(input);
                              Console.WriteLine(output);
                        }
                        else if (choice ==2)
                        {
                  Console.WriteLine("Input temperature in C:");
                  input=Double.Parse(Console.ReadLine());
                  output = myCSTempConvComp.cToF(input);
                  Console.WriteLine(output);
                  }
                  else
                        {
                        next= false;
                        Console.WriteLine ("See you next time.");
                        }
                  }
            }
            catch (Exception e)
            {
                  Console.WriteLine(e.ToString());
            }
      }
}
```



Hình 6.10: Lời gọi phương thức đồng bộ từ xa của thành phần phân tán

Các bước xây dựng server và client

>csc TempConvComp.cs

>csc /r:TempConvComp.exe TempConvCSClient.cs

Để kích hoạt thành phần server phân tán và client của nó, chúng ta chạy dòng lệnh sau:

>TempconvComp.exe

>TempConvCSClient.exe

Client tham chiếu tới thành phần từ xa bằng **Activator.GetObject()** và gọi phương thức của thành phần từ xa này. Hình 6.10 chỉ ra lời gọi phương thức đồng bộ từ xa của một thành phần phân tán.

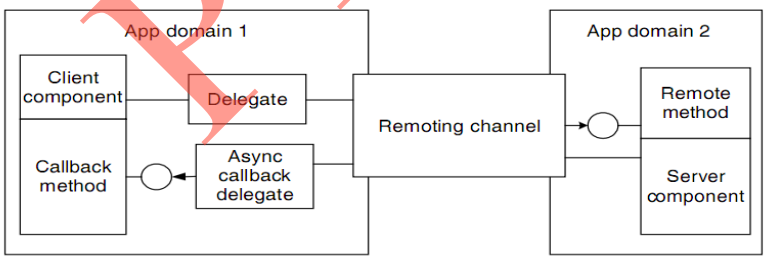
6.3.4. Gọi không đồng bộ từ xa giữa các thành phần phân tán .NET

Gọi không đồng đồng bộ từ xa dựa trên Remoting Delegate. Nó sẽ không khóa client trong khi chờ thông báo từ các thành phần từ xa. Ví dụ, giả sử một ai đó muốn được thông báo ngay khi giá cổ phiếu đạt tới mức độ nào đó. Thay vì thông báo giá cổ phiếu tại mọi thời điểm, tại sao không để cho server thực hiện tổng hợp cho bạn còn bạn có thể làm bất cứ cái gì bạn muốn. Trong một số trường hợp khác, các công việc thực hiện trên server mất khá nhiều thời gian, tại sao không để cho server thông báo cho bạn khi công việc đó đã được hoàn thành.

Chúng ta sử dụng lại thành phần **TempConvComp** được trình bày trước đây và tạo các lời gọi không đồng bộ từ server tới client. Giống như một chu trình, khi client thực hiện lời gọi đồng bộ tới phương thức của thành phần từ xa, nó truyền một phương thức gọi lại tới server để sau đó được gọi lại thông qua **Remoting Delegate**. Có hai Delegate: Một là *Mydelegate* trỏ tới phương thức từ xa *cToF* của thành phần từ xa có tên *TempConvDotNET*. Delegate không đồng bộ khác là *AsynchCallback*, được truyền vào phương thức *BeginInvoke* của *Mydelegate*

```
using System;
using System.Threading;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Messaging;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
public class Client {
     public delegate double MyDelegate(double c)
     public static int main(string [] agrs)
```

```
            TcpChannel chan = new TcpChannel();
            ChannelServices.RegisterChannel(chan);
            CoTempConv obj =
            (CoTempConv)Activator.GetObject(typeof(CoTempConv),
            "tcp://localhost:4000/TempConvCompDotNet");
            if(obj == null)
             System.Console.WriteLine("Could     not     locate
server");
            else {
                 AsyncCallback cb =
                          new AsyncCallback(Client.MyCallBack);
                 MyDelegate d = new MyDelegate(obj.cToF);
                 IAsyncResult ar = d.BeginInvoke(32, cb, null);
            }
            System.Console.WriteLine("Hit <enter> to exit ...
");
            System.console.ReadLine();
            return 0;
      }
      public static void MyCallBack(IAsyncResult ar)
      {
            MyDelegate                    d                    =
(MyDelegate)((AsyncResult)ar).AsyncDelegate;
            Coinsole.WriteLine(d.EndInvoke(ar));
            Coinsole.WriteLine("See you next time");
      }
}
```

Hình 6.11: Gọi lại phương thức không đồng bộ của thành phần phân tán

Ở đây, chúng ta có thể nhận thấy hai Delegate. Một là *MyDelegate* trỏ tới phương thức từ xa "cToF" của thành phần phân tán, hai là *AsynchCallback* trỏ tới phương thức quay lui "MyCallBack". Hình 6.11 minh họa lời gọi phương thức quay lui đồng bộ của thành phần phân tán được trình bày ở trên.

Tham số đầu tiên của *BeginInvoke* là 32 độ C và tham số thứ hai là một Delegate gọi lại. Gọi lại sẽ không khóa chương trình client. Khi thành phần phân tán hoàn thành công việc chuyển đổi, phương thức gọi lại sẽ được gọi và *IAsyncResult* được trả lại cho client.

6.4 MÔ HÌNH TRIỂN KHAI THÀNH PHẦN .NET

Thành phần .NET có thể đươc triển khai như một thành phần riêng hoặc thành phần chia sẻ chung trong một file assembly. Assembly là một đơn vị triển khai cơ bản trong .NET.

Thành phần riêng là thành phần trong đó nó sẽ được plug-in. Thành phần chia sẻ chung không biết thành phần nào sẽ sử dụng nó mà phải được đăng ký trong Global Assembly Cache (GAC). Thành phần chia sẻ hỗ trợ nhiều phiên bản thực thi thành phần cạnh nhau.

6.4.1. Triển khai riêng

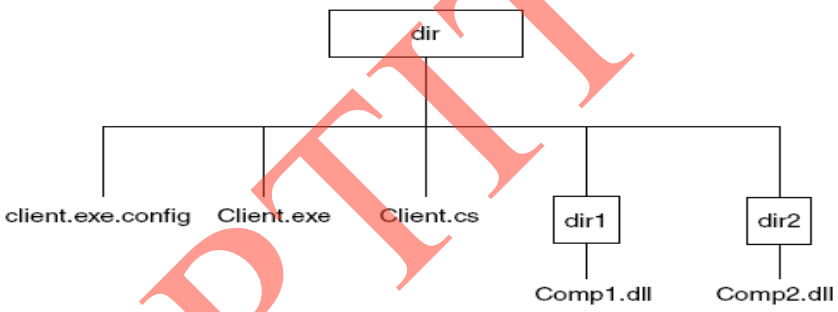
Thành phần riêng được triển khai trên một thư mục các client hoặc thư mục con các client. Đây là triển khai đơn giản nhất được thực hiện bằng cách copy tất cả các thành phần tới nơi mà client cư trú. Gỡ bỏ việc triển khai được thực hiện bằng cách xóa bỏ mọi thành phần .dll trong thư mục. Thành phần riêng không hỗ trợ kiểm soát việc xác định phiên bản và chỉ triển khai trong nội bộ của một công ty. Hình 6.12 minh họa ví dụ về triển khai riêng.

>csc /t:library /out:dir1\comp1.dll comp1.cs

>csc /t:library /out:dir2\comp2.dll comp2.cs

>csc /r:dir1\comp1.dll, dir2\comp2.dll /out:client.exe client.cs

Cần có một file miêu tả cấu hình XML nếu các thành phần không được đặt trong cùng thư mục với client của nó. Đường dẫn riêng phải được xác định trong thẻ con probing của thẻ assembly building trong file cấu hình ứng dụng với phần mở rộng là “config.”.



Hình 6.12: Cấu trúc thư mục triển khai thành phần riêng [21]

Sau đây là ví dụ của file cấu hình:

```
<configuration>
     <runtime>
          <assemblyBinding>
               <probing privatePath = “dir1; dir2”/>
          </assemblyBinding>
     </runtime>
</configuration>
```

CLR sẽ sử dụng đường dẫn riêng để tìm kiếm thành phần cần thiết để tải nếu nó không tìm thấy thành phần cần thiết trong thư mục hiện thời.

6.4.2 Triển khai chia sẻ chung

Cách triển khai thành phần có khả năng dùng lại phổ biến nhất là triển khai thành phần với một tên để đăng ký nó với GAC. Thành phần chia sẻ có strong name có thể là duy nhất được xác định bởi cặp khóa public/private. Thành phần chia sẻ đã đăng ký với GAC có thể được chia sẻ ở mọi nơi. Các bước cần để tạo ra một thành phần .NET chia sẻ là:

Bước 1: Tạo một cặp khóa public/private bằng cách sử dụng sn.exe

>sn –k mykey.snk

Khóa public dùng để xác minh lại khóa private. Khóa private được thể hiện như là một chữ ký số được lưu trong thành phần assembly. Khóa public được lưu trong file manifest của assembly. Khi client tham chiếu đến thành phần, mã thông báo của khóa public được lưu trong assembly của client.

Bước 2: Nhúng các dòng sau vào mã nguồn của thành phần

```
using System.Reflection:
[assembly:AssemblyKeyFile ("mykey.snk")]
[assembly:AssemblyDelaySign (false)]
[assembly:AssemblyVersion ("1,2,3,4")]
```

Lệnh sau sẽ gán chữ ký vào thành phần ngay lập tức

>csc /t:library mythành phần.cs

Lệnh tiếp theo sẽ lưu mã thông báo của khóa public vào thành phần của client

>csc /r:mythành phần.dll /out:myapplication.exe myapplication.cs

Nếu cần trì hoãn việc gán chữ ký, chúng ta có thể gán chữ ký sau bằng cách dùng lệnh:

>sn –R mythành phần.dll mykey.sn

Chữ ký được xác minh khi thành phần được đăng ký với GAC trong bước 3 để đảm bảo rằng thành phần không bị thay đổi trong khi assembly được xây dựng. Trong khi chạy, mã thông báo của khóa public trong manifest của client được xác minh với khóa public mà là một phần của định danh thành phần. Nếu chúng tương xứng với nhau thì nghĩa là đây chính là thành phần mong muốn.

Hình 6.13 chỉ ra cặp khóa public và private trong manifest của thành phần và của thành phần phía client. Số phiên bản của thành phần chia sẻ được đánh dấu bởi bốn số khác nhau và cách nhau bởi dấu chấm theo định dạng *major.minor.build.revision*. Việc thực thi được cài đặt bằng cách xác định phiên bản của .NET. .NET framework cho phép các thành phần có phiên bản khác nhau có cùng tên chạy trên các ứng dụng khác nhau. CLR kiểm tra các số major và minor trước tiên. Theo mặc định, chỉ cho phép phần major.minor hợp lệ. Số build theo mặc định là số tương thích ngược. Nói các khác, phiên bản 1.2.3.0 tương thích với 1.2.0.0 nhưng 1.2.3.0 không tương thích với 1.2.4.0. Nếu số phiên bản trong manifest của thành phần không tương thích với thành phần nào trong GAC, nó sẽ nạp một thành phần có phiên bản khác nhau về số revision. Số revision được gọi là Quick Fix Engineering (QFE) mặc định cho phép luôn tương thích. Chính sách phiên bản mặc đích có thể được tùy biến bằng cách ghi đè lên đặc tả assembly trong file cấu hình. Ví dụ:

```
<assemblyIdentity>
   <binding Redirect oldVersion = "1.2.3.4" newVersion =
"2.0.0.0"/>
</assemblyIdentity>
```

Điều này nghĩa là thành phần có phiên bản 2.0.0.0 sẽ được nạp thay vì thành phần 1.2.3.4.

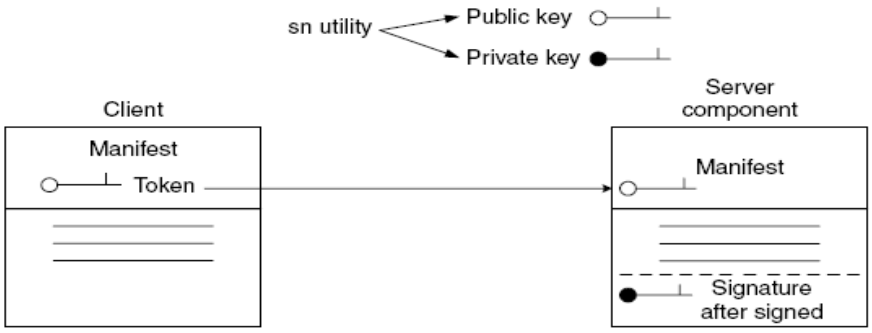
Bước 3: Đăng ký thành phần chia sẻ với GAC

>gacutil /I mythành phần.dll

Bước 4: Sử dụng thành phần chia sẻ

Client phải tạo tham chiếu đến thành phần chia sẻ để được sử dụng

>csc /t:exe /r:mythành phần.dll /out:myapp.exe myapp.cs



Hình 6.13: cặp khóa public/private trong thành phần .NET

Để sử dụng lại thành phần chia sẻ, mã nguồn client phải sử dụng không gian tên có sẵn trong assembly. Một ví dụ về triển khai thành phần chia sẻ TempConvComp như sau:

```
using System;
using System.Reflection;
[assembly:AssemblyVersion("1.0.0.0")]
[assembly:AssemblyKeyFile("originator.key")]
namespace TempConv
{
     public class TempConvComp
     {
          public TempConvComp()
          {
          }
          public double cToF(double c)
          {
          // how to control output format
               return (int)((c*9/5.0+32)*100)/100.0;
          }
          public double fToC(double f)
          {
          // how to control output format
               return (int)((f-32)*5/9.0*100)/100.0;
          }
     }
}
```

Các bước sau chỉ ra cách xây dựng thành phần chia sẻ:

>sn -k originator.key

>csc /t:library /out:TempConv.dll TempConvComp.cs

>gacutil /i TempConv.dll

>csc /r:TempConv.dll /t:exe /out:TempConvCSClient.exe

TempConvCSClient.cs

Mã nguồn của chương trình client tương tự TempConvCSClient.cs như trong phần 6.2.

6.5 KẾT LUẬN

Chương này tập trung trình bày công nghệ thành phần .NET. Một thành phần .NET là một file MSIL được triển khai như một file tích hợp và có thể chứa nhiều mô-đun. Một thành phần .NET có thể được triển khai như một thành phần riêng hoặc một thành phần được chia sẻ. Một thành phần .NET có thể là một thành phần của địa phương được truy cập bởi các thành phần khác trong cùng một miền ứng dụng hoặc có thể là một thành phần được phân phối bởi các thành phần khác được truy cập từ xa qua kênh điều khiển.

BÀI TẬP

1. Phân biệt tham chiếu và tham trị trong .NET. Tham khảo Reference vs. Value: Types, Passing and Marshalling http://blogs.msdn.com/b/tq/archive/2005/10/06/478059.aspx
2. Phân biệt các mô hình thành phần trong .NET
3. So sánh mô hình thành phần của .NET và J2EE
4. Sử dụng Tool để viết chương trình chuyển đổi nhiệt độ
5. Sử dụng Tool để viết chương trình chuyển đổi tỷ giá tiền
6. Xây dựng module quản lý mua hàng trong Hệ quản lý bán sách online với mô hình .NET
7. Xây dựng module quản lý mượn sách trong Hệ quản lý thư viện với mô hình .NET
8. Xây dựng module quản lý cho thuê xe ôtô với mô hình thành phần .NET
9. Xây dựng module quản lý đăng ký học tín chỉ với mô hình thành phần .NET
10. Xây dựng module quản lý nghe nhạc online với mô hình thành phần .NET
11. Xây dựng module quản lý đăng ký tour du lịch online với mô hình .NET
12. Xây dựng module quản lý đăng ký mua vé máy bay online với mô hình thành phần .NET

CHƯƠNG 7 : KIẾN TRÚC VÀ MẪU THIẾT KẾ

Chương này tập trung trình bày:

- Khái niệm về mẫu thiết kế
- Định dạng mẫu thiết kế
- Phân loại và sử dụng mẫu thiết kế

7.1 KHÁI NIỆM MẪU THIẾT KẾ

Sử dụng mẫu thiết kế (design pattern) là một giải pháp đã được áp dụng rộng rãi để giải quyết những vấn đề thường hay xảy ra giống nhau trong thực tế. Khái niệm mẫu thiết kế lần đầu tiên được đề xuất trong ngành xây dựng bởi kiến trúc sư Christopher Alexander vào những năm 1970. Ông cho rằng mẫu thiết kế là một giải pháp hiệu quả để giải quyết nhiều vấn đề thường hay lặp đi lặp lại trong thực tế kiến trúc.

Tại một Hội nghị về ngôn ngữ Lập trình Hướng đối tượng năm 1987, Kent Beck và War Cunningham đã trình bày áp dụng những ý tưởng của Alexander cho thiết kế và phát triển phần mềm đặc biệt dành cho lập trình hướng đối tượng. Bài báo này đã thu hút nhiều quan tâm nghiên cứu xây dựng mẫu thiết kế cho phát triển phần mềm. Đến năm 1994, cuốn sách "Design Patterns: Elements of Reusable Object-Oriented Software" của Gang of Four (GoF) – Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides – với đề xuất 23 mẫu thiết kế đã lan rộng và ảnh hưởng mạnh mẽ ứng dụng của các mẫu thiết kế. Để hiểu mẫu thiết kế là gì, chúng ta hãy xem một số ý kiến sau đây:

- Mẫu thiết kế là một cách hiệu quả để truyền tải những những thiết kế có chất lượng. Nó có thể là từ vựng của một phần tử thiết kế để giải quyết vấn đề đang gặp phải hay là một ngôn ngữ chung để trao đổi kinh nghiệm.
- Mẫu thiết kế không phải là một phát minh mà đúng ra là một phần tài liệu bàn về cách giải quyết một vấn đề được xem là hữu hiệu nhất qua kinh nghiệm xây dựng nhiều hệ thống phần mềm.
- Không phải cái gì tốt nhất trong thực tế đều được xem là mẫu mà phải thỏa mãn Quy luật "ít nhất ba". Luật này cho rằng *mẫu đó phải được chứng tỏ là lặp đi lặp lại và là tốt nhất trong ít nhất ba hệ thống đã xây dựng*. Ý nghĩa của luật này là nhằm đảm bảo rằng đã có một số chuyên gia phần mềm ứng dụng giải pháp đề xuất để giải quyết một số vấn đề trong thiết kế phần mềm rồi.

Mặc dù các mẫu thiết kế thường được xem xét trong bối cảnh phát triển phần mềm hướng đối tượng nhưng chúng có thể được áp dụng trong nhiều lĩnh vực khác với các cách tiếp cận lập trình khác nhau. Chỉ với một số thay đổi nhỏ, một mẫu thiết kế có thể được điều chỉnh để tạo ra mẫu thiết kế mới.

Các mẫu thiết kế không cung cấp các giải pháp cho mọi vấn đề trong thiết kế và phát triển phần mềm thực tế mà chỉ đưa ra các giải pháp tái sử dụng để giải quyết các vấn đề phát triển phần mềm thường gặp trong một bối cảnh cụ thể nào đó. Điều này có nghĩa rằng nó chỉ là các mô hình cung cấp các giải pháp tốt nhất cho một vấn đề trong một ngữ cảnh

cụ thể nào đó nhưng không thể hiệu quả trong ngữ cảnh khác. Vì vậy, các giải pháp của các mẫu thiết kế được đề xuất có thể áp dụng trong ngữ cảnh này nhưng không áp dụng được trong một số ngữ cảnh khác.

Mẫu thiết kế và Framework

Mẫu thiết kế không tồn tại dưới dạng thành phần phần mềm mà phải được cài đặt tại thời điểm sử dụng, không nên nhầm lẫn giữa framework với mẫu thiết kế. Framework là một dạng phần mềm gồm các thành phần phối hợp hoạt động với nhau để cung cấp một kiến trúc cho một miền ứng dụng nên nó có thể cài đặt nhiều mẫu thiết kế. **Bảng 7.1** sau đây nêu lên một số đặc điểm thể hiện sự khác biệt này [10]:

Bảng 7-1: Mẫu thiết kế và Framework

Mẫu thiết kế	Framework
Mẫu thiết kế là giải pháp lặp đi lặp lại đối với vấn đề nảy sinh trong vòng đời của ứng dụng phần mềm	Framework là nhóm thành phần cộng hợp với nhau để cung cấp một kiến trúc có thể sử dụng lại cho những ứng dụng
Mục đích là: • Giúp cải tiến chất lượng phần mềm (sử dụng lại, bảo trì, mở rộng…) • Giảm thời gian phát triển	Mục đích là: • Giúp cải tiến chất lượng phần mềm (sử dụng lại, bảo trì, mở rộng…) • Giảm thời gian phát triển
Mẫu là một cấu trúc logic	Framework là một dạng phần mềm
Mô tả mẫu thường độc lập với ngôn ngữ lập trình và chi tiết cài đặt	Framework đặc trưng cho cài đặt
Mẫu là tổng quát hơn và có thể sử dụng trong nhiều ứng dụng	Framework cung cấp chức năng cho miền cụ thể
Mẫu thiết kế không tồn tại dưới dạng thành phần phần mềm. Nó cần được cài đặt mỗi khi sử dụng	Framework không phải là một ứng dụng đầy đủ, các ứng dụng có thể xây dựng bằng cách kế thừa từ các thành phần có sẵn
Mẫu thiết kế là cách thiết kế tốt và được sử dụng để thiết kế các framework	Mãu thiết kế có thể sử dụng trong thiết kế và cài đặt các framework nghĩa là framework có thể chứa nhiều mẫu thiết kế

7.2 ĐỊNH DẠNG MẪU THIẾT KẾ

Vì các mẫu thiết kế được sử dụng bởi nhiều người nên chúng cần được thể hiện theo một định dạng được chấp nhận rộng rãi. Nghĩa là, kiểu định dạng phải dễ hiểu và tổng quát để có thể cài đặt với nhiều ngôn ngữ lập trình khác nhau. Hơn nữa vấn đề quan trọng là cần phải thể hiện được cách sử dụng các mẫu này trong thực tế ngữ cảnh tương ứng. Sau đây là một số đặc trưng của các mẫu thiết kế:

- *Tên mẫu:* Tên viết ngắn gọn của mẫu thường là một hoặc hai từ. Ví dụ, tên mẫu phải chọn làm sao thể hiện được mục đích của mô hình và được sử dụng như một tên duy nhất đặc trưng cho mẫu đó.
- *Phân loại:* Các mẫu được phân chia thành nhiều loại khác nhau. Nhóm mẫu tạo dựng (creational) là các mẫu tập trung vào cách tạo các đối tượng; nhóm mẫu cấu trúc (structural) tập trung vào việc sắp xếp các đối tượng với nhau thành một cấu trúc tổng thể lớn hơn; nhóm mẫu hành vi (behavioral) liên quan đến hành vi, cộng tác giữa các đối tượng để thực hiện một mục tiêu nào đó.
- *Mục đích:* Một mô tả chung ngắn gọn (một hoặc hai câu) về mô hình. Ví dụ, mẫu *Observer* mô tả phụ thuộc 1-nhiều giữa các đối tượng để khi một đối tượng thay đổi trạng thái thì tất cả phụ thuộc của nó có thể nhận biết và cập nhật tự động.
- *Định danh:* Các mẫu đều có định danh của mẫu đó.
- *Động lực:* Xác định vấn đề mà mẫu thiết kế này có thể được sử dụng để giải quyết.
- *Khả năng áp dụng:* Lĩnh vực mà mẫu thiết kế này có thể được sử dụng.
- *Cấu trúc:* Những biểu đồ lớp và thành phần với UML nhằm thể hiện cách hoạt động mẫu thiết kế này.
- *Các bên tham gia:* Xác định các đối tượng liên quan hoặc những công việc mà đối tượng đó được yêu cầu thực hiện.
- *Cộng tác:* Xác định sự cộng tác của các bên tham gia.
- *Kết quả:* Lợi ích và hạn chế của mẫu thiết kế này (thêm những gợi ý để giải quyết các vấn đề về hạn chế).
- *Cài đặt:* Những gợi ý, chỉ dẫn về cách sử dụng mẫu này bao gồm cả những thủ thuật, các cách ứng dụng hiệu quả và những điều nên tránh.
- *Mã nguồn mẫu:* Có ví dụ triển khai đầy đủ cho mẫu này được viết dưới dạng ngôn ngữ C++, java, C#…
- *Các áp dụng:* Ví dụ về các lĩnh vực mà mẫu này đã được áp dụng trong thực tế.
- *Các mẫu liên quan:* Các mẫu thiết kế tương tự như mẫu này hoặc cũng mang lại hiệu quả tương tự khi sử dụng như mẫu hiện thời.

7.3 PHÂN LOẠI MẪU THIẾT KẾ

Từ khi các mẫu thiết kế được đưa ra bởi nhóm GoF, đã có nhiều mẫu được đề xuất thêm [10]. Tài liệu này tập trung trình bày một số mẫu thông dụng được đưa ra bởi nhóm GoF. Hệ thống các mẫu này có thể nói là khá đầy đủ cho những người mới làm quen với mẫu thiết kế để giải quyết các vấn đề của thiết kế và phát triển phần mềm hiện nay. Hệ thống 23 mẫu thiết kế của GoF được chia thành ba nhóm: Nhóm tạo dựng, nhóm cấu trúc và nhóm hành vi.

Nhóm mẫu tạo dựng

- Mục đích của mẫu nhóm này nhằm giải quyết công việc thường xuyên là tạo ra các đối tượng.
- Các mẫu sẽ tạo ra một cơ chế đơn giản, thống nhất khi tạo các thể hiện của đối tượng.

- Cho phép đóng gói các chi tiết về các lớp nào được khởi tạo và cách các thể hiện này được tạo ra
- Nó khuyến khích sử dụng *interface* nhằm giảm độ liên kết.
- Nhóm này gồm các mẫu thiết kế sau đây: *Abstract Factory, Factory Method, Builder, Prototype, và Singleton*.

Nhóm mẫu cấu trúc

- Nhóm này chủ yếu giải quyết vấn đề một đối tượng ủy thác trách nhiệm cho những đối tượng khác. Điều này dẫn đến kiến trúc phân tầng của các thành phần với độ kết nối thấp.
- Tạo điều kiện giao tiếp giữa các đối tượng khi một đối tượng không thể truy nhập đối tượng khác theo cách thông thường hay khi một đối tượng không thể sử dụng được do giao diện không tương thích.
- Cung cấp các cách để cấu trúc một đối tượng với quan hệ hợp thành được tạo ra đầy đủ. Nhóm này liên quan tới các quan hệ cấu trúc giữa các thể hiện, bằng cách sử dụng các quan hệ kế thừa, liên kết, phụ thuộc. Để xem thông tin về mẫu này phải dựa vào biểu đồ lớp của mẫu.
- Nhóm này gồm các mẫu sau đây: *Adapter, Bridge, Composite, Decorator, Façade, Proxy* và *Flyweight*.

Nhóm mẫu hành vi

- Nhóm mẫu thiết kế này liên quan đến các quan hệ gán trách nhiệm để cung cấp các chức năng giữa các đối tượng trong hệ thống.
- Mô tả cơ chế giao tiếp giữa các đối tượng và xác định cơ chế chọn các thuật toán khác nhau bởi các đối tượng khác nhau tại thời gian chạy.
- Đối với các mẫu thuộc nhóm này ta có thể dựa vào biểu đồ giao tiếp và biểu đồ tuần tự để giải thích cách chuyển giao của các chức năng.
- Nhóm này gồm có một số mẫu sau đây: *Interpreter, Template Method, Chain of Responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy* và *Visitor*.

Ngoài ra, hai mẫu interface và abstract được xem là thuộc nhóm mẫu cơ bản và đã được sử dụng rộng rãi trong thiết kế các ngôn ngữ lập trình như Java. Các mẫu cơ bản này sẽ sinh viên đã quen thuộc khi học lập trình Java nên không đề cập trong tài liệu này và dành như bài tập cho bạn đọc. Nhiều ví dụ với mã nguồn đầy đủ về các mẫu thiết kế bạn đọc có thể tham khảo tại trang web: https://www.tutorialspoint.com/design_pattern/index.htm.

7.4 SỬ DỤNG MẪU THIẾT KẾ

7.4.1 Khi nào sử dụng mẫu thiết kế?

Mẫu thiết kế là mô tả tổng quát của một giải pháp đối với một vấn đề được "lặp đi lặp lại" trong các dự án phần mềm. Như vậy, một mẫu thiết kế có thể được xem như là một "khuôn mẫu" có sẵn được áp dụng để giải quyết một vấn đề trong các tình huống khác nhau.

Mẫu thiết kế được hiểu theo nghĩa tái sử dụng ý tưởng hơn là tái sử dụng mã lệnh. Mẫu thiết kế cho phép các nhà thiết kế ngồi lại cùng nhau để giải quyết một vấn đề nào đó mà không phải mất nhiều thời gian tranh cãi. Nhiều dự án phần mềm thất bại là do những người phát triển không có được sự hiểu biết chung về các vấn đề trong kiến trúc phần mềm. Do đó, mẫu thiết kế có thể sử dụng như một nguồn cung cấp những thuật ngữ và khái niệm để nhanh chóng hình dung ra "bức tranh" của giải pháp trong quá trình thiết kế. Hơn nữa, nếu mẫu thiết kế được sử dụng một cách hiệu quả thì dễ dàng đưa ra được kiến trúc hệ thống ban đầu và việc bảo trì phần mềm sau này cũng được tiến hành thuận lợi hơn.

*Mẫu thiết kế hỗ trợ tái sử dụng kiến trúc và mô hình thiết kế phần mềm quy mô lớn***.** Cần phân biệt mẫu thiết kế với framework. Framework hỗ trợ tái sử dụng mô hình thiết kế và mã nguồn ở mức chi tiết hơn. Trong khi đó, mẫu thiết kế được vận dụng ở mức tổng quát hơn, giúp các nhà phát triển hình dung và nhận biết các cấu trúc tĩnh và động cũng như quan hệ tương tác giữa các giải pháp trong quá trình thiết kế ứng dụng đối với một lĩnh vực riêng biệt.

Mẫu thiết kế là đa tương thích nghĩa là nó không phụ thuộc vào ngôn ngữ lập trình, công nghệ hoặc các nền tảng lớn như J2EE của Sun hay Microsoft .NET. Tiềm năng ứng dụng của mẫu thiết kế là rất lớn. Các kiến trúc dựa trên mẫu thiết kế đã được sử dụng khá nhiều trong các phần mềm mã nguồn mở, trong nền tảng J2EE hoặc .NET... Trong các dạng ứng dụng này, có thể dễ dàng nhận ra một số tên lớp chứa các mẫu thiết kế như Factory, Proxy, Adapter...

7.4.2 Sử dụng mẫu thiết kế như thế nào?

Các mẫu thiết kế cần phải thay đổi để phù hợp với tình huống sử dụng cụ thể. Sau đây là một số lý do vì sao cần sự điều chỉnh như vậy:

- *Sự khác biệt về ngôn ngữ biên dịch:* Các ngôn ngữ lập trình kể cả các ngôn ngữ hướng đối tượng đều có những khác biệt nhau như cú pháp, biên dịch...Do đó, các mẫu thiết kế cần phải hiệu chỉnh sao cho phù hợp với ngôn ngữ đang sử dụng cho dự án.
- *Sự khác biệt về quan điểm:* Thông thường, có thể có nhiều giải pháp đối với mỗi vấn đề đặt ra trong khi thiết kế. Do đó, một mẫu thiết kế được đưa ra bởi một chuyên gia hoặc một nhóm các chuyên gia không có nghĩa là nó đã là hoàn hảo.
- *Sự khác biệt về kiểu:* Thiết kế và kiểu mã hóa có thể nảy sinh từ nhiều nguồn như kinh nghiệm, nguyên tắc viết mã của doanh nghiệp, các yêu cầu của một thư viện hoặc framework. Nếu một mô hình không phù hợp với kiểu mà dự án sử dụng thì chúng ta phải điều chỉnh nó.

- *Vấn đề khác nhau:* Mỗi một mẫu thiết kế phải được thể hiện hết sức tổng quát nên một số mô hình lớp trình bày trong mẫu có thể sẽ không được sử dụng. Ví dụ, trong mẫu thiết kế *Observer* (xem Chương 10), các lớp *ConcreteObserver* và *ConcreteSubject* có thể không được sử dụng vì không phù hợp với tình huống thiết kế của chúng ta.
- *Thành phần, kế thừa và đa kế thừa:* Các ngôn ngữ hướng đối tượng và các nhà phát triển thường có sự khác biệt trong cách sử dụng kế thừa và thành phần. Ví dụ, trong C++ có đa kế thừa nhưng Java thì lại không có.
- *Đơn giản hóa:* Một mẫu thiết kế thường là tổng quát hóa cho nhiều tính năng và do đó có thể sẽ phức tạp hơn thực tế thiết kế của yêu cầu dự án đề ra. Vì vậy, chúng ta có thể làm đơn giản mẫu đó nhằm mang lại hiệu quả cao hơn.

7.5 KẾT LUẬN

Chương này đã trình bày một tổng quát về khái niệm mẫu thiết kế, các nhóm mẫu thiết kế và những đặc trưng của mẫu thiết kế. Một bảng so sánh về sự khác biệt giữa mẫu thiết kế và framework đã đượt liệt kê. Đồng thời cũng đã trình bày một số hướng dẫn khi nào và cách sử dụng các mẫu này. Chi tiết mô hình và cài đặt các mẫu thiết kế sẽ được trình bày trong các chương tiếp theo.

BÀI TẬP

1. Sử dụng khái niệm lớp giao diện *interface* để xây dựng một ứng dụng tính và hiển thị lương của các loại nhân viên khác nhau trong một doanh nghiệp phần mềm cho trong Bảng 7.2. Yêu cầu cài đặt như sau: Dịch vụ tính lương cung cấp bởi các đối tượng thuộc loại khác nhau như *CategoryA, CategoryB*...được cài đặt từ *getSalary ()* của *interface SalaryCaculator*; Lớp nhân viên *Employee* nhận *SalaryCaculator* như là kiểu thành phần và cài đặt phương thức hiển thị *display()*.

    a. Vẽ biểu đồ UML để thể hiện các quan hệ trên.

    b. Cài đặt đầy đủ với lớp thực thi MainApp. Lương được tính theo công thức lương cơ bản cộng với lương làm thêm giờ.

Bảng 7.2: Phân loại nhân viên

Nhân viên	Loại
Lập trình, thiết kế và tư vấn	Loại A, lương cơ bản 2000, tăng giờ 15/giờ
Đại diện bán hàng, quản lý bán hàng, kế toán, nhân viên kiểm chứng	Loại B, lương cơ bản 1500, tăng giờ 10/giờ
….	…
Nhân viên bán hàng, nhân viên tiếp thị	Loại C, lương cơ bản 800, tăng giờ 5/giờ

2. Thiết kế *interface Search* khai báo phương thức tìm một mặt hàng như sách *Book* trong một danh sách. Thiết kế và cài đặt hai lớp tìm kiếm nhị phân *BinarySearch* và tìm kiếm tuần tự *LinearSearch* để thực thi các cách tìm kiếm trong danh sách.

3. Thiết kế *interface AddressValidator* khai báo các phương thức để xác nhận các phần khác nhau của một địa chỉ. Thiết kế và cài đặt hai lớp USAAddress (Hoa kỳ) và VNAddress (Việt nam) để xác nhận các địa chỉ của các nước tương ứng.

4. Trong một doanh nghiệp, nhân viên thường được thể hiện dạng phân cấp lớp với lớp cơ sở là nhân viên *Employee* và sau đó là nhân viên đại diện bán hàng *SalesRep,* nhân viên tư vấn *Consultant*…Hãy tạo một số phương thức sau đây trong lớp *Employee*:

    - Lưu và hiển thị dữ liệu nhân viên
    - Truy nhập thuộc tính nhân viên như tên, id
    - Tính thu nhập hàng tháng cho nhân viên

    a. Sử dụng lớp trừu tượng abstract để cài đặt yêu cầu trên. Chú rằng các phương thức đầu là chung cho các loại nhân viên, nhưng tính thu nhập hàng tháng lại là khác nhau cho các loại nhân viên.

    b. Sử dụng lớp interface để cài đặt. So sánh hai cách cài đặt này.

CHƯƠNG 8: CÁC MẪU THIẾT KẾ TẠO DỰNG

Chương này tập trung trình bày nhưng mẫu thiết kế tạo dựng, nội dung bao gồm các mẫu thiết kế:

- Mẫu factory method
- Mẫu Singleton
- Mẫu Abstract factory
- Mẫu Prototype
- Mẫu Builder

8.1 MẪU THIẾT KẾ FACTORY METHOD

8.1.1 Đặt vấn đề

Trong các ngôn ngữ lập trình hướng đối tượng như Java, các lớp con trong cây phân cấp lớp có thể cài đặt đè phương thức lớp cha để cung cấp các kiểu chức năng khác nhau cho cùng tên phương thức. Khi một đối tượng ứng dụng biết chính xác chức năng cần thiết, nó sẽ khởi tạo trực tiếp lớp nào cung cấp chức năng đó.

Tuy nhiên, có những tình huống mà đối tượng ứng dụng chỉ biết cần phải truy nhập lớp bên trong cây phân cấp nhưng không biết chính xác chọn lớp con nào. Như vậy, cần phải cài đặt một quy tắc chọn lớp dựa vào một số yếu tố như trạng thái của ứng dụng đang chạy, cấu hình…Tuy nhiên, cách làm như vậy có thể dẫn đến nhiều bất lợi:

- Gây nên sự kết nối chặt giữa đối tượng ứng dụng và cây phân cấp dịch vụ.
- Khi quy tắc chọn lớp thay đổi thì mọi đối tượng ứng dụng cũng thay đổi theo.
- Quy tắc chọn lớp đòi hỏi đối tượng ứng dụng phải biết đầy đủ về sự tồn tại và chức năng của mỗi lớp nên cài đặt có thể có những câu lệnh điều kiện bất hợp lý.

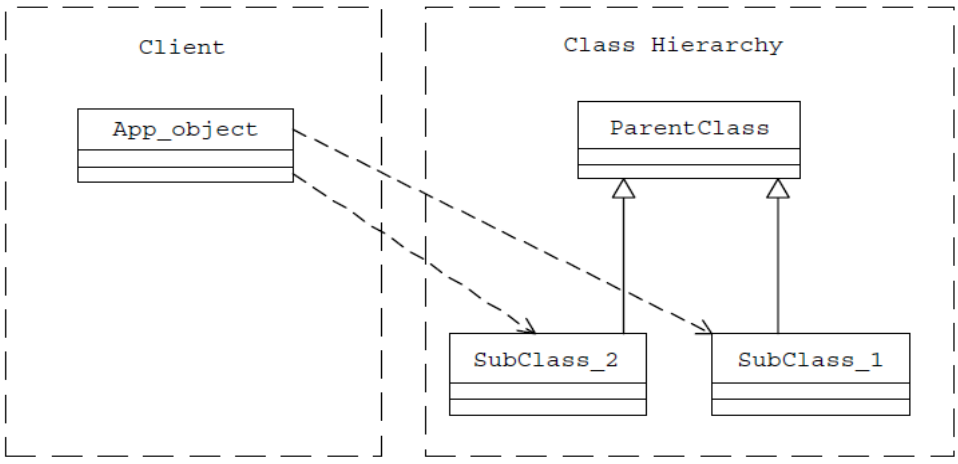
8.1.2 Cấu trúc mẫu

Mẫu thiết kế Factory method có thể khắc phục nhược điểm trên bằng cách đóng gói chức năng yêu cầu và khởi tạo cũng như chọn lớp thích hợp bằng một phương thức gọi là *factoryMethod*. Phương thức này cũng được định nghĩa như một phương thức trong một lớp:

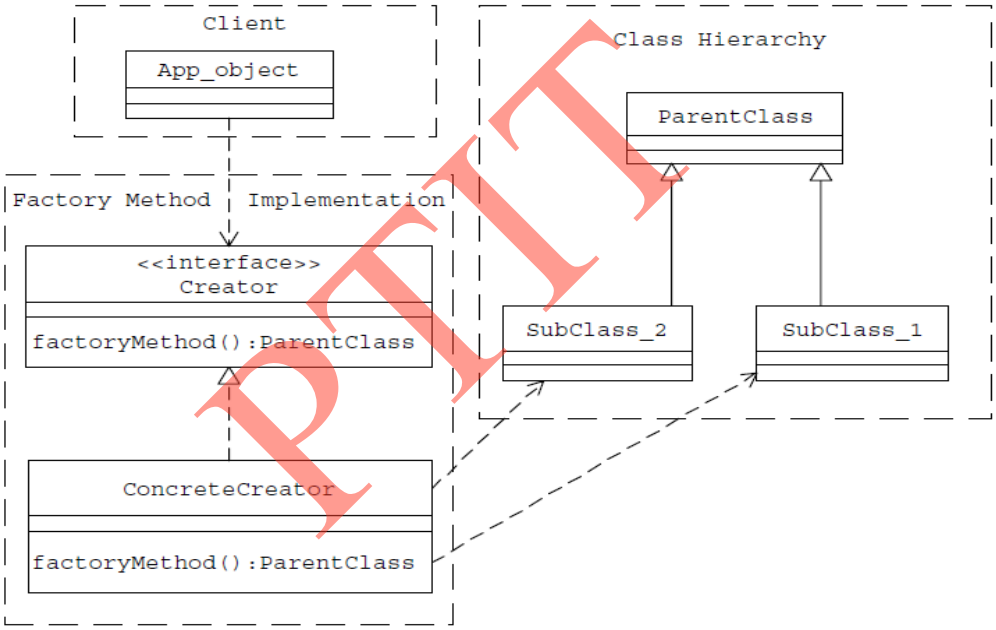
- Chọn lớp thích hợp từ cây phân cấp lớp dựa vào ngữ cảnh ứng dụng và nhưng yếu tố khác
- Khởi tạo lớp đã chọn và trả lại một thể hiện của kiểu lớp cha

Khác với biểu đồ Hình 8.1, trong Hình 8.2, *factoryMethod* trả lại thể hiện lớp đã chọn như đối tượng của kiểu lớp cha nên đối tượng ứng dụng không cần biết sự tồn tại của các lớp trong phân cấp. Một cách đơn giản để thiết kế *factoryMethod* là tạo ra một lớp trừu tượng *abstract* hay một giao tiếp *interface Creator* chỉ để khai báo *factoryMethod()*. Khi đó một lớp con *ConcreateCreator* sẽ được thiết kế để cài đặt toàn bộ *factoryMethod()*. Ta cũng có

thể làm cách khác là cài đặt lớp Creator cụ thể với cài đặt mặc định *factoryMethod()* trong đó. Các lớp con khác của lớp Creator sẽ cài đè lên quy tắc chọn lớp cụ thể *factoryMethod()*.



Hình 8.1: Đối tượng client truy nhập trực tiếp cây phân cấp dịch vụ



Hình 8.2: Đối tượng client truy nhập cây phân cấp dịch vụ qua factoryMethod

8.1.3 Tình huống áp dụng

Sau đây là một số tình huống có thể áp dụng Factory method:

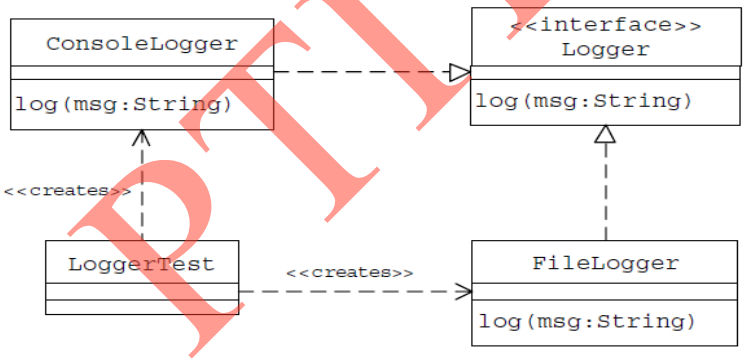
- Khi muốn tạo ra một framework để có thể mở rộng sau này, việc sử dụng mẫu *factory method* sẽ cho phép quyết định mềm dẻo khi chỉ ra loại đối tượng nào được tạo ra.
- Khi muốn một lớp con, mở rộng từ một lớp cha, quyết định lại đối tượng được khởi tạo.
- Khi ta biết khi nào thì khởi tạo một đối tượng nhưng không biết loại đối tượng nào được khởi tạo.

- Khi ta cần một vài khai báo chồng phương thức khởi tạo với danh sách các tham số như nhau, điều mà Java không cho phép. Thay vì điều đó ta sử dụng các *Factory Method* với các tên khác nhau.
- *Factory Method* thường được cài đặt cùng với *Abstract Factory* và *Prototype* và sẽ được trình bày sau. Khi đó lớp chứa *Factory Method* thường được gọi là *Template Method*.

8.1.4 Ví dụ

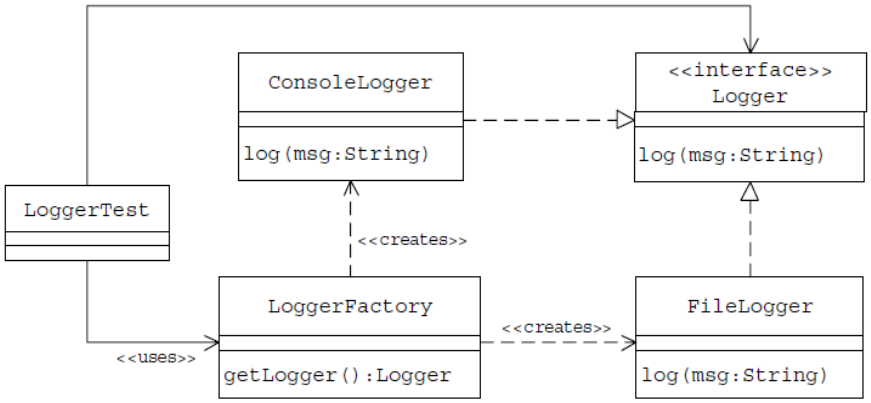
Thiết kế chức năng đưa ra log các thông điệp trong một ứng dụng. Log thông điệp thích hợp vào đúng thời điểm cần thiết đem lại nhiều ích lợi để bắt lỗi và theo dõi ứng dụng. Vì chức năng log thông điệp có thể cần đến cho nhiều client khác nhau, nên có thể cài đặt chức năng này trong một lớp *interface Logger*. Khi đó, các lớp cài đặt của Logger có thể xử ly để đưa ra thông điệp dưới dạng khác nhau cho các phương tiện khác nhau.

Ví dụ, hai cài đặt *FileLogger* có chức năng lưu thông điệp đến vào file log; *ConsoleLogger* hiển thị thông điệp đến trên màn hình. Hãy xét xem một đối tượng ứng dụng *LoggerTest* sử dụng các dịch vụ cung cấp bởi các cài đặt này sẽ như thế nào. Ta có thể xây dựng bằng cách sử dụng file tính chất *logger.properties*. Điều này đòi hỏi đối tượng ứng dụng phải biết sự tồn tại của *Logger* cũng như các lớp con của nó và cung cấp cài đặt để chọn và khởi tạo cài đặt *Logger* (Hình 8.3).



Hình 8.3: LoggerTest truy nhập trực tiếp

Khi áp dụng mẫu Factory method, việc chọn và khởi tạo *Logger* thích hợp có thể được đóng gói bên trong phương thức *getLogger* trong lớp *LoggerFactory* (Hình 8.4). *LoggerFactory* với phương thức *getLogger* đóng vai trò như *ConcreteCreator* như trong Hình 8.2.



Hình 8.4: LoggerTest truy nhập với LoggerFactory

Mã nguồn java cho cài đặt với factory method.

```
//Logger
public interface Logger {
public void log(String msg);
}
//FileLogger
public class FileLogger implements Logger {
public void log(String msg) {
FileUtil futil = new FileUtil();
futil.writeToFile("log.txt", msg, true, true);
}
}
//ConsoleLogger
public class ConsoleLogger implements Logger {
public void log(String msg) {
System.out.println(msg);
}
}
//LoggerFactory
public class LoggerFactory {
public boolean isFileLoggingEnabled() {
Properties p = new Properties();
try {
p.load(ClassLoader.getSystemResourceAsStream(
"Logger.properties"));
String fileLoggingValue =
p.getProperty("FileLogging");
if (fileLoggingValue.equalsIgnoreCase("ON") == true)
return true;
else
return false;
} catch (IOException e) {
return false;
}
```

```
}
//Factory Method
public Logger getLogger() {
if (isFileLoggingEnabled()) {
return new FileLogger();
} else {
return new ConsoleLogger();
}
}
}
//LoggerTest
public class LoggerTest {
public static void main(String[] args) {
LoggerFactory factory = new LoggerFactory();
Logger logger = factory.getLogger();
logger.log("A Message to Log");
}
}
```

8.2 MẪU THIẾT KẾ SINGLETON

8.2.1 Đặt vấn đề

Hãy xét tình huống một đối tượng có chức năng kết nối dữ liệu để cập nhật thông tin như thay đổi, thêm, xóa…Rõ ràng, khi đó chỉ có duy nhất một đối tượng trong vòng đời của ứng dụng này. Mẫu *Singleton* là lớp được sử dụng trong những trường hợp như thế để đảm bảo rằng có một và chỉ một thể hiện đối tượng. Lớp *Singleton* bảo đảm khởi tạo duy nhất bằng cách sử dụng tham chiếu đến chính nó.

8.2.2 Cấu trúc mẫu

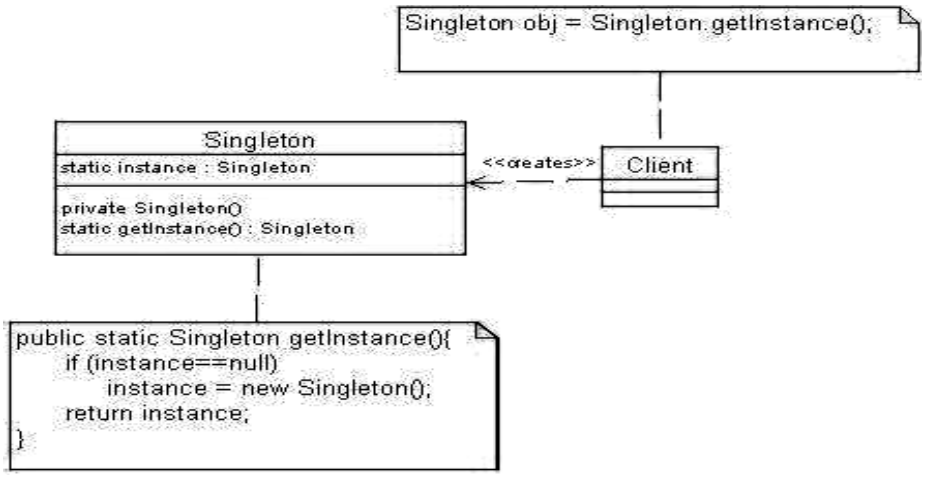
Biểu đồ lớp cho *Singleton* được thể hiện như trong Hình 8.5, trong đó hàm *getInstance()* đặt ở dạng *static* và khởi tạo đối tượng trong chính lớp *Singleton*.

8.2.3 Tình huống áp dụng

Mẫu *singleton* thông thường được cài đặt với các mẫu *abstract factory*, *builder*, *prototype*.

8.2.4 Ví dụ

Trở lại ví dụ trong 8.1 khi thiết kế với mẫu factory method. Cài đặt *FileLogger* của giao tiếp *interface Logger* nhằm chuyển thông điệp vào file *log.txt*. Nếu chỉ có một thể hiện vật lý của đối tượng đại diện thì sẽ tốt hơn. Trong một ứng dụng, khi những đối tượng client khác nhau truyền những thông điệp vào một file thì sẽ có khả năng là nhiều thể hiện của lớp *FileLogger* sử dụng bởi từng đối tượng client. Điều này có thể dẫn đến nhiều vấn đề do các đối tượng khác nhau đồng thời truy nhập vào cùng file.

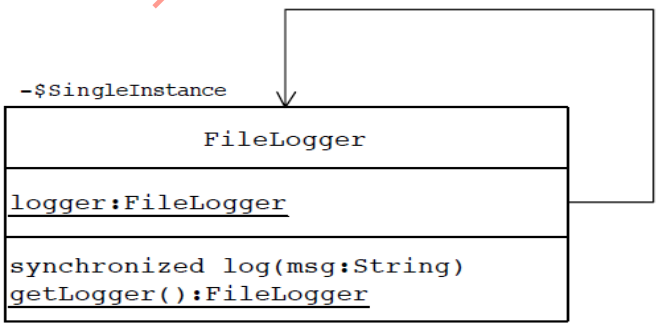


Hình 8.5: Biểu đồ lớp của singleton

Một trong những giải pháp là duy trì một thể hiện của lớp *FileLogger* trong biến toàn cục của ứng dụng. Thể hiện này có thể được truy nhập bởi tất cả client bằng cách cung cấp cho nó một điểm truy nhập toàn cục duy nhất. Tuy nhiên, cách làm như vậy không thể giải quyết trọn vẹn vấn đề. Một là không ngăn ngừa được client tạo các thể hiện mới từ lớp *FileLogger*. Hai là nó cũng không tránh được nhiều luồng trong cùng client thực thi phương thức *log*.

Một giải pháp khác là áp dụng khái niệm *giám sát* (đảm bảo không có hai luồng được phép truy nhập cùng một đối tượng tại cùng thời điểm) và khai báo phương thức *log(String)* là đồng bộ. Điều này tránh được nhiều luồng cùng thực thi một phương thức nhưng không ngăn các đối tượng client tạo nhiều thể hiện của lớp *FileLogger*.

Ngoài việc cần khai báo tính đồng bộ của phương thức *log*, chúng ta cũng phải cần đảm bảo rằng chỉ tồn tại một thể hiện của *FileLogger* suốt trong vòng đời của ứng dụng. Để thực hiện được việc này, chúng ta sẽ thay đổi lớp *FileLogger* thành lớp singleton (xem Hình 8.6).



Hình 8.6: Lớp FileLogger thể hiện như singleton

```
//Singleton Filelogger class
public class Filelogger implements Logger{
private static FileLogger logger;
//Ngăn ngừa client sử dụng cấu tử
```

```
private FileLogger(){
}
public static Filellogger getFileLogger(){
if (logger == null){
logger = new FileLogger();
}
return logger;
}
public synchronized void log(String msg){
FileUtile futil = new FileUtil();
Futil.writeToFile("log.txt", msg, true, true);
}
}
//Lớp LoggerFactory
public class LoggerFactory {
public boolean isFileLoggingEnabled() {
Properties p = new Properties();
try {
p.load(ClassLoader.getSystemResourceAsStream(
"Logger.properties"));
String fileLoggingValue =
p.getProperty("FileLogging");
if (fileLoggingValue.equalsIgnoreCase("ON") == true)
return true;
else
return false;
} catch (IOException e) {
return false;
}
}
//Factory Method
public Logger getLogger() {
if (isFileLoggingEnabled()) {
return new FileLogger();
```

```
} else {
return new ConsoleLogger();
}
}
}
```

Việc tạo private cho cấu tử *private FileLogger()* là nhằm ngăn các đối tượng client tạo đối tượng *FileLogger* nhưng những phương thức khác trong *FileLogger* có thể truy nhập được cấu tử này.

8.3 MẪU THIẾT KẾ ABSTRACT FACTORY

8.3.1 Đặt vấn đề

Trong các hệ điều hành hay phần mềm có giao diện đồ hoạ, thường có một bộ công cụ cung cấp một giao diện người dùng dựa trên chuẩn "nhìn và cảm nhận". Ví dụ, trong chương trình trình diễn tài liệu Powerpoint, có rất nhiều kiểu giao diện như vậy và cả những hành vi giao diện người dùng khác nhau được thể hiện ở đây như thanh cuộn tài liệu, cửa sổ, nút bấm, hộp soạn thảo...

Nếu xem chúng là các đối tượng thì chúng có một số đặc điểm và hành vi khá giống nhau về mặt hình thức nhưng lại khác nhau về cách thực hiện. Chẳng hạn đối tượng nút bấm và cửa sổ, hộp soạn thảo có cùng các thuộc tính là chiều rộng, chiều cao, toạ độ…và có các phương thức là *Resize(), SetPosition()*...Tuy nhiên, các đối tượng này không thể gộp chung được vào một lớp vì các đối tượng ở đây dù có cùng giao diện nhưng cách thực hiện các hành vi tương ứng lại hoàn toàn khác nhau.

Vấn đề đặt ra là có thể xây dựng một lớp tổng quát chứa những điểm chung của các đối tượng này để từ đó có thể dễ dàng sử dụng lại. Trong chương trình xây dựng Powerpoint, lớp *WidgetFactory* đã được xây dựng để các lớp của các đối tượng cửa sổ, nút bấm, hộp soạn thảo kế thừa từ lớp này.

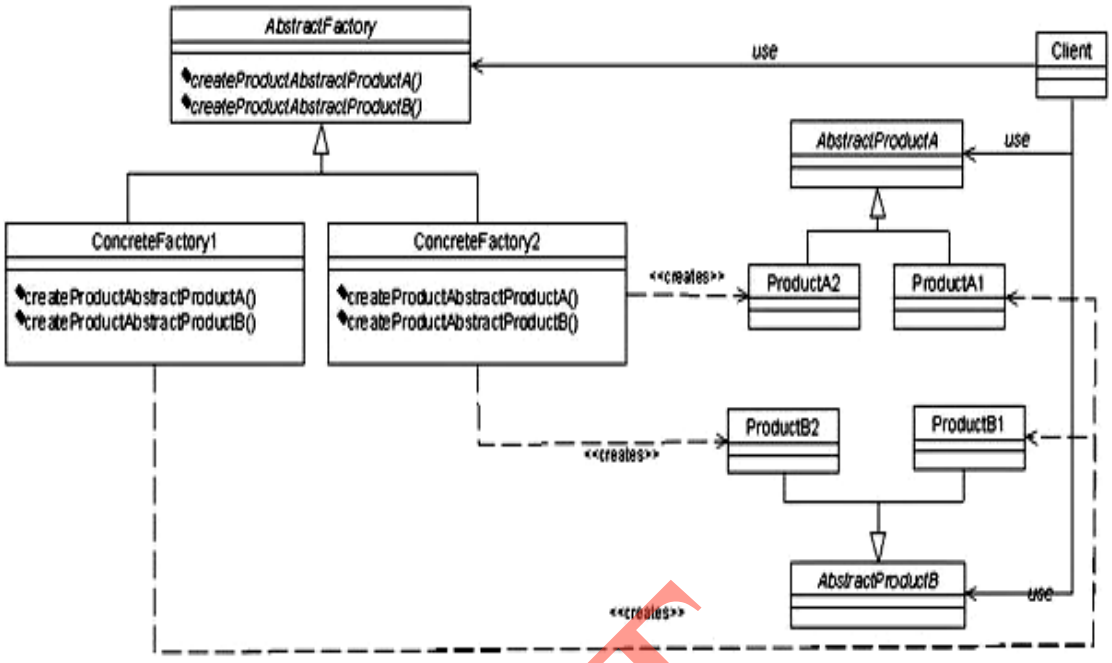
Mẫu *AbstractFactory* là một mẫu thiết kế nhằm cung cấp cho trình khách một giao tiếp *interrface* để cho các đối tượng thuộc các lớp khác nhau nhưng có chung giao tiếp có thể hoạt động mà không phải trực tiếp làm việc với từng lớp con cụ thể.

8.3.2 Cấu trúc mẫu

Cấu trúc mẫu được cho trong Hình 8.7. Trong đó:

- *AbstractFactory:* là lớp trừu tượng, tạo ra các đối tượng thuộc hai lớp trừu tượng là *AbstractProductA* và *AbstractProductB.*
- *ConcreteFactoryX:* là lớp kế thừa từ *AbstractFatory*, lớp này sẽ tạo ra một đối tượng cụ thể.

- *AbstractProduct:* là các lớp trừu tượng, các đối tượng cụ thể sẽ là các thể hiện của các lớp dẫn xuất từ lớp này.



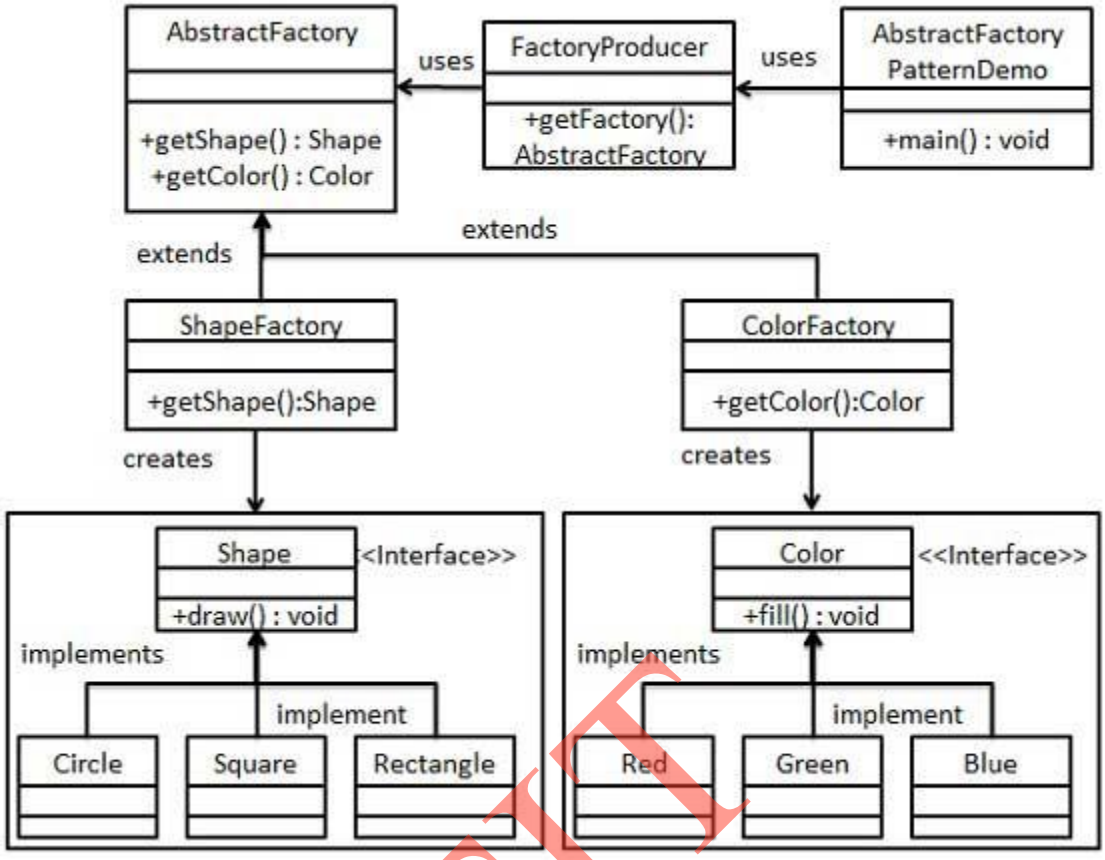
Hình 8.7: Cấu trúc Abstract Factory

8.3.3 Tình huống áp dụng

- Ứng dụng sẽ được cấu hình với một hoặc nhiều họ sản phẩm và các đối tượng cần phải được tạo ra như là một tập hợp để có thể tương thích với nhau.
- Phía trình khách không phụ thuộc vào việc những sản phẩm được tạo ra như thế nào.
- Chúng ta muốn cung cấp một tập các lớp và muốn thể hiện các ràng buộc, các mối quan hệ giữa chúng mà không phải là các thực thi của chúng.
- Mẫu abstract factory thường được cài đặt cùng với các mẫu singleton, factory method và đôi khi còn dùng với prototype.

8.3.4 Ví dụ

Chúng ta tạo ra các lớp giao tiếp *Shape* và *Color* và những lớp cụ thể cài đặt các giao tiếp này https://www.tutorialspoint.com/design_pattern/abstract_factory_pattern.htm). Sau đó tạo ra một factory trừu tượng *AbstractFactory*. Các lớp Factory *ShapeFactory* và *ColorFactory* là những lớp mở rộng của *AbstractFactory*. Đồng thời cũng tạo ra lớp *FactoryProducer* (Hình 8.8). Lớp *AbstractFactoryPatternDemo* sử dụng *FactoryProducer* để lấy đối tượng *AbstractFactory*. Nó sẽ truyền thông tin *CIRCLE / RECTANGLE / SQUARE* cho *Shape* đến *AbstractFactory* để lấy kiểu đối tượng cần thiết. Nó cũng truyền thông tin *RED / GREEN / BLUE* cho *Color* đến *AbstractFactory* để có được kiểu màu đối tượng cần.



Hình 8.8: Biểu đồ lớp cho factory trừu tượng của Shape và Color

Mã nguồn java của Hình 8.8 được cho dưới đây

```
public interface Shape {
   void draw();
}
public class Rectangle implements Shape {
   @Override
   public void draw() {
      System.out.println("Inside Rectangle::draw() method.");
   }
}
public class Square implements Shape {
   @Override
   public void draw() {
      System.out.println("Inside Square::draw() method.");
   }
}
```

```
public class Circle implements Shape {
   @Override
   public void draw() {
      System.out.println("Inside Circle::draw() method.");
   }
}

public class Green implements Color {
   @Override
   public void fill() {
      System.out.println("Inside Green::fill() method.");
   }
}
public class Blue implements Color {
   @Override
   public void fill() {
      System.out.println("Inside Blue::fill() method.");
   }
}
public abstract class AbstractFactory {
   abstract Color getColor(String color);
   abstract Shape getShape(String shape) ;
}
public class ShapeFactory extends AbstractFactory {
   @Override
   public Shape getShape(String shapeType){
      if(shapeType == null){
         return null;
      }
      if(shapeType.equalsIgnoreCase("CIRCLE")){
         return new Circle();
      }else if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();
      }else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }
```

```
      return null;
   }
      @Override
   Color getColor(String color) {
      return null;
   }
}
public class ColorFactory extends AbstractFactory {
   @Override
   public Shape getShape(String shapeType){
      return null;
   }
   @Override
   Color getColor(String color) {
      if(color == null){
         return null;
      }
      if(color.equalsIgnoreCase("RED")){
         return new Red();
      }else if(color.equalsIgnoreCase("GREEN")){
         return new Green();

      }else if(color.equalsIgnoreCase("BLUE")){
         return new Blue();
      }

      return null;
   }
}
public class FactoryProducer {
   public static AbstractFactory getFactory(String choice){
      if(choice.equalsIgnoreCase("SHAPE")){
         return new ShapeFactory();
```

```
      }else if(choice.equalsIgnoreCase("COLOR")){
         return new ColorFactory();
      }
      return null;
   }
}
public class AbstractFactoryPatternDemo {
   public static void main(String[] args) {
      //get shape factory
      AbstractFactory shapeFactory =
FactoryProducer.getFactory("SHAPE");
      //get an object of Shape Circle
      Shape shape1 = shapeFactory.getShape("CIRCLE");
      //call draw method of Shape Circle
      shape1.draw();
      //get an object of Shape Rectangle
      Shape shape2 = shapeFactory.getShape("RECTANGLE");
      //call draw method of Shape Rectangle
      shape2.draw();
      //get an object of Shape Square
      Shape shape3 = shapeFactory.getShape("SQUARE");
      //call draw method of Shape Square
      shape3.draw();
      //get color factory
      AbstractFactory colorFactory =
FactoryProducer.getFactory("COLOR");
      //get an object of Color Red
      Color color1 = colorFactory.getColor("RED");
      //call fill method of Red
      color1.fill();
      //get an object of Color Green
      Color color2 = colorFactory.getColor("Green");

      //call fill method of Green
      color2.fill();
```

```
        //get an object of Color Blue
        Color color3 = colorFactory.getColor("BLUE");
        //call fill method of Color Blue
        color3.fill();
    }
}
```

8.4 MẪU THIẾT KẾ BUILDER

8.4.1 Đặt vấn đề

Các ứng dụng lớn thường có nhiều chức năng phức tạp và giao diện đồ sộ. Khi đó, việc khởi tạo ứng dụng thường gặp nhiều khó khăn. Nếu dồn tất cả công việc này cho một hàm khởi tạo thì sẽ rất khó kiểm soát và hơn nữa không phải lúc nào các thành phần ứng dụng cũng được khởi tạo một cách đồng bộ. Vì có những thành phần được tạo ra vào lúc dịch chương trình nhưng cũng có thành phần được tạo ra tuỳ theo từng yêu cầu của người dùng hay ngữ cảnh của ứng dụng. Do vậy, cách tốt nhất là giao công việc này cho một đối tượng chịu trách nhiệm khởi tạo và phân chia việc khởi tạo ứng dụng để có thể tiến hành khởi tạo một cách riêng biệt ở các hoàn cảnh khác nhau.

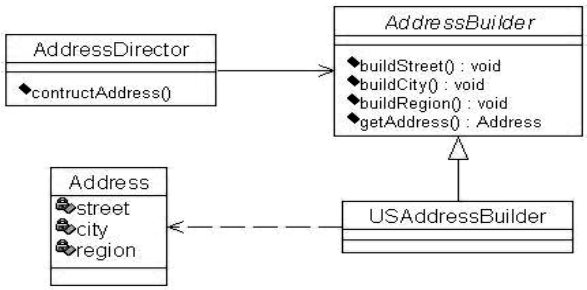
Hãy tưởng tượng việc tạo ra đối tượng giống như việc chúng ta tạo ra chiếc xe đạp. Đầu tiên là tạo ra khung xe, sau đó là bánh xe, xích, líp...Việc tạo ra các bộ phận này không nhất thiết phải được thực hiện một cách đồng thời hay theo một trật tự nào cả và nó có thể được tạo ra một cách độc lập bởi nhiều người. Nhưng trong một mô hình sản xuất như vậy, bao giờ việc tạo ra chiếc xe cũng được khép kín để tạo ra chiếc xe hoàn chỉnh, đó là nhà máy sản xuất xe đạp. Ta gọi đối tượng nhà máy sản xuất xe đạp này là *người xây dựng* (builder)

Builder là mẫu thiết kế được tạo ra để chia công việc khởi tạo một đối tượng phức tạp thành khởi tạo các đối tượng riêng rẽ để từ đó có thể tiến hành khởi tạo đối tượng trong các ngữ cảnh khác nhau.

8.4.2 Cấu trúc mẫu

Cấu trúc mẫu được cho trong Hình 8.9. Trong đó:

- *AddressDirector* là lớp tạo ra đối tượng *Address*
- *AddressBuilder* là lớp trừu tượng cho phép tạo ra một đối tượng *Address* bằng các phương thức khởi tạo từng thành phần của *Address*.
- *USAddressBuilder* là lớp tạo ra các *Address*. *USAddressBuilder* sẽ tạo ra địa chỉ theo chuẩn của USA.



Hình 8.8: Biểu đồ lớp với Builder

8.4.3 Tình huống áp dụng

Mẫu Builder thường được cài đặt cùng với các mẫu như Abstract Factory. Ý nghĩa quan trọng của Abstract Factory là tạo ra một dòng các đối tượng dẫn xuất (cả đơn giản và phức tạp). Ngoài ra Builder còn thường được cài đặt kèm với mẫu Composite. Mẫu Composite thường là những gì mà Builder tạo ra.

8.4.4 Ví dụ

Xét lớp Address với các thành phần như sau: Street, City và Region. Ta phân rã việc khởi tạo một đối tượng Address thành các phần: buildStreet, buildCity và buildRegion.

```
Address.java
Address.java
class Address{
private String street;
private String city;
private String region;
/**
 * @return the city
 */
public String getCity() {
return city;
}
/**
 * @param city the city to set
 */
public void setCity(String city) {
this.city = city;
}
/**
 * @return the region
 */
public String getRegion() {
return region;
}
/**
 * @param region the region to set
```

```
 */
public void setRegion(String region) {
this.region = region;
}
/**
 * @return the street
 */
public String getStreet() {
return street;
}
/**
 * @param street the street to set
 */
public void setStreet(String street) {
this.street = street;
}
}
AddressBuilder.java

abstract class AddressBuilder{
abstract public void buildStreet(String street){}
abstract public void buildCity(String city){}
abstract public void buildRegion(String region){}
}

USAddressBuilder.java

class USAddressBuilder extends AddressBuilder {
private Address add;
public void buildStreet(String street){
add.setStreet(street);
}
public void buildCity(String city){
add.setCity(city);
}
public void buildRegion(String region){
add.setRegion(region);
}
public Address getAddress(){
return add;
}
}

AddressDirector.java

class AddressDirector{
public void Contruct(AddressBuilder builder, String street,
```

```
String city, String region){
builder.buildStreet(street);
builder.buildCity(city);
builder.buildRegion(region);
}
}

Client.java
class Client{
AddressDirector director = new AddressDirector();
USAddressBuilder b = new USAddressBuilder();
director.Contruct(b. "Street 01", "City 01", "Region 01");
Address add = b.getAddress();
}
```

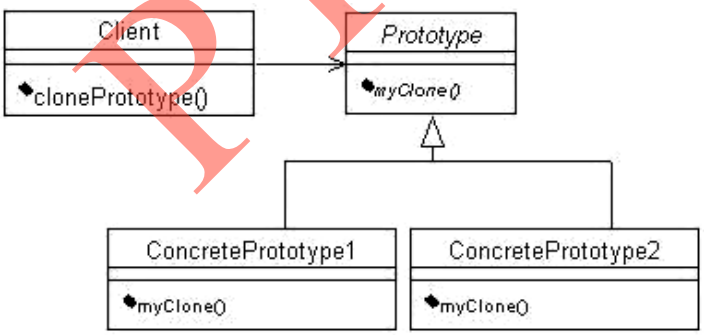
8.5 MẪU THIẾT KẾ PROTOTYPE

8.5.1 Đặt vấn đề

Như đã trình bày trong các mục trên, cả hai mẫu factory method và abstract factory cho phép tiến trình độc lập tạo ra các đối tượng. Prototype cũng giải quyết vấn đề tương tự nhưng theo cách khác linh hoạt hơn. Nó là mẫu thiết kế có thể chỉ định một đối tượng đặc biệt để khởi tạo. Nó sử dụng một thể hiện cơ bản rồi sau đó sao chép ra các đối tượng khác từ mẫu đối tượng này.

8.5.2 Cấu trúc mẫu

Cấu trúc mẫu được cho trong Hình 8.10



Hình 8.10: Biểu đồ lớp cho prototype

Trong đó:

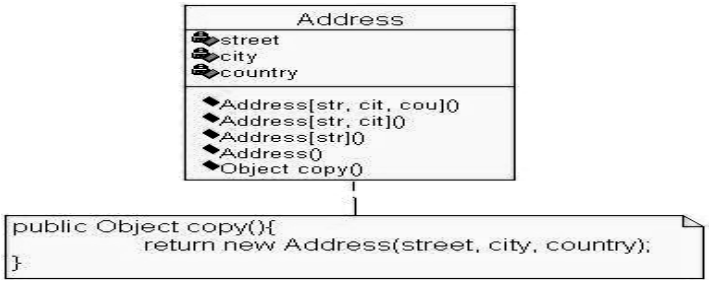
- *Prototype* là lớp trừu tượng cài đặt phương thức *myClone()* – một phương thức copy bản thân đối tượng đã tồn tại.
- *ConcretePrototype1* và *ConcretePrototype2* là các lớp kế thừa từ lớp *Prototype*.

8.5.3 Tình huống áp dụng

Khi muốn khởi tạo một đối tượng bằng cách sao chép từ một đối tượng đã tồn tại. Mặc dù đôi khi đối chọi nhau, mẫu prototype và abstract factory có thể kết hợp với nhau.

8.5.4 Ví dụ

Hệ quản ly địa chỉ (Hình 8.11).



Hình 8.11: Biểu đồ lớp Address với prototype

8.6 KẾT LUẬN

Chương này đã trình bày một số mẫu thiết kế tạo dựng nhằm tạo ra các đối tượng với những đặc trưng phù hợp với các tình huống mong muốn. Điều này sẽ đem lại hiệu quả và chất lượng cho thiết kế. Các mẫu đều kèm theo biểu đồ và ví dụ áp dụng để có thể hình hung cụ thể cách cài đặt các mẫu này trong thực tế.

BÀI TẬP

1. Thêm logger mới DBLogger trong Ví dụ 8.1 để log thông điệp vào cơ sở dữ liệu.
2. Tạo lớp con của lớp *LoggerFactory* và cài đè phương thức *getLogger* để cài đặt một quy tắc chọn lớp khác.
3. Bổ sung thêm các lớp địa chỉ trong Hình 8.8 và viết một chương trình để chạy kết quả
4. Hãy xây dựng ứng dụng quản lý dữ liệu khách hàng dựa trên mẫu abstract factory với yêu cầu như sau:
   - Chức năng cơ bản là xác thực và lưu dữ liệu khách hàng nhập vào gồm: account, địa chỉ và dữ liệu thẻ tín dụng
   - Ứng dụng phải hoạt động được trong cả hai mode là máy bàn và qua mạng
   - Ứng dụng có thể sử dụng RMI và lưu dữ liệu vào máy chủ trung tâm. Khi máy chủ từ xa không hoạt động, người sử dụng có thể thao tác với máy để bàn
5. Hoàn thiện ví dụ trong mẫu thiết kế Builder
6. Thiết kế và cài đặt lớp kết nối dữ liệu như singleton
7. Cài đặt mẫu prototype trong ví dụ 8.5.4

CHƯƠNG 9: CÁC MẪU THIẾT KẾ CẤU TRÚC

Chương này tập trung trình bày một số mẫu thiết kế cấu trúc, nội dung bao gồm các mẫu thiết kế sau đây:

- Mẫu Adapter
- Mẫu Bridge
- Mẫu Composite
- Mẫu Decorator
- Mẫu Façade
- Mẫu Flyweight
- Mẫu Proxy

9.1 MẪU ADAPTER

9.1.1 Đặt vấn đề

Xét hai tình huống sau đây:

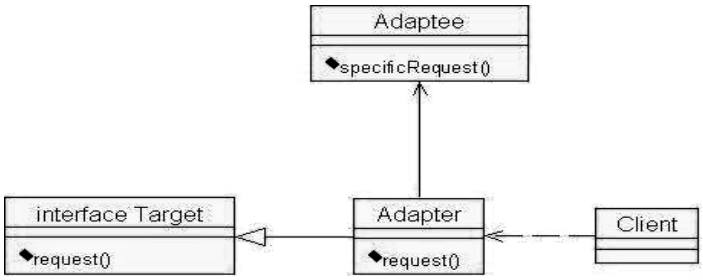
- Một lớp công cụ được thiết kế nhằm sử dụng lại nhưng không thể dùng được do giao diện của nó không thích hợp với ứng dụng được yêu cầu trong một miền nào đó. Adapter đã đưa ra một giải pháp cho vấn đề này.
- Ta muốn sử dụng một lớp đã tồn tại, tuy nhiên giao diện của nó không phù hợp với giao diện của một lớp mà ta yêu cầu. Vì vậy ta muốn tạo ra một lớp có khả năng dùng lại bằng cách kết hợp lớp đó với các lớp không liên quan hoặc không được dự đoán trước và có giao diện tương thích. Với đối tượng adapter, ta cần sử dụng một vài lớp con đã tồn tại. Nhưng để làm cho các giao diện của chúng tương thích với nhau bằng việc phân lớp các giao diện đó là việc làm không thực tế, để làm được điều này ta dùng một đối tượng adapter để biến đổi giao diện lớp cha của nó.

Adapter là mẫu thiết kế dùng để biến đổi giao diện của một lớp thành một giao diện khác mà client yêu cầu. Nó giúp cho các lớp không thể hoạt động với nhau bằng cách biến đổi giao diện sao cho chúng có thể tương thích với nhau.

9.1.2 Cấu trúc mẫu

Cấu trúc được cho trong Hình 9.1, trong đó:

- *Target* là một *Interface* xác định các chức năng, yêu cầu mà Client cần sử dụng
- *Adaptee* là lớp chứa các chức năng mà *Target* cần sử dụng nhằm tạo ra các chức năng mà *Target* cần cung cấp cho Client.
- *Adapter* cài đặt phương thức từ *Target* và sử dụng đối tượng lớp *Adaptee, Apdater* có nhiệm vụ gắn kết *Adaptee* vào *Target* để có được chức năng mà Client mong muốn.



Hình 9.1: Cấu trúc Adapter

9.1.3 Tình huống áp dụng

- Khi ta muốn sử dụng một lớp có sẵn nhưng cổng giao tiếp của nó không tương thích với yêu cầu hiện tại.
- Muốn tạo một lớp tái sử dụng có thể hoạt động được với những lớp khác không liên hệ gì với nó và không nhất thiết phải tương thích qua cổng giao tiếp.

9.1.4 Ví dụ

Một hệ thống quản lý Phone cần thực hiện một số chức năng nào đó, trong đó có một phương thức trả về số điện thoại tương ứng với một xâu ký tự đầu vào. Giả sử trước đó ta đã có một lớp có chức năng chuyển các ký tự số từ một xâu và ta muốn sử dụng chức năng đó vào hệ thống lấy số điện thoại. Mã nguồn được cho dưới đây:

```
//Giao tiếp PhoneTarget
public interface PhoneTarget{
        public String getPhoneNumber(String message);
        //lấy số điện thoại trong một xâu
}
public GetNumberAdaptee{
        public String getNumber(String str){
                //lấy ra dạng số trong một xâu
                <…get number>
        }
        …
}
public Adapter implements PhoneTarget{
        public String getPhoneNumber(String message){
                GetNumberAdaptee obj = new GetNumberAdaptee;
                String str = obj.getNumber(message);
                return "84"+str;
        }
}
```


9.2 MẪU BRIDGE

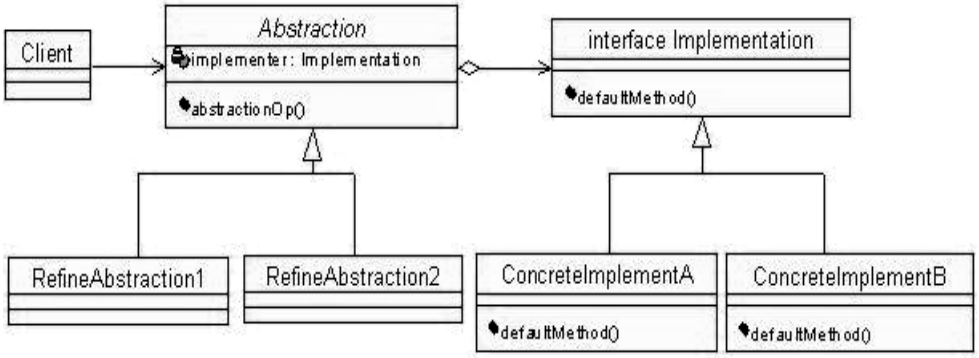
9.2.1 Đặt vấn đề

Khi một lớp trừu tượng cần bổ sung thêm một vài thành phần thì cách thông thường là sử dụng kế thừa. Nghĩa là định nghĩa một giao tiếp cho lớp trừu tượng đó và các lớp con cụ thể thực hiện nó theo các cách khác nhau. Nhưng cách tiếp cận như vậy thường là không

đủ mềm dẻo. Do tính kế thừa ràng buộc thành phần cần bổ sung thêm, nên sẽ khó thay đổi, mở rộng, và sử dụng lại các lớp trừu tượng. Trong trường hợp như vậy, sử dụng một mẫu Bridge là thích hợp nhất. Bridge là mẫu thiết kế sử dụng Interface để tách riêng cài đặt phương thức của một lớp trừu tượng để hai đối tượng đó có thể biến đổi độc lập với nhau.

9.2.2 Cấu trúc mẫu

Cấu trúc được cho trong Hình 9.2



Hình 9.2: Mẫu Bridge

Trong đó:

- *Abstraction:* là một lớp trừu tượng khai báo các chức năng và cấu trúc cơ bản. Lớp này có một thuộc tính là một thể hiện của giao tiếp *interface Implementation*, thể hiện sẽ thực hiện các chức năng *abstractionOp()* của lớp Abstraction.
- *Interface Implementation:* là giao tiếp thực thi của lớp các chức năng nào đó của Abstraction.
- *RefineAbstractionX:* là định nghĩa các chức năng mới hoặc các chức năng đã có trong Abstraction.
- *ConcreteImplementY:* là các lớp định nghĩa tường minh các thực thi trong lớp giao tiếp Implementation

9.2.3 Tình huống áp dụng

Mẫu Bridge thường được áp dụng khi :

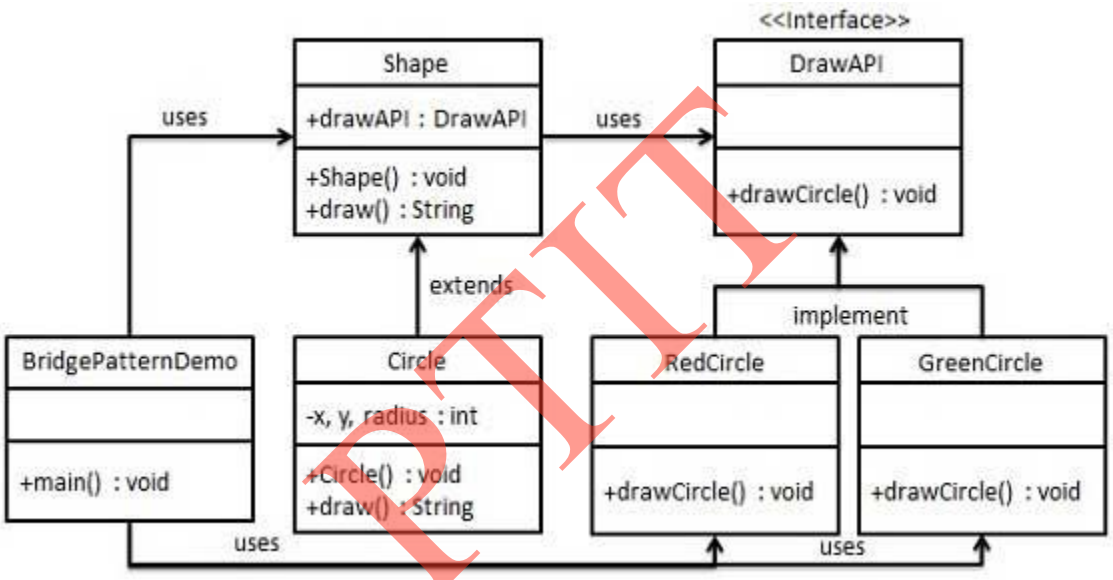
- Ta muốn tránh một ràng buộc cố định giữa một abstraction và một thành phần bổ sung thêm của nó. Các abstraction và các thành phần cài đặt của chúng nên có khả năng mở rộng bằng việc phân chia lớp. Trong trường hợp này, Bridge pattern cho phép ta kết hợp các abstraction và các thành phần bổ sung thêm khác nhau và mở rộng chúng một cách độc lập.
- Thay đổi trong thành phần được bổ sung thêm của một abstraction mà không ảnh hưởng đối với các client, tức là mã của chúng không nên biên dịch lại. Nghĩa là ta muốn làm ẩn đi hoàn toàn các thành phần bổ sung thêm của một abstraction khỏi các client.
- Ta có một sự phát triển rất nhanh các lớp, hệ thống phân cấp lớp chỉ ra là cần phải tách một đối tượng thành hai phần. Khi đó ta muốn tạo ra sự mềm dẻo giữa hai

thành phần ảo và thực thi của một thành phần, và tránh đi mối quan hệ tĩnh giữa chúng

Bridge có một cấu trúc tương tự như một đối tượng của Adapter, nhưng Bridge có mục đích khác. Nó chia giao diện từ các phần cài đặt của nó ra riêng rẽ để từ đó có thể linh hoạt hơn và độc lập nhau trong khi Adapter đồng nghĩa với việc thay đổi giao diện của một đối tượng đã tồn tại. Ví dụ sau đây sẽ chỉ ra việc sử dụng mẫu Bridge để vẽ hình tròn có màu khác nhau bằng cách dùng cùng một phương thức của lớp trừu tượng nhưng thể hiện cài đặt khác nhau qua lớp giao tiếp Bridge.

9.2.4 Ví dụ

Interface *DrawAPI* thể hiện như là cầu nối Bridge và các lớp cụ thể *RedCircle, GreenCircle* cài đặt giao tiếp này. Shape là một lớp trừu tượng và sẽ sử dụng đối tượng của *DrawAPI*. Lớp *BridgePatternDemo* sẽ sử dụng lớp *Shape* để vẽ hình tròn với màu khác nhau [23].



Hình 9.3: Sử dụng Bridge để vẽ hình tròn có màu khác nhau

```
public interface DrawAPI {
   public void drawCircle(int radius, int x, int y);
}
public class RedCircle implements DrawAPI {
   @Override
   public void drawCircle(int radius, int x, int y) {
      System.out.println("Drawing Circle[ color: red, radius: " +
radius + ", x: " + x + ", " + y + "]");
   }
}
public class GreenCircle implements DrawAPI {
```

```
   @Override
   public void drawCircle(int radius, int x, int y) {
      System.out.println("Drawing Circle[ color: green, radius: "
+ radius + ", x: " + x + ", " + y + "]");
   }
}
public abstract class Shape {
   protected DrawAPI drawAPI;

   protected Shape(DrawAPI drawAPI){
      this.drawAPI = drawAPI;
   }
   public abstract void draw();
}
public class Circle extends Shape {
   private int x, y, radius;

   public Circle(int x, int y, int radius, DrawAPI drawAPI) {
      super(drawAPI);
      this.x = x;
      this.y = y;
      this.radius = radius;
   }

   public void draw() {
      drawAPI.drawCircle(radius,x,y);
   }
}
public class BridgePatternDemo {
   public static void main(String[] args) {
      Shape redCircle = new Circle(100,100, 10, new RedCircle());
      Shape greenCircle = new Circle(100,100, 10, new
GreenCircle());

      redCircle.draw();
      greenCircle.draw();
```

```
    }
}
```

9.3 MẪU COMPOSITE

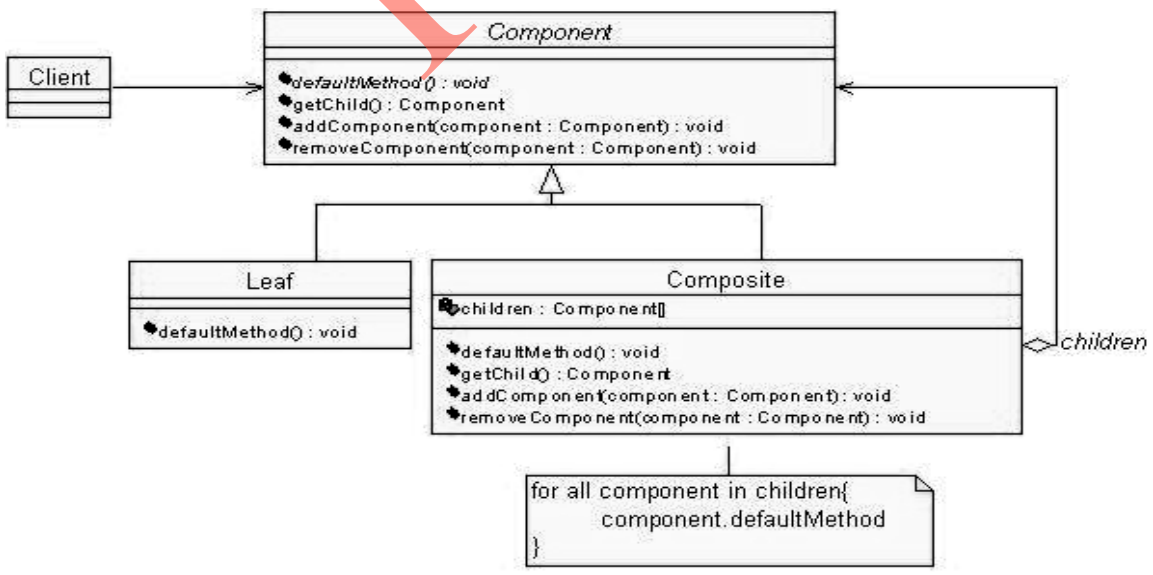
9.3.1 Đặt vấn đề

Các ứng dụng đồ họa như bộ công cụ soạn thảo hình vẽ và các hệ thống lưu giữ biểu đồ cho phép người sử dụng xây dựng các biểu đồ phức tạp khác xa với các thành phần nguyên thủy và đơn giản ban đầu. Người sử dụng có thể nhóm một số các thành phần để tạo ra các thành phần khác lớn hơn, và các thành phần lớn hơn này lại có thể được nhóm lại để tạo ra các thành phần lớn hơn nữa. Một cài đặt đơn giản là xác định các lớp cho các thành phần đồ họa cơ bản như Text và Line, cộng với các lớp khác cho phép hoạt động như các khuôn chứa các thành phần cơ bản đó.

Nhưng có một vấn đề với cách tiếp cận này là mã sử dụng các lớp đó phải tác động lên các đối tượng nguyên thủy và các đối tượng bao hàm các thành phần nguyên thủy ấy là khác nhau ngay cả khi thời gian người sử dụng tác động lên chúng là như nhau. Sự phân biệt các đối tượng này làm cho ứng dụng trở nên phức tạp hơn. Mẫu Composite xem xét việc sử dụng các thành phần đệ quy để làm cho các client không có sự phân biệt trên.

Giải pháp của mẫu Composite là xây dựng một lớp trừu tượng để biểu diễn cả hai thành phần nguyên thủy và các lớp chứa chúng. Lớp này cũng xác định các thao tác truy nhập và quản lý các đối tượng con của nó. Như vậy, Composite là mẫu thiết kế dùng để tạo ra các đối tượng trong các cấu trúc cây để biểu diễn hệ thống phân lớp: bộ phận – toàn bộ. Composite cho phép các client tác động đến từng đối tượng và các thành phần của đối tượng một cách thống nhất.

9.3.2 Cấu trúc mẫu



Hình 9.4: Mẫu composite

Trong đó:

- *Component*: là một giao tiếp interface định nghĩa các phương thức cho tất cả các phần của cấu trúc cây. Nó có thể được thực thi như một lớp trừu tượng khi ta cần cung cấp các hành vi cho tất cả các kiểu con. Bình thường, các *Component* không có các thể hiện, các lớp con hoặc các lớp thực thi, nhưng nó có thể hiện và được sử dụng để tạo nên cấu trúc cây.
- *Composite*: là lớp được định nghĩa bởi các thành phần mà nó chứa. *Composite* chứa một nhóm các *Component*, vì vậy nó có các phương thức để thêm vào hoặc loại bỏ các thể hiện của *Component*. Những phương thức được định nghĩa trong *Component* được thực thi để thực hiện các hành vi đặc tả cho lớp *Composite* và để gọi lại phương thức đó trong các đỉnh của nó. Lớp *Composite* được gọi là lớp nhánh hay lớp chứa.
- *Leaf*: là lớp thực thi từ giao tiếp *Component*. Sự khác nhau giữa lớp *Leaf* và *Composite* là lớp *Leaf* không chứa các tham chiếu đến các *Component* khác, lớp *Leaf* đại diện cho mức thấp nhất của cấu trúc cây

9.3.3 Tình huống áp dụng

- Khi có một mô hình thành phần với cấu trúc nhánh-lá, toàn bộ-bộ phận, ...hay khi cấu trúc có thể có vài mức phức tạp và động.
- Mẫu thường dùng làm thành phần liên kết đến đối tượng cha là dây chuyền trách nhiệm (Chain of Responsibility) sẽ được trình bày sau. Mẫu Decorator cũng thường được sử dụng với Composite. Khi Decorator và Composite cùng được sử dụng với nhau, chúng thường sẽ có một lớp cha chung. Vì vậy Decorator sẽ hỗ trợ thành phần giao diện với các phương thức như Add, Remove và GetChild. Mẫu Flyweight giúp cho chúng ta chia sẻ thành phần, nhưng chúng sẽ không tham chiếu đến cha của chúng. Mẫu Iterator có thể dùng để duyệt mẫu Composite. Mẫu Visitor định vị thao tác và hành vi nào sẽ được phân phối qua các lớp lá và Composite. Các mẫu liên quan sẽ được làm sáng to trong các phần tiếp theo.

9.3.4 Ví dụ

Ta hãy xem xét ví dụ dự án *Project*, một *Project* là một hợp nhiều tác vụ *Task* (Leaf), ta cần tính tổng thời gian của dự án thông qua thời gian của tất cả các tác vụ. Mã nguồn được cho trong bảng sau đây:

```
public interface TaskItem{
    public double getTime();
}
public class Project implements TaskItem{
    private String name;
    private ArrayList subtask = new ArrayList();
    public Project(){ }
    public Project(String newName){
        name = newName;
    }
```

```
    public String getName(){ return name; }
    public ArrayList getSubtasks(){ return subtask; }
    public double getTime(){
        double totalTime = 0;
        Iterator items = subtask.iterator();
        while(items.hasNext()){
            TaskItem item = (TaskItem)items.next();
            totalTime += item.getTime();
        }
        return totalTime;
    }

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

    public void addTaskItem(TaskItem element){
        if (!subtask.contains(element)){
            subtask.add(element);
        }
    }
    public void removeTaskItem(TaskItem element){
        subtask.remove(element);
    }
}
public class Task implements TaskItem{
    private String name;
    private double time;

    public Task(){ }
    public Task(String newName, double newTimeRequired){
        name = newName;
        time = newTimeRequired;
    }

    public String getName(){ return name; }
    public double getTime(){
        return time;
    }
        public void setName(String newName){ name = newName; }
    public void setTime(double newTimeRequired){ time =
newTimeRequired; }
}
```


9.4 MẪU DECORATOR

9.4.1 Đặt vấn đề

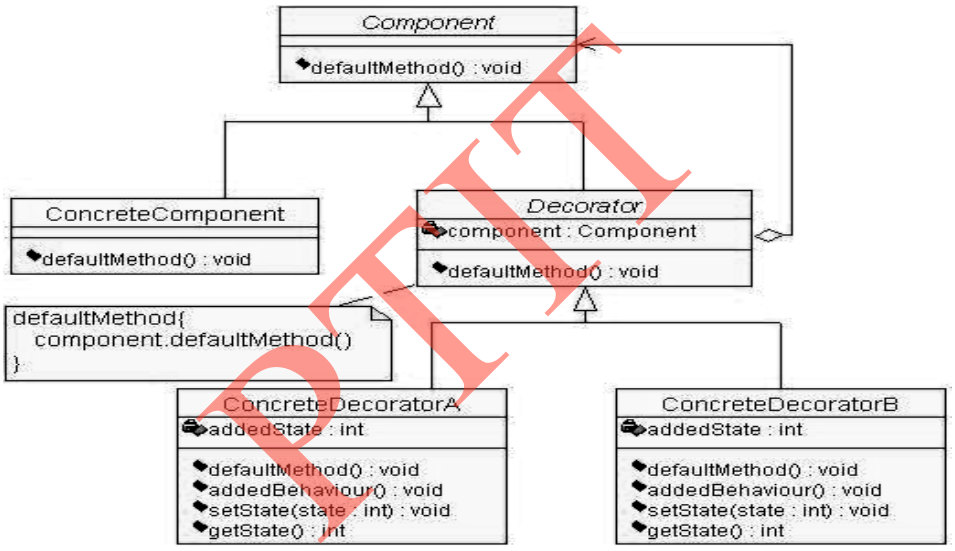
Mẫu decorator được sử dụng để mở rộng chức năng của đối tượng mà không phải thay đổi mã nguồn ban đầu hay sử dụng kế thừa. Điều này có thể thực hiện được bằng cách tạo một

đối tượng bao phủ quanh đối tượng đã có với việc gắn các chức năng bổ sung cho các đối tượng đó (gán động).

9.4.2 Cấu trúc mẫu

Cấu trúc mẫu của decorator được cho trong Hình 9.5. Trong đó:

- *Component:* là một interface chứa các phương thức ảo (ở đây là defaultMethod)
- *ConcreteComponent:* là một lớp kế thừa từ Component, cài đặt các phương thức cụ thể (defaultMethod được cài đặt tường minh)
- *Decorator:* là một lớp ảo kế thừa từ Component đồng thời cũng chứa một thể hiện của Component, phương thức defaultMethod trong Decorator sẽ được thực hiện thông qua thể hiện này.
- *ConcreteDecoratorX:* là các lớp kế thừa từ Decorator, khai báo tường minh các phương thức, đặc biệt trong các lớp này khai báo tường minh các "trách nhiệm" cần thêm vào khi trong thời gian chạy.



Hình 9.5: Mẫu decorator

9.4.3 Tình huống áp dụng

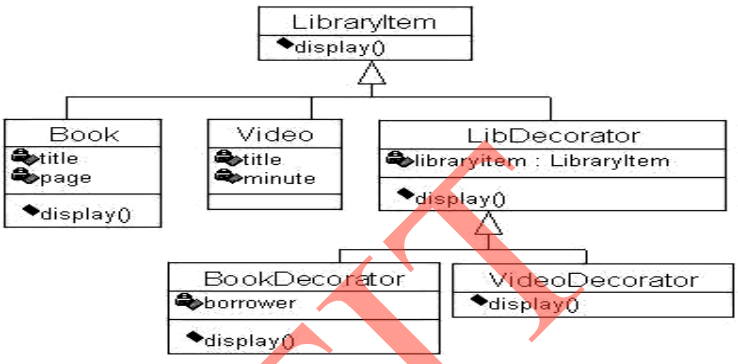
- Khi chúng ta muốn thay đổi động mà không ảnh hưởng đến người dùng và không phụ thuộc vào giới hạn các lớp con; muốn một thành phần có thể thêm vào hoặc loại bỏ khi hệ thống đang chạy. Hoặc khi một số đặc tính phụ thuộc mà bạn muốn áp dụng một cách động và muốn kết hợp chúng vào trong một thành phần.
- Mẫu Decorator khác với Adapter, Decorator chỉ thay đổi nhiệm vụ của đối tượng mà không thay đổi giao diện của nó như Adapter. Adapter mang đến cho đối tượng một giao diện mới hoàn toàn. Decorator cũng có thể coi như một Composite bị thoái hoá với duy nhất một thành phần.
- Decorator kết hợp thêm phần nhiệm vụ vào đối tượng và cho phép chúng ta thay đổi bề ngoài của một đối tượng. Trong khi đó, mẫu strategy (sẽ được trình bày

trong Chương 10) cho phép chúng ta thay đổi bên trong của đối tượng. Chúng là hai cách có thể thay phiên nhau để ta thay đổi một đối tượng.

9.4.4 Ví dụ

Giả sử trong thư viện có các tài liệu: sách, video...Các loại tài liệu này có các thuộc tính khác nhau, phương thức hiển thị của giao tiếp *LibraryItem* sẽ khác nhau. Giả sử, ngoài các thông tin về các thuộc tính trên, đôi khi ta muốn hiển thị thông tin về độc giả đã mượn một cuốn sách nào đó (chức năng hiển thị độc giả này không phải lúc nào cũng muốn hiển thị), hoặc muốn xem một đoạn video (không thường xuyên). Decorator sẽ hộ trợ xây dựng đối tượng thực hiện nhiệm vụ này (Hình 9.6).

Giao tiếp *interface LibraryItem* định nghĩa phương thức *display()* cho tất cả các tài liệu của thư viện và các lớp tài liệu tương ứng. Các decorator cho *Book* và *Video* kế thừa từ decorator của thư viện *LibDecorator*.



Hình 9.6: Sử dụng Decorator cho hệ quản lý thư viện

Mã nguồn được cho dưới đây.

```
//Định nghĩa giao tiếp
public interface LibraryItem{
	public void display();	//đây là defaultMethod
	}
//Định nghĩa các lớp tài liệu
public class Book implements LibraryItem{
	private String title;
	private int page;
	public Book(String s, int p){
		title = s;
		page = p;
	}
	public void display(){
		System.out.println("Title: " + title);
		System.out.println("Page number: " + page);
	}
}
public class Video implements LibraryItem{
	private String title;
```

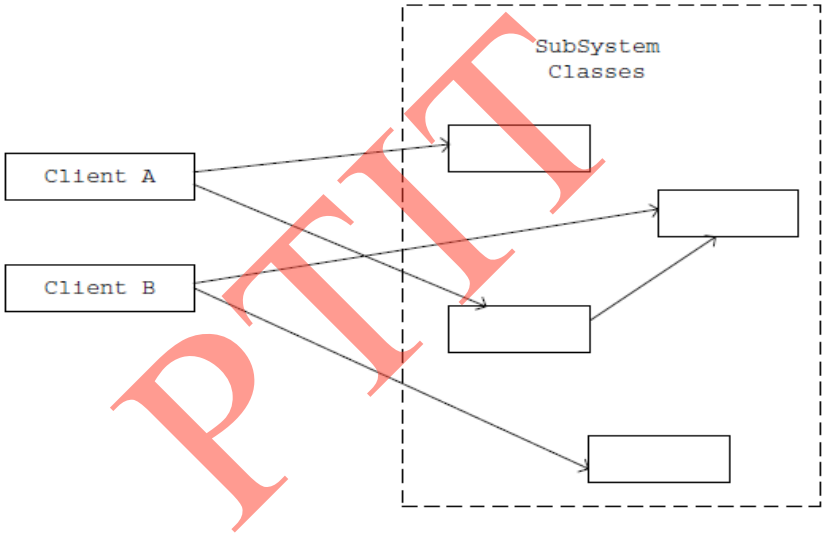
```
	private int minutes;
	public Video(String s, int m){
		title = s;
		minutes = m;
	}
	public void display(){
		System.out.println("Title: " + title);
		System.out.println("Time: " + minutes);
	}
}
//Lớp trừu tượng Decorator thư viện
public abstract class LibDecorator implements LibraryItem{
	private LibraryItem libraryitem;
	public LibDecorator(LibraryItem li){
		libraryitem = li;
	}
	public void display(){
		libraryitem.display();
	}
}

//Các lớp Decorator cho mỗi tài liệu thư viện cần bổ sung

//trách nhiệm ở thời điểm chạy

public class BookDecorator extends LibDecorator{
	private String borrower;
	public BookDecorator(LibraryItem li, String b){
		super(li);
		borrower = b;
	}
	public void display(){
		super.display();
		System.out.println("Borrower: " + borrower);
	}
}
public class VideoDecorator extends LibDecorator{
	public VideoDecorator(LibraryItem li){
		super(li);
	}
	public void display(){
		super.display();
		play(); //phương thức play video
	}
}
```

9.5 MẪU FAÇADE

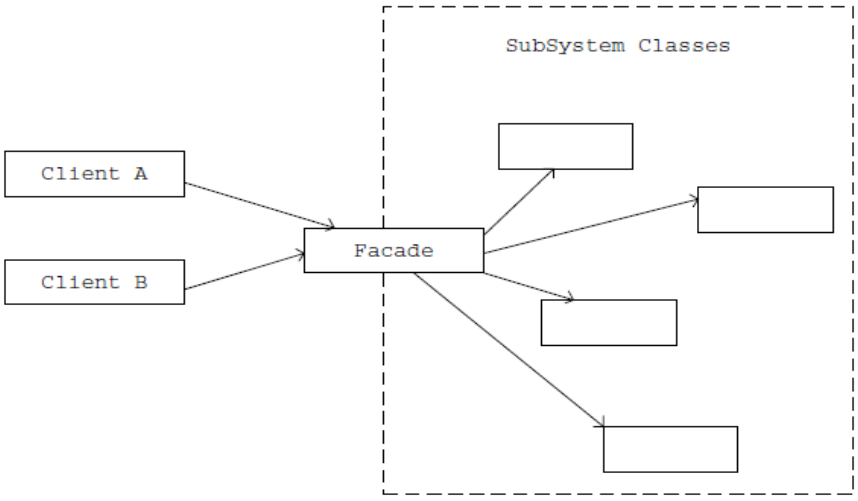
9.5.1 Đặt vấn đề

Thông thường để giảm độ phức tạp, ta sẽ phân rã một hệ thống thành các hệ thống con trong đó mỗi hệ thống con bao gồm các lớp liên kết nhau để cung cấp các chức năng nào đó. Ví dụ, lớp *Account*, địa chỉ *Address* và thẻ tín dụng *CreditCard* hoạt động cùng nhau để cung cấp các chức năng trực tuyến cho khác hàng.

Thông thường thiết kế một hệ thống là phải tuân theo nguyên lý sao cho tối thiểu hóa được sự giao tiếp và phụ thuộc giữa các hệ thống con. Một cách để đạt được mục tiêu này là đưa ra đối tượng façade nhằm cung cấp một giao diện để client dễ dàng giao tiếp với hệ thống con. Nghĩa là khi đó các client sẽ tương tác với đối tượng façade thay vì tương tác với các hệ thống con và vai trò tương tác với các hệ thống con do façade đảm nhiệm. Mẫu façade cung cấp một giao diện thống nhất cho một tập các giao diện trong một hệ thống con và như vậy sẽ làm cho các hệ thống con được sử dụng dễ dàng hơn. So sánh hai thể hiện có và không có façade (Hình 9.7 và 9.8).



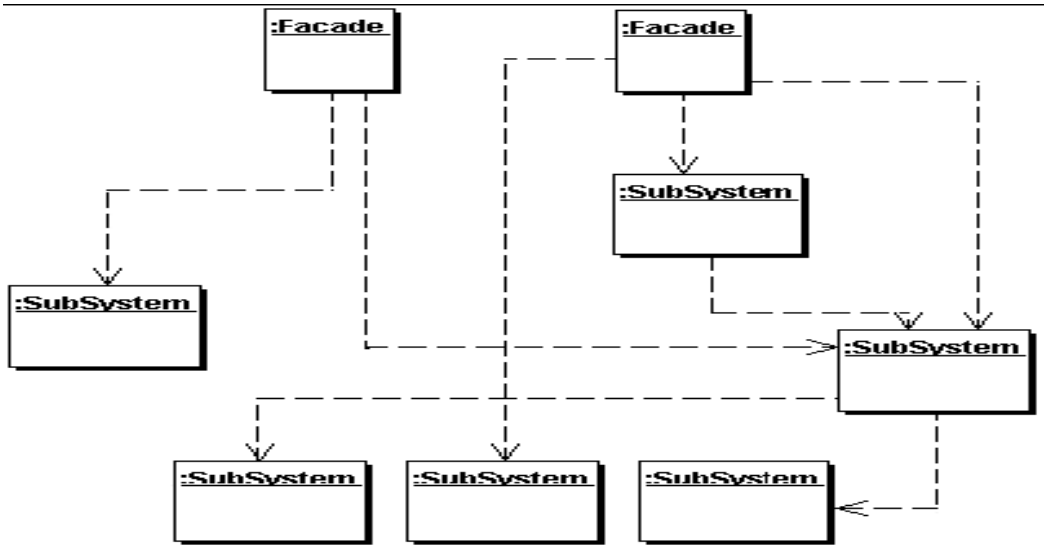
Hình 9.7: Tương tác với hệ thống con không có façade



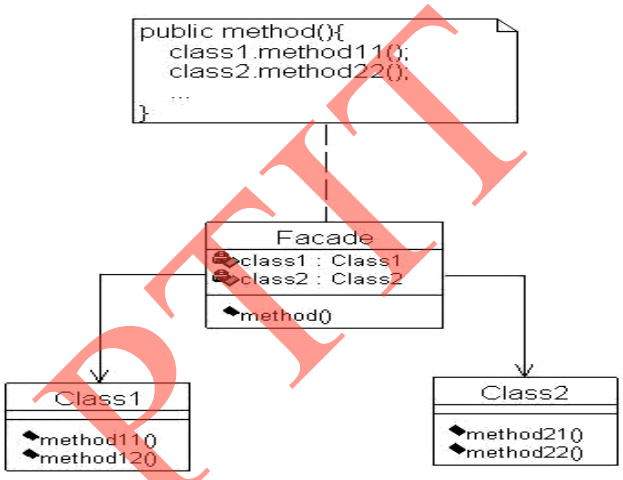
Hình 9.8: Tương tác với hệ thống con qua façade

9.5.2 Cấu trúc mẫu

Hình 9.8 là hệ với các hệ thống con và façade. Một ví dụ cụ thể hơn thể hiện ở Hình 9.9.



Hình 9.9: Kiến trúc mẫu Façade



Hình 9.10: Ví dụ cụ thể với Façade

Trong đó

- Class1 và Class2 là các lớp đã có trong hệ thống
- Façade là lớp sử dụng các phương thức của Class1 và Class2 để tạo ra một giao diện mới nên thường ít phức tạp hơn khi sử dụng riêng Class1 và Class2.

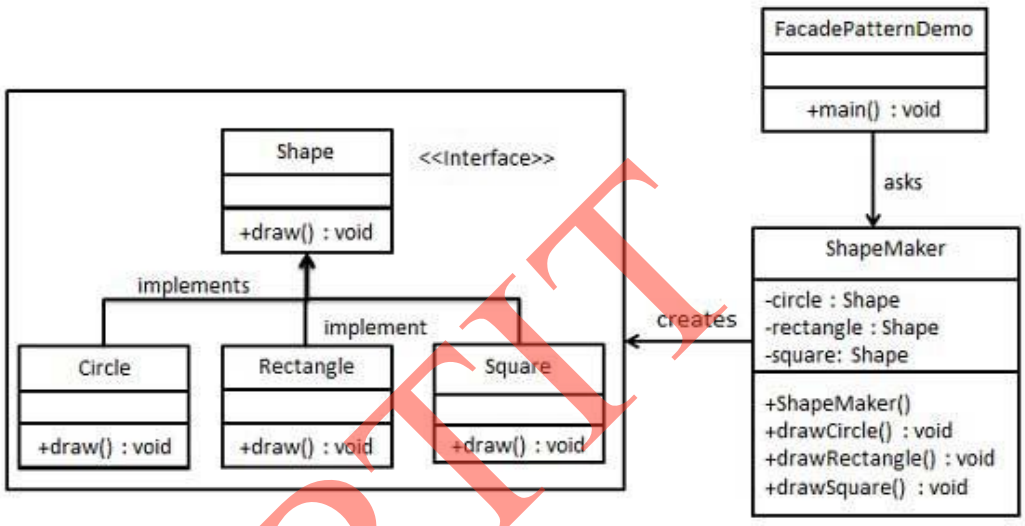
9.5.3 Tình huống áp dụng

- Façade làm cho một hệ thống phức tạp dễ sử dụng hơn bằng cách cung cấp một giao tiếp đơn giản mà không cần loại bỏ các lựa chọn phức tạp.
- Façade có thể được sử dụng cùng với mẫu Abstract Factory để cung cấp một giao diện hoạt động cho các hệ thống con độc lập. Abstract Factory cũng có thể được sử dụng như một sự thay thế cho Façade để ẩn các lớp nền đặc biệt.

- Tương tự như Mediator ở chỗ trừu tượng hóa chức năng của một lớp đã tồn tại. Façade chỉ đơn thuần là trừu tượng giao diện cho một đối tượng hệ thống con để làm nó dễ sử dụng hơn nhưng không định nghĩa một chức năng mới và lớp hệ thống con không hề biết gì về nó. Tuy nhiên, Mediator thường trừu tượng hóa chức năng trung tâm không thuộc về bất kỳ đối tượng cộng tác nào.

9.5.4 Ví dụ

Giả sử một hệ thống cũ đã có các thông tin về hình dạng *Circle, Square, Rectangle* (Hình 9.11). Ta xây dựng *interface Shape* và các lớp cụ thể cài đặt *Shape*. Một façade *ShapeMaker* được tạo ra để gửi lời gọi nhận được từ client đến các lớp cụ thể này. Lớp *FacadePatternDemo* là lớp client như vậy sẽ sử dụng lớp *ShapeMaker* này [23].



Hình 11: Sử dụng façade


```
public interface Shape {
   void draw();
}
public class Rectangle implements Shape {
   @Override
   public void draw() {
      System.out.println("Rectangle::draw()");
   }
}
public class Circle implements Shape {

   @Override
   public void draw() {
```

```
      System.out.println("Circle::draw()");
   }
}
public class ShapeMaker {
   private Shape circle;
   private Shape rectangle;
   private Shape square;

   public ShapeMaker() {
      circle = new Circle();
      rectangle = new Rectangle();
      square = new Square();
   }

   public void drawCircle(){
      circle.draw();
   }
   public void drawRectangle(){
      rectangle.draw();
   }
   public void drawSquare(){
      square.draw();
   }
}
public class FacadePatternDemo {
   public static void main(String[] args) {
      ShapeMaker shapeMaker = new ShapeMaker();

      shapeMaker.drawCircle();
      shapeMaker.drawRectangle();
      shapeMaker.drawSquare();
   }
}
```

9.6 MẪU FLYWEIGHT

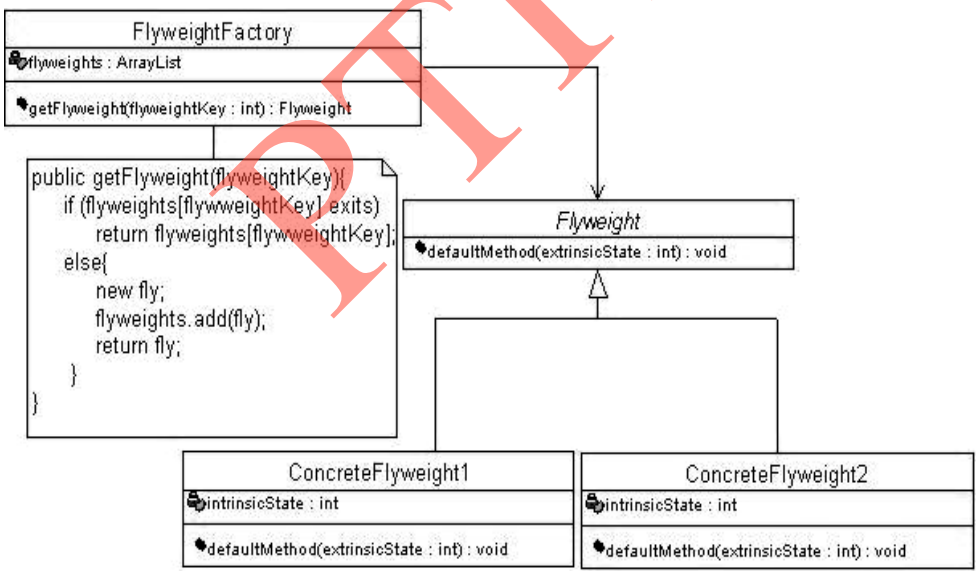
9.6.1 Đặt vấn đề

Mẫu flyweight được sử dụng để thiết kế cách tạo đối tượng hiệu quả hơn. Một số ứng dụng có thể sử dụng các đối tượng xuyên suốt pha thiết kế, nhưng như vậy việc cài đặt không tốt sẽ gây nhiều khó khăn. Trong tình huống như vậy có thể dùng mẫu thiết kế Flyweight để giải quyết hiệu quả việc phối hợp hoạt động giữa một lượng lớn các đối tượng.

Khi sử dụng mẫu này cần chú ý rằng các hiệu ứng của nó đòi hỏi phải biết rõ nó được sử dụng ở đâu và sử dụng như thế nào. Nên sử dụng mẫu này khi tất cả các điều sau đây thỏa mãn:

- Ứng dụng sử dụng một số lượng lớn đối tượng.
- Chi phí lưu trữ bởi số lượng các đối tượng là lớn.
- Hầu hết trạng thái của các đối tượng có thể chịu tác động từ bên ngoài.
- Ứng dụng không yêu cầu đối tượng đồng nhất vì các đối tượng flyweight có thể bị phân rã. Việc kiểm tra tính đồng nhất sẽ trả về đúng cho các đối tượng được định nghĩa dựa trên các khái niệm khác nhau.

9.6.2 Cấu trúc mẫu



Hình 9.12: Mẫu Flyweight

Trong đó:

- *FlyweightFactory*: tạo ra và quản lý các đối tượng *Flyweight*
- *Flyweight*: là một giao tiếp *interface* định nghĩa các phương thức chuẩn
- *ConcreteFlyweightX*: là các lớp thực thi của *Flyweight*, các thể hiện của các lớp này sẽ được sử dụng tuỳ thuộc vào điều kiện bên ngoài.

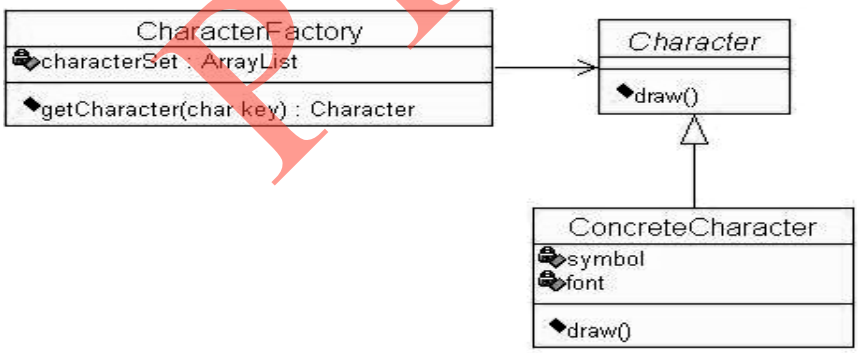
9.6.3 Tình huống áp dụng

- Ứng dụng sử dụng nhiều đối tượng giống hoặc gần giống nhau. Với các đối tượng gần giống nhau, những phần không giống nhau có thể tách rời với các phần giống nhau để cho phép các phần giống nhau có thể chia sẻ.
- Nhóm các đối tượng gần giống nhau có thể được thay thế bởi một đối tượng chia sẻ mà các phần không giống nhau đã được loại bỏ.
- Nếu ứng dụng cần phân biệt các đối tương gần giống nhau trong trạng thái gốc của chúng.
- Mẫu Flyweight thường kết hợp với mẫu Composite và thường tốt nhất là cài đặt các mẫu State và Strategy giống như là flyweight.

9.6.4 Ví dụ

Một ví dụ truyền thống của mẫu flyweight là các ký tự được lưu trong một bộ xử lí văn bản. Mỗi kí tự đại diện cho một đối tượng có dữ liệu là loại font, kích thước font, và các dữ liệu định dạng khác. Với một tài liệu lớn thì bộ xử lí văn bản sẽ gặp khó khăn khi xử lý với cấu trúc dữ liệu như thế này. Vì hầu hết dữ liệu dạng này là lặp lại nên phải có một cách để giảm việc lưu trữ này.

Trong Flyweight mỗi đối tượng ký tự sẽ chứa một tham chiếu đến một đối tượng định dạng riêng rẽ mà chính đối tượng này sẽ chứa các thuộc tính cần thiết. Điều này sẽ giảm một lượng lớn lưu trữ bằng cách kết hợp mọi kí tự có định dạng giống nhau trở thành các đối tượng đơn chỉ chứa tham chiếu đến cùng một đối tượng chứa định dạng chung đó (Hình 9.13).



Hình 9.13: Biểu diễn flyweight của kí tự

Giao tiếp *interface Character* định nghĩa phương thức tạo một kí tự và Lớp kí tự *ConcreteCharacter* cài đặt giao tiếp *Character*. Sở dĩ ta định nghĩa *Character* và *ConcreteCharacter* riêng là vì trong cấu trúc sẽ có một phần nữa là phần không giống nhau giữa các đối tượng *Character* mà ở đây ta không bàn tới.

```
public interface Character {
public void draw();
}
```

```
public class ConcreteCharacter implements Character{
	private String symbol;
	private String font;
	public ConcreteCharacter(String s, String f){
		this.symbol = s;
		this.font = f;
	}
	public void draw() {
		System.out.println("Symbol " + this.symbol + " with
font " + this.font );
	}
}

import java.util.*;
public class CharacterFactory {
	private Hashtable pool = new Hashtable<String, Character>();

	public int getNum() {
		return pool.size();
	}

	public Character get(String symbol, String fontFace) {
		Character c;
		String key = symbol + fontFace;
		if ((c = (Character)pool.get(key)) != null) {
			return c;
		} else {
			c = new ConcreteCharacter(symbol, fontFace);
			pool.put(key, c);
			return c;
		}
	}
}
//Lớp Test

import java.util.*;
public class Test {
	public static void main(String[] agrs) {
		CharacterFactory characterfactory = new
CharacterFactory();
		ArrayList<Character> text = new
ArrayList<Character>();
		text.add(0, characterfactory.get("a", "arial"));
		text.add(1, characterfactory.get("b", "time"));
		text.add(2, characterfactory.get("a", "arial"));
		text.add(0, characterfactory.get("c", "arial"));
		for (int i = 0; i < text.size(); i++){
```

```
                    Character c = (Character)text.get(i);
                          c.draw();
                }
        }
}
```

Lớp khởi tạo các ký tự theo symbol và font, mỗi lần khởi tạo nó sẽ lưu các đối tượng này vào vùng nhớ riêng của nó, nếu đối tượng ký tự symbol + font đã có thì nó sẽ lấy ra chứ không khởi tạo lại. Như vậy 'a' + 'arial' gọi hai lần nhưng chỉ khởi tạo có một lần mà thôi.

9.7 MẪU PROXY

9.7.1 Đặt vấn đề

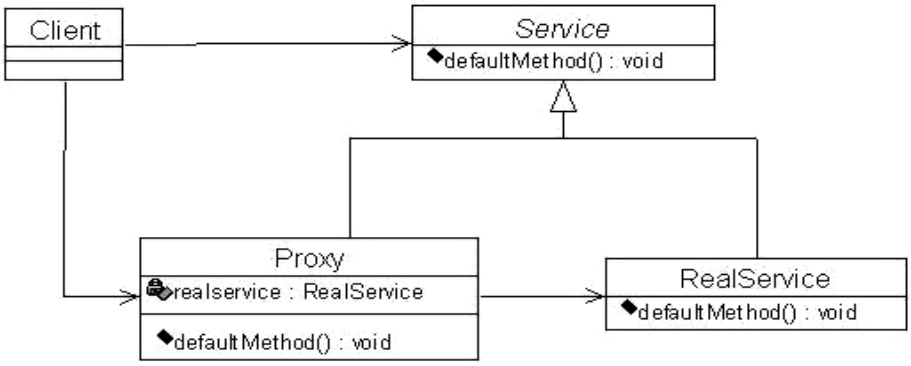
Khi cần điều khiển truy nhập tới một đối tượng được thực hiện từ quá trình khởi tạo nó cho tới khi thực sự cần sử dụng nó. Trong trường hợp như vậy, ta nên dùng mẫu thiết kế proxy. Mẫu này có thể áp dụng vào những tình huống cần phải tham chiếu tới một đối tượng linh hoạt hơn, tinh tế hơn so với việc sử dụng con trỏ đơn giản. Proxy cung cấp một đại diện cho một đối tượng để điều khiển việc truy nhập nó. Có thể sử dụng proxy để đếm số tham chiếu tới đối tượng thực, do đó nó có thể tự động giải phóng khi không tham chiếu hay tải một đối tượng vào bộ nhớ khi nó được tham chiếu lần đầu tiên. Hoặc cần kiểm tra đối tượng thực nào đó có được khóa hay không trước khi nó bị truy nhập để đảm bảo không đối tượng nào khác có thể thay đổi nó. Sau đây là một số kiểu proxy thường được sử dụng:

- Một *remote proxy* từ xa cung cấp một biểu diễn cục bộ cho một đối tượng trong một không gian địa chỉ khác.
- Một *virtual proxy* ảo tạo ra một đối tượng có chi phí cao theo yêu cầu.
- Một *protection proxy* bảo vệ điều khiển việc truy nhập đối tượng gốc. Các protection proxy rất hữu ích khi các đối tượng có các quyền truy nhập khác nhau.
- Một *smart reference proxy* là sự thay thế cho một con trỏ rỗng cho phép thực hiện các chức năng thêm vào khi một đối tượng được truy nhập.

9.7.2 Cấu trúc mẫu

Hình 9.14 là cấu trúc mẫu proxy. Trong đó:

- *Service:* là giao tiếp định nghĩa các phương thức chuẩn cho một dịch vụ nào đó
- *RealService:* là một cài đặt của giao tiếp Service, lớp này sẽ khai báo tường minh các phương thức của Service, lớp này xem như thực hiện tốt tất cả các yêu cầu từ Service
- *Proxy:* kế thừa Service và sử dụng đối tượng của RealService



Hình 9.14: Mẫu Proxy

9.7.3 Tình huống áp dụng

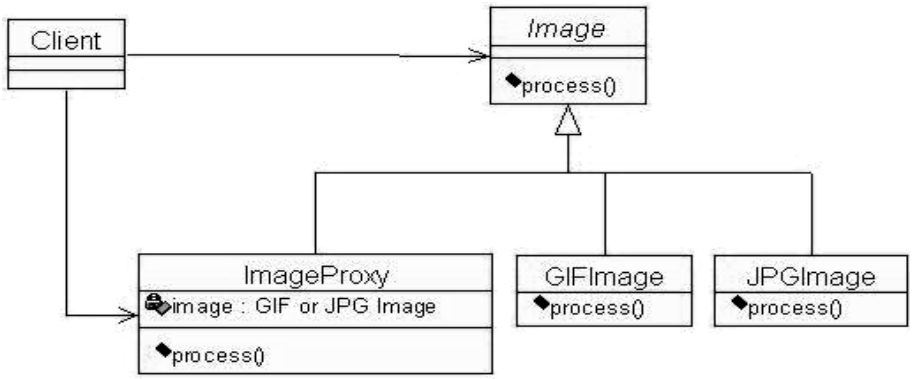
- Sử dụng mẫu Proxy khi cần một tham chiếu đến một đối tượng một cách phức tạp thay vì theo cách bình thường.
- Proxy truy nhập đối tượng từ xa (Remote proxy) – có thể sử dụng khi bạn cần một tham chiếu định vị cho một đối tượng trong không gian địa chỉ (JVM)
- Proxy ảo (Virtual proxy) – lưu giữ các thông tin thêm vào về một dịch vụ thực vì vậy chúng có thể hoãn lại sự truy xuất vào dịch vụ này
- Proxy bảo vệ (Protection proxy) – xác thực quyền truy xuất vào một đối tượng thực

Nhận xét: Thông thường mẫu Adapter cung cấp một giao diện khác với đối tượng mà nó thích nghi. Trong trường hợp này, Proxy cung cấp cùng một giao diện giống như chủ thể. Mặc dù decorator có thể có cài đặt tương tự như các proxy, nhưng decorator có một mục đích khác. Một decorator bổ sung thêm nhiều nhiệm vụ cho một đối tượng nhưng ngược lại proxy điều khiển truy cập đến một đối tượng. Proxy tuỳ biến theo nhiều cấp khác nhau mà chúng có thể sẽ được cài đặt giống như một decorator. Một protection proxy có thể được cài đặt chính xác như một decorator. Mặt khác một proxy truy cập đối tượng từ xa sẽ không chứa một tham chiếu trực tiếp đến chủ thể thực sự nhưng chỉ duy nhất có một tham chiếu trực tiếp, giống như ID của host và địa chỉ trên host vậy. Một proxy ảo sẽ bắt đầu với một tham chiếu gián tiếp chẳng hạn như tên file nhưng rốt cuộc rồi cũng sẽ đạt được một tham chiếu trực tiếp.

9.7.4 Ví dụ

Ví dụ lớp *Image* là một *interface* định nghĩa các phương thức xử lý ảnh và có các lớp con là *GIFImage* và *JPGImage*. Theo hướng đối tượng thì thiết kế như thế có vẻ hợp lý, Client chỉ cần sử dụng lớp *Image* là đủ, còn tuỳ thuộc vào loại ảnh sẽ có các phương thức khác nhau. Nhưng trong trường hợp ứng dụng web chẳng hạn, một lúc có thể đọc đến hàng trăm ảnh các loại và còn muốn xử lý tuỳ vào một điều kiện nào đó (ví dụ chỉ xử lý khi là ảnh JPG hoặc GIF). Nếu đặt điều kiện *IF Image* (sau đó sẽ tùy điều kiện này rồi xử lý riêng) thì không hợp lý; nếu đặt trong Client mỗi lần cần thay đổi IF lại sửa Client thì cũng không hợp lý khi Client là một ứng dụng lớn.

Khi sử dụng Proxy, lớp *ImageProxy* chỉ là lớp đại diện cho *Image*, kế thừa từ *Image* và sử dụng các lớp *GIFImage* hay *JPGImage*. Khi cần thay đổi ta chỉ cần thay đổi trên Proxy mà không cần tác động đến Client và Image (Hình 9.15).



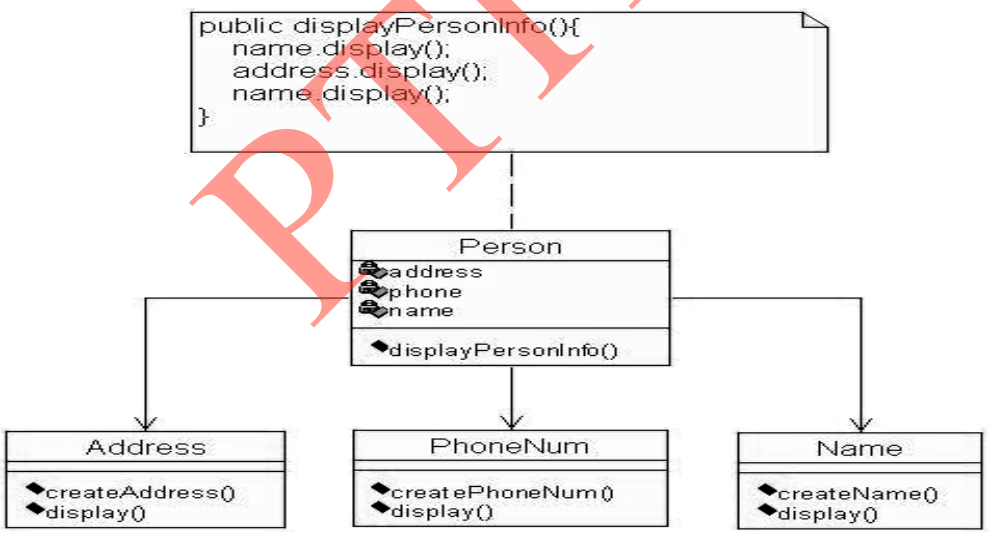
Hình 9.15: Proxy cho xử lý ảnh

```
public interface Image {
	public void process();
}
public class JPGImage implements Image {
	public void process() {
		System.out.print("JPG Image");
	}
}
public class GIFImage implements Image {
	public void process() {
		System.out.print("GIF Image");
	}
}
public class ImageProxy implements Image {
	private Image image;
	public void process() {
	if (image == null)
		image = new JPGImage();//tạo đối tượng ảnh JPG, chỉ
mang tính minh họa
	image.process();
	}
}
```

9.8 KẾT LUẬN

Chương này đã trình bày một số mẫu thiết kế kiến trúc. Mỗi mẫu đều trình bày lý do cần có của mẫu đó, cấu trúc của mẫu, các tình huống áp dụng và ví dụ áp dụng cụ thể. Đồng thời, cũng nêu lên các tình huống sử dụng cũng như khả năng kết hợp với các mẫu thiết kế khác. Việc tìm hiểu các mẫu này có ý nghĩa quan trọng trong việc nâng cao chất lượng thiết kế phần mềm.

BÀI TẬP

1. Hoàn thiện mẫu adapter bằng cách thêm mã client với main() để thực thi chương trình
2. Hoàn thiện mẫu bridge bằng cách thêm mã client với main() để thực thi chương trình
3. Hoàn thiện mẫu composite bằng cách thêm mã client với main() để thực thi chương trình
4. Hoàn thiện mẫu decorator bằng cách thêm mã client với main() để thực thi chương trình
5. Giả sử có các lớp liên quan đến quản lý thông tin cá nhân gồm địa chỉ *Address*, số điện thoại *PhoneNum*, tên *Name*. Để quản lý các thông tin trên dựa vào tận dụng lại hệ thống cũ bằng cách xây một lớp người *Person* sử dụng lại các lớp ở trên (Hình 9.15). Sử dụng mẫu façade để xây dựng mã trình và thêm mã client với main() để thực thi chương trình
6. 

Hình 9.15: Quản lý thông tin cá nhân

7. Hoàn thiện mẫu flyweight bằng cách thêm mã client với main() để thực thi chương trình
8. Hãy thiết kế và cài đặt façade để có thể sử dụng bởi những client khác nhau đưa ra yêu cầu mua sắm gồm các loại hàng khác nhau, comment về mặt hàng…
9. Hoàn thiện mã trình mẫu Proxy trong ví dụ 9.7 và thêm client để thực thi chương trình

CHƯƠNG 10: CÁC MẪU THIẾT KẾ HÀNH VI

Chương này tập trung trình bày nhưng mẫu thiết kế hành vi, nội dung bao gồm các mẫu thiết kế:

- Mẫu chuỗi trách nhiệm (chain of responsibity)
- Mẫu lệnh (command)
- Mẫu lặp (iterator)
- Mẫu phiên dịch (interpreter)
- Mẫu trung gian (mediator)
- Mẫu hồi tưởng (memento)
- Mẫu quan sát (observer)
- Mẫu chiến lược (strategy)
- Mẫu trạng thái (state)
- Mẫu phương thức mẫu (template method)

Các mẫu hành vi (behavior pattern) là các mẫu tập trung vào giải quyết các vấn đề của thuật toán và sự phân chia trách nhiệm giữa các đối tượng. Mẫu hành vi không chỉ mô hình các đối tượng mà còn mô hình cách trao đổi thông tin giữa chúng. Nhờ đó, nó giúp chúng ta tập trung hơn vào việc xây dựng cách thức liên kết giữa các đối tượng thay vì các luồng điều khiển.

Mẫu hành vi sử dụng tính kế thừa để phân phối hành vi giữa các lớp. Ví dụ xem xét hai mẫu Template Method và Interpreter. Template Method là mẫu đơn giản và thông dụng hơn. Nó định nghĩa trừu tượng từng bước của một thuật toán; mỗi bước sử dụng một hàm trừu tượng hoặc một hàm nguyên thuỷ. Các lớp con của nó cài đặt thuật toán cụ thể bằng cách cụ thể hoá các hàm trừu tượng. Mẫu Interpreter biểu diễn văn phạm như một hệ thống phân cấp của các lớp và trình phiên dịch như một thủ tục tác động lên các thể hiện của các lớp đó.

Mẫu hành vi kiểu đối tượng lại sử dụng đối tượng thành phần thay vì thừa kế. Một vài mẫu miêu tả cách thức nhóm các đối tượng ngang hàng hợp tác với nhau để thi hành các tác vụ mà không một đối tượng riêng lẻ nào có thể tự thực thi được. Một vấn đề quan trọng được đặt ra ở đây là bằng cách nào các đối tượng ngang hàng đó biết được sự tồn tại của nhau. Cách đơn giản nhất là lưu trữ các tham chiếu trực tiếp đến nhau trong các đối tượng ngang hàng nhưng như thế lại dư thừa. Mẫu Mediator tránh sự thừa thãi này bằng cách xây dựng một kết nối trung gian, liên kết gián tiếp các đối tượng ngang hàng.

Mẫu chuỗi trách nhiệm Chain of Responsibility xây dựng mô hình liên kết thậm chí còn "lỏng" hơn. Nó cho phép gửi yêu cầu đến một đối tượng thông qua chuỗi các đối tượng "ứng viên". Mỗi ứng viên có khả năng thực hiện yêu cầu tuỳ thuộc vào các điều kiện

trong thời gian chạy. Số lượng ứng viên là một con số mở và ta có thể lựa chọn những ứng viên nào sẽ tham gia vào chuỗi trong thời gian chạy.

Mẫu Observer xây dựng và vận hành một sự phụ thuộc giữa các đối tượng. Một ví dụ cụ thể của mẫu này là mô hình MVC (Model/View/Controller) của Smalltalk trong đó tất cả các views của model đều được thông báo khi trạng thái của model có sự thay đổi.

Một số mẫu hành vi khác lại quan tâm đến việc đóng gói các hành vi vào một đối tượng và "uỷ thác" các yêu cầu cho nó. Mẫu Strategy đóng gói một thuật toán trong một đối tượng bằng cách xây dựng một cơ chế khiến cho việc xác định và thay đổi thuật toán mà đối tượng sử dụng trở nên đơn giản. Trong khi đó mẫu Command lại đóng gói một yêu cầu vào một đối tượng để nó có thể được truyền như một tham số, được lưu trữ trong một danh sách hoặc thao tác theo những cách thức khác. Mẫu State đóng gói các trạng thái của một đối tượng, làm cho đối tượng có khả năng thay đổi hành vi của mình khi trạng thái thay đổi. Mẫu Visitor đóng gói các hành vi vốn được phân phối trong nhiều lớp và mẫu Iterator trừu tượng hoá cách thức truy cập và duyệt các đối tượng trong một tập hợp.

10.1 MẪU CHUỖI TRÁCH NHIỆM

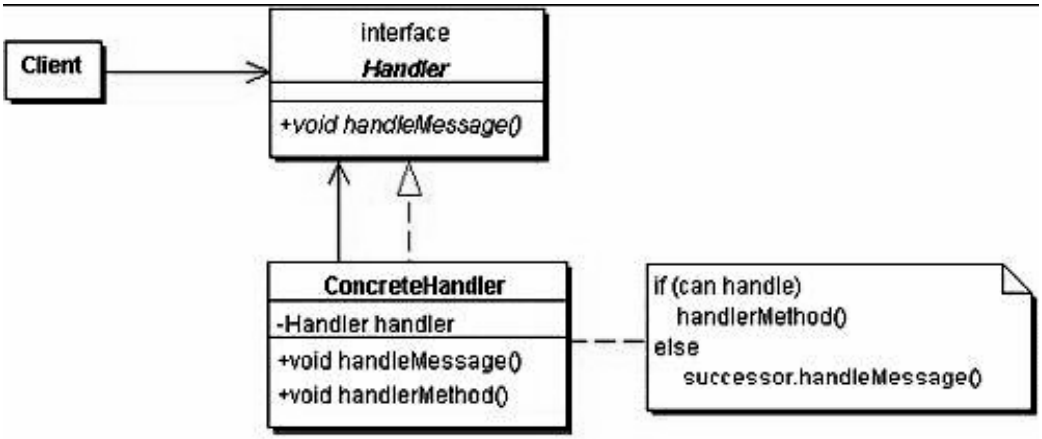
10.1.1 Đặt vấn đề

Mẫu chuỗi trách nhiệm (chain of responsibility) thiết lập một chuỗi đối tượng bên trong một hệ thống, trong đó các thông điệp hoặc có thể được thực hiện ở tại một mức nơi mà nó được nhận lần đầu hoặc được chuyển đến một đối tượng để thực thi.

10.1.2 Cấu trúc mẫu

Cấu trúc mẫu này được cho trong Hình 10.1. Trong đó:

- *Handler*: là một giao tiếp định nghĩa phương thức sử dụng để chuyển thông điệp qua các lần thực hiện tiếp theo.
- *ConcreteHandler*: là một cài đặt của giao tiếp *Handler*. Nó giữ một tham chiếu đến một *Handler* tiếp theo. Việc thực thi phương thức *handlerMessage* có thể xác định làm thế nào để thực hiện phương thức và gọi một *handlerMethod*, chuyển tiếp thông điệp đến cho *Handler* tiếp theo hoặc kết hợp cả hai.



Hình 10.1: Chuỗi trách nhiệm

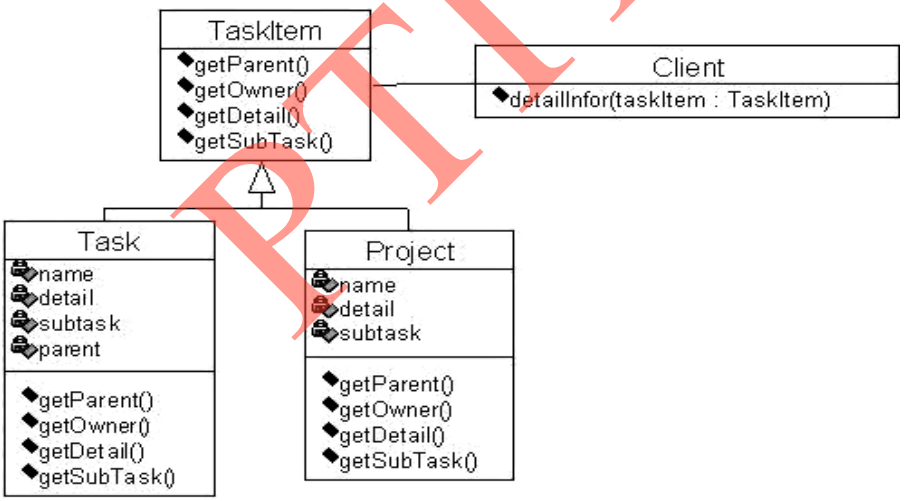
10.1.3 Tình huống áp dụng

- Có một nhóm các đối tượng trong hệ thống có thể đáp ứng tất cả các loại thông điệp giống nhau.
- Các thông điệp phải được thực hiện bởi một vài đối tượng trong hệ thống.
- Các thông điệp được thực thi theo mô hình "thực hiện – chuyển tiếp", một vài sự kiện có thể được thực hiện ở mức mà chúng được nhận hoặc tạo ra, trong khi một số sự kiện khác phải được chuyển tiếp đến một vài đối tượng khác.

10.1.4 Ví dụ

Một hệ thống quản lý thông tin cá nhân có thể được sử dụng để quản lý các dự án. Hãy hình dung biểu diễn dự án như một cấu trúc cây gồm đỉnh là một dự án, các đỉnh con là các tác vụ của dự án đó và mỗi đỉnh con tác vụ lại phân rã thành một tập các đỉnh con tác vụ khác. Để quản lý cấu trúc này ta thực hiện như sau:

- Ở mỗi đỉnh ta lưu các thông tin: tên tác vụ, đỉnh cha, tập các đỉnh con
- Ta xét một thông điệp duyệt từ đỉnh gốc để in ra các thông tin
- Như vậy với thông điệp này, việc in thông tin ở một đỉnh là chưa đủ, mà phải chuyển tiếp đến in thông tin các đỉnh con tiếp theo và chuyển tiếp cho đến khi không còn đỉnh con thì mới dừng.



Hình 10.2: Quản lý thông tin cá nhân

Giao tiếp *TaskItem* định nghĩa các phương thức cho *Project* cơ sở và các tác vụ. Lớp *Project* cài đặt giao tiếp *TaskItem*, nó là lớp đại diện cho các đỉnh gốc trên cùng của cây

```
public interface TaskItem {
    public TaskItem getParent();
    public String getDetails();
    public ArrayList getProjectItems();
}
public class Project implements TaskItem {
```

```
    private String name;
    private String details;
    private ArrayList subtask = new ArrayList();

    public Project(){ }
    public Project(String newName, String newDetails){
        name = newName;
        details = newDetails;
    }
        public String getName(){
        return name;
    }
    public String getDetails(){
        return details;
    }
    public ProjectItem getParent(){
        return null;
//vì project là ở mức cơ sở, đỉnh gốc trên cùng nên không có cha
    }
    public ArrayList getSubTask(){
        return subtask;
    }

    public void setName(String newName){
        name = newName;
    }
    public void setDetails(String newDetails){
        details = newDetails;
    }

    public void addTask(TaskItem element){
        if (!subtask.contains(element)){
            subtask.add(element);
        }
    }

    public void removeProjectItem(TaskItem element){
        subtask.remove(element);
    }
}
//Lớp Task thực thi giao tiếp TaskItem đại diện cho các tác vụ,
//các đỉnh không phải ở gốc của cây
public class Task implements TaskItem {
    private String name;
    private ArrayList subtask = new ArrayList();
    private String details;
    private TaskItem parent;
```

```
    public Task(TaskItem newParent){
        this(newParent, "", "");
    }
    public Task(TaskItem newParent, String newName, String
newDetails,){
            parent = newParent;
            name = newName;
            details = newDetails;
    }

    public String getDetails(){
        if (primaryTask){
            return details;
        }
        else{
            return parent.getDetails() + EOL_STRING + "\t" +
details;
        }
    }

    public String getName(){
        return name;
    }
    public ArrayList getSubTask(){ return subtask; }
    public ProjectItem getParent(){
        return parent;
    }

    public void setName(String newName){
        name = newName;
    }
    public void setParent(TaskItem newParent){
        parent = newParent;
    }
    public void setDetails(String newDetails){
        details = newDetails;
    }

    public void addSubTask(TaskItem element){
        if (!subtask.contains(element)){
            subtask.add(element);
        }
    }

    public void removeSubTask(TaskItem element){
        subtask.remove(element);
```

```
    }
}
//Lớp thực thi test mẫu
public class RunPattern {
    public static void main(String [] arguments){
        Project project = new Project("Project 01", "Detail of
Project 01");
        //Khởi tạo, thiết lập các tác vụ con …
        detailInfor(project);
    }
      private static void detailInfor (TaskItem item){
        System.out.println("TaskItem: " + item);
        System.out.println("  Details: " + item.getDetails());
        System.out.println();
        if (item.getSubTask() != null){
            Iterator subElements = item.getSubTask().iterator();
            while (subElements.hasNext()){
                detailInfor ((TaskItem)subElements.next());
            }
        }
    }
}
```

10.2 MẪU COMMAND

10.2.1 Đặt vấn đề

Đôi khi chúng ta gặp tình huống cần phải gửi yêu cầu đến một đối tượng mà không biết cụ thể về đối tượng nhận yêu cầu cũng như phương thức xử lý yêu cầu. Ví dụ về bộ công cụ giao diện người sử dụng cho phép xây dựng các menu. Rõ ràng, nó không thể xử lý trực tiếp yêu cầu trên các đối tượng menu vì cách thức xử lý phụ thuộc vào từng ứng dụng cụ thể.

Mẫu Command cho phép bộ công cụ trên gửi yêu cầu đến các đối tượng chưa xác định bằng cách biến chính yêu cầu thành một đối tượng. Khái niệm quan trọng nhất của mẫu này là lớp trừu tượng *Command* có chức năng định nghĩa giao diện cho việc thực thi các yêu cầu. Trong trường hợp đơn giản nhất, ta chỉ cần định nghĩa một thủ tục ảo *Execute* trên giao diện đó. Các lớp con cụ thể của *Command* sẽ xác định cặp đối tượng nhận yêu cầu - thao tác bằng cách lưu trữ một tham chiếu đến đối tượng nhận yêu cầu và định nghĩa lại thủ tục *Execute* để gọi các thủ tục xử lý.

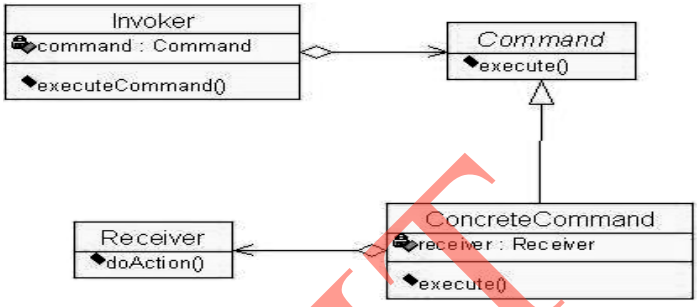
Với cách làm như vậy, chương trình sẽ có nhiệm vụ tạo ra các *menu, menuItem* và gán mỗi *menuItem* một đối tượng thuộc lớp con cụ thể của *Command*. Khi người sử dụng chọn một *menuItem*, nó sẽ gọi hàm *command.execute()* mà không cần *command* tham chiếu đến loại lớp con cụ thể nào của lớp *Command*. Hàm *execute()* sẽ có nhiệm vụ xử lý yêu cầu. Mẫu *Command* đóng gói yêu cầu như là một đối tượng, làm cho nó có thể được truyền như

một tham số, được lưu trữ trong một danh sách hoặc thao tác theo những cách thức khác nhau.

10.2.2 Cấu trúc mẫu

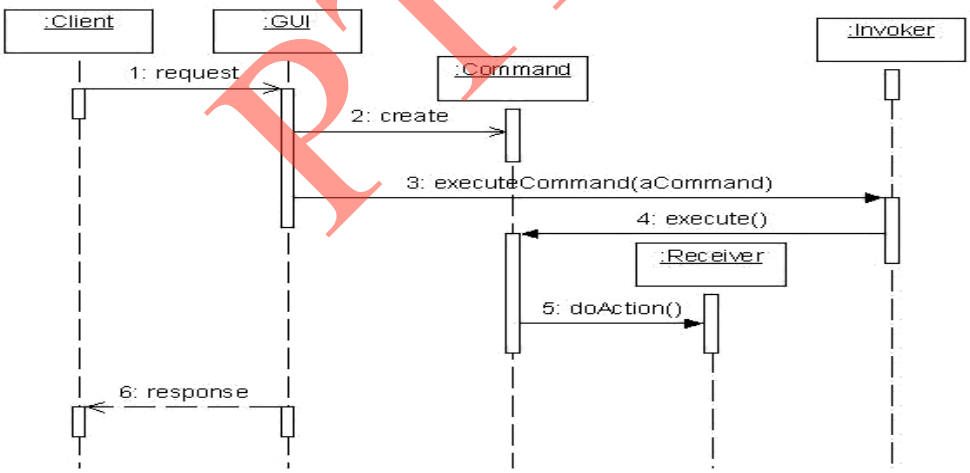
Cấu trúc mẫu được cho trong Hình 10.3, trong đó:

- *Command*: là một giao tiếp định nghĩa các phương thức cho *Invoker* sử dụng
- *Invoker*: lớp này thực hiện các phương thức của đối tượng *Command*
- *Receiver*: là đích đến của *Command* và là đối tượng thực hiện hoàn tất yêu cầu, nó có tất cả các thông tin cần thiết để thực hiện điều này
- *ConcreteCommand*: là một cài đặt của giao tiếp Command. Nó lưu giữa một tham chiếu *Receiver* mong muốn.



Hình 10.3: Mẫu Command

Luồng thực thi tuần tự của mẫu Command thể hiện trong Hình 10.4:



Hình 10.4: Biểu đồ tuần tự của command

Client gửi yêu cầu đến giao diện GUI của ứng dụng. Ứng dụng khởi tạo một đối tượng *Command* thích hợp cho yêu cầu đó (đối tượng này sẽ là các *ConcreteCommand*). Sau đó ứng dụng gọi phương thức *executeCommand()* với tham số là đối tượng Command vừa khởi tạo. *Invoker* khi được gọi thông qua phương thức *executeCommand()* sẽ thực hiện gọi phương thức *execute()* của đối tượng Command tham số. Đối tượng Command này sẽ gọi tiếp phương thức *doAction()* của thành phần Receiver được khởi tạo từ đầu, *doAction()* chính là phương thức chính để hoàn tất yêu cầu của Client.

10.2.3 Tình huống áp dụng

Mẫu Command được áp dụng khi:

- Tham số hoá các đối tượng theo thủ tục mà chúng thi hành.
- Xác định, xếp hàng và thực thi các yêu cầu tại những thời điểm khác nhau.
- Cho phép quay lại. Thủ tục execute của lớp Command có thể lưu lại trạng thái để cho phép quay lại các biến đổi mà nó gây ra. Khi đó lớp Command trừu tượng cần định nghĩa thêm hàm Unexecute để đảo ngược các biến đổi. Các lệnh đã được thực thi sẽ được lưu trong một danh sách, từ đó cho phép undo và redo không giới hạn mức.
- Cần hỗ trợ ghi lại các lệnh đã được thực thi để thi hành lại trong trường hợp hệ thống gặp sự cố.
- Thiết kế một hệ thống với các thủ tục được xây dựng dựa trên các thủ tục nguyên thuỷ. Cấu trúc này thường gặp trong các hệ thống thông tin hỗ trợ "phiên giao dịch". Một phiên giao dịch là một tập hợp các thay đổi lên dữ liệu. Mẫu Command cung cấp cách thức mô tả phiên giao dịch. Nó có giao diện chung cho phép khởi xướng các phiên làm việc với cùng một cách thức và cũng cho phép dễ dàng mở rộng hệ thống với các phiên giao dịch mới.
- Một Composite có thể được sử dụng để cài đặt các lệnh Commands. Một Memmento có thể lưu lại các trạng thái để Command yêu cầu phục hồi lại các hiệu ứng của nó. Một command phải được sao lưu trước khi nó được thay thế bằng các hành động trước đó như là một Prototype.

10.2.4 Ví dụ

Mã nguồn tiêu biểu cài đặt ứng dụng trên được cho dưới đây.

```
public interface Command{
	public void execute();
}
public class ConcreteCommand implements Command{
	private Receiver receiver;
	public void setReceiver(Receiver receiver){
		this.receiver = receiver;
	}
	public Receiver getReceiver(){
		return this.receiver;
	}

	public void execute (){
		receiver.doAction();
	}
}
```

```
public class Receiver{
	private String name;
	public Receiver(String name){
		this.name = namel
	}
	public void doAction(){
		System.out.print(this.name + " fulfill request!");
	}
}

public class Invoker{
	public void executeCommand(Command command){
		command.execute();
	}
}

public class Run{
	public static void main(String[] agrs){
		Command command = new ConcreteCommand();
		command.setReceiver(new Receiver("NguyenD"));
		Invoker invoker = new Invoker();
		Invoker.executeCommand(command);
	}
}
```

10.3 MẪU ITERATOR

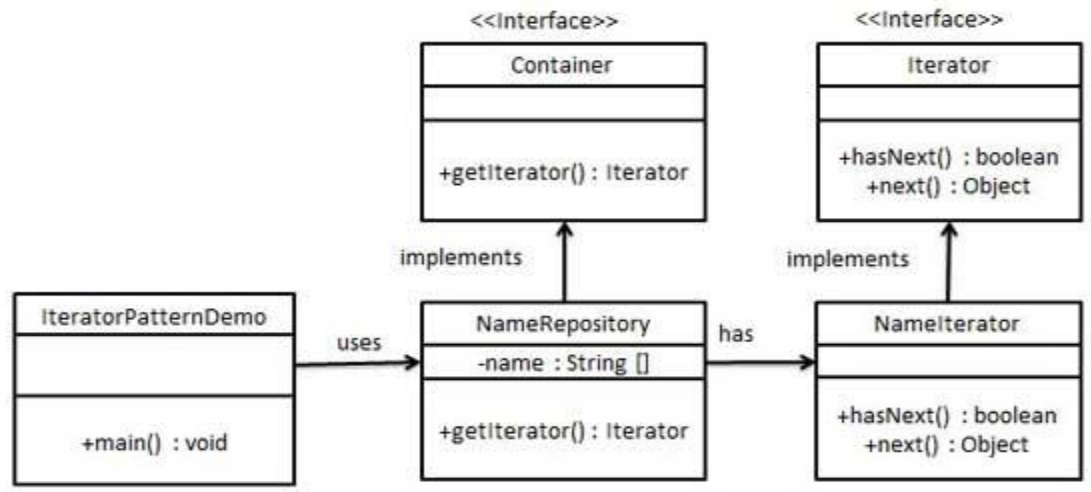
10.3.1 Đặt vấn đề

Một đối tượng tập hợp xem như là một danh sách cung cấp các phương thức truy cập các thành phần của nó. Tuy nhiên, đôi lúc chúng ta cần duyệt các thành phần của danh sách theo những cách thức và tiêu chí khác nhau nhưng không mở rộng giao diện của danh sách *List* với các phương thức cho các cách thức duyệt.

Mẫu Iterator cho phép chúng ta duyệt danh sách một cách dễ dàng bằng cách tách chức năng truy cập và duyệt ra khỏi danh sách và đặt vào đối tượng iterator. Lớp *Iterator* sẽ định nghĩa một giao diện để truy cập các thành phần của danh sách, đồng thời quản lý cách thức duyệt danh sách hiện thời. Như vậy, mẫu *Iterator* cung cấp khả năng truy cập và duyệt các thành phần của một tập hợp mà không cần quan tâm đến cách thức biểu diễn bên trong. Iterator thường được sử dụng trong Java và.Net.

10.3.2 Cấu trúc mẫu

Cấu trúc mẫu được cho trong Hình 10.5. Iterator là interface gồm phương thức duyệt danh sách, *Container* lấy duyệt từ Interator. Lớp cài đặt *Container* sẽ chịu trách nhiệm cài đặt Iterator và sử dụng nó. *NameIteratorDemo* sẽ sử dụng *NameRepository* để in ra danh sách.



Hình 10.5: Cấu trúc mẫu Iterator

10.3.3 Tình huống áp dụng

- Truy nhập các thành phần của một tập hợp mà không cần quan tâm đến cách thức biểu diễn bên trong.
- Hỗ trợ nhiều phương án duyệt của các đối tượng tập hợp.
- Cung cấp giao diện chung cho việc duyệt các cấu trúc tập hợp.
- Iterator thường được sử dụng để duyệt một cấu trúc đệ quy như Composite.
- Đa hình một Iterator dựa trên FactoryMethod để tạo thể nghiệm cho các lớp con tương ứng của Iterator.
- Iterator thường được sử dụng cùng với mẫu Memento. Một Iterator có thể sử dụng một Memento để nắm bắt trạng thái của một Iterator và khi đó Iterator lưu trữ các memento ở bên trong.

10.3.4 Ví dụ

Ví dụ này tiếp tục cấu trúc mẫu và cho mã nguồn sau đây:

```
public interface Iterator {
   public boolean hasNext();
   public Object next();
}
public interface Container {
   public Iterator getIterator();
}
ublic class NameRepository implements Container {
   public String names[] = {"Minh" , "Ngoc" ,"Thanh" , "Lan"};
   @Override
```

```
   public Iterator getIterator() {
      return new NameIterator();
   }
private class NameIterator implements Iterator {
      int index;
      @Override
      public boolean hasNext() {
               if(index < names.length){
            return true;
         }
         return false;
      }
      @Override
      public Object next() {
             if(this.hasNext()){
            return names[index++];
         }
         return null;
      }
   }
}
public class IteratorPatternDemo {
   public static void main(String[] args) {
      NameRepository namesRepository = new NameRepository();
      for(Iterator iter = namesRepository.getIterator();
iter.hasNext();){
         String name = (String)iter.next();
         System.out.println("Name : " + name);
      }
   }
}
```


10.4 MẪU INTERPRETER

10.4.1 Đặt vấn đề

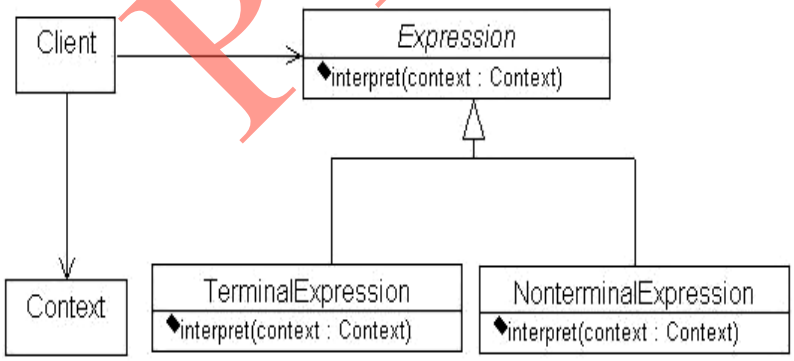
Nếu một dạng bài toán có tần suất xuất hiện tương đối lớn, người ta thường biểu diễn các thể hiện cụ thể của nó bằng các câu trong một ngôn ngữ đơn giản. Sau đó xây dựng một trình biên dịch để giải quyết bài toán bằng cách biên dịch các câu.

Ví dụ, tìm kiếm các xâu thoả mãn một mẫu cho trước là một bài toán thường gặp và thông thường là tạo một ngôn ngữ dùng để biểu diễn các mẫu của xâu. Thay vì xây dựng từng thuật toán riêng biệt tương ứng mỗi mẫu với một tập các xâu, người ta xây dựng một thuật toán tổng quát để có thể phiên dịch các biểu diễn thành tập các xâu tương ứng. Mẫu Interpreter mô tả cách thức xây dựng cấu trúc ngữ pháp cho những ngôn ngữ đơn giản, cách thức biểu diễn câu trong ngôn ngữ và cách thức phiên dịch các câu đó. Như vậy, Interpreter đưa ra một biểu diễn ngôn ngữ, xây dựng cách diễn đạt ngôn ngữ đó cùng với một trình phiên dịch sử dụng cách diễn tả trên để phiên dịch các câu.

10.4.2 Cấu trúc mẫu

Cấu trúc mẫu được cho trong Hình 10.6. Trong đó:

- *Expression:* là một giao tiếp mà thông qua nó, client tương tác với các biểu thức
- *TerminalExpression:* là một cài đặt của giao tiếp *Expression*, đại diện cho các nốt cuối trong cây cú pháp
- *NonterminalExpression:* là một cài đặt khác của giao tiếp *Expression*, đại diện cho các nút chưa kết thúc trong cấu trúc của cây cú pháp. Nó lưu trữ một tham chiếu đến *Expression* và triệu gọi phương thức diễn giải cho mỗi phần tử con.
- *Context:* chứa thông tin cần thiết cho một vài vị trí trong khi diễn giải. Nó có thể phục vụ như một kênh truyền thông cho các thể hiện của *Expression*.
- *Client:* hoặc là xây dựng hoặc là nhận một thể hiện của cây cú pháp ảo. Cây cú pháp này bao gồm các thể hiện của *TerminalExpression* và *NoterminalExpression* để tạo nên câu đặc tả. Client triệu gọi các phương thức diễn giải với ngữ cảnh thích hợp khi cần thiết.



Hình 10.6: Cấu trúc mẫu Interpreter

10.4.3 Tình huống áp dụng

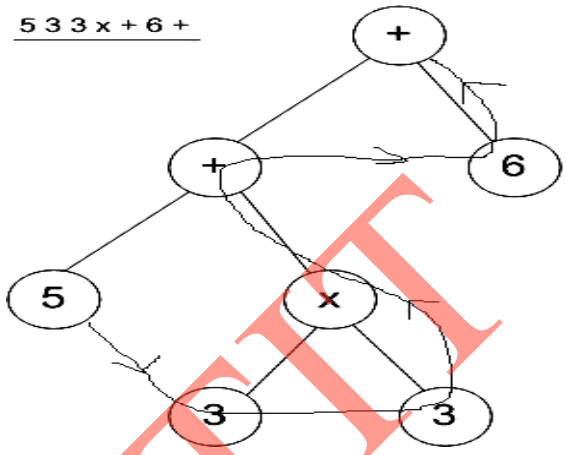
Sử dụng mẫu Interpreter khi cần phiên dịch một ngôn ngữ mà ta có thể miêu tả các câu bằng cấu trúc cây cú pháp. Mẫu này hoạt động hiệu quả nhất khi:

- Cấu trúc ngữ pháp đơn giản. Với các cấu trúc ngữ pháp phức tạp, cấu trúc lớp của ngữ pháp trở nên quá lớn và khó kiểm soát, việc tạo ra các cây cú pháp sẽ tốn thời gian và bộ nhớ.

- Hiệu quả không phải là yếu tố quan trọng nhất. Các cách thức biên dịch hiệu quả nhất thường không áp dụng trực tiếp mẫu Interpreter mà phải biến đổi các biểu diễn thành các dạng khác trước.
- Cây cú pháp trừu tượng là một thể nghiệm trong mẫu Composite. Flyweight chỉ ra cách chia sẻ ký pháp đầu cuối trong phạm vi của cây cú pháp trừu tượng. Interpreter thường sử dụng một Iterator để duyệt cấu trúc. Visitor có thể được sử dụng để duy trì hành vi trên mỗi nút trong cây cú pháp trừu tượng của lớp.

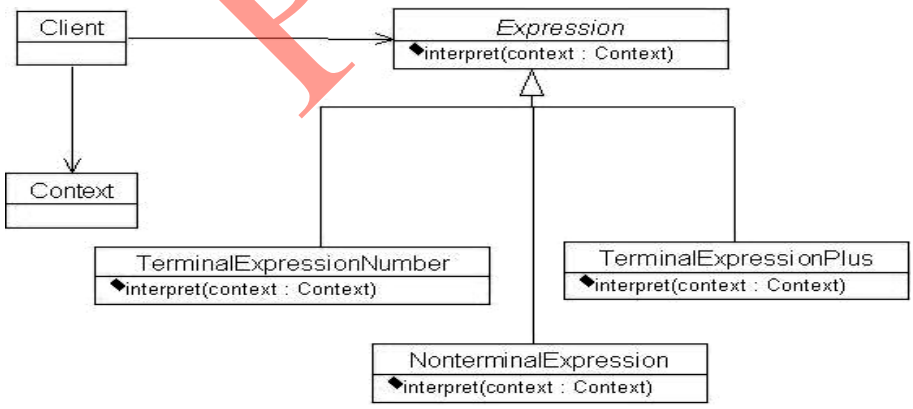
10.4.4 Ví dụ

Tính kết quả của biểu thức 5 + 3 x 3 + 6. Bài toán này có thể chia thành các bài toán con: Tính 3 x 3 = a; sau đó tính 5 + a = b; sau đó tính b + 6. Ta biểu diễn bài toán thành cấu trúc cây và sử dụng cách duyệt cây.



Hình 10.7: Duyệt cây tính toán

Dưới đây là biểu đồ cho ví dụ này (Hình 10.8) và mã nguồn tương ứng



Hình 10.8: Biểu đồ các lớp

```
import java.util.Stack;
public class Context extends Stack<Integer>{
}
public interface Expression {
        public void interpret(Context context);
}
```

```
public class TerminalExpressionNumber implements Expression {
        private int number;
        public TerminalExpressionNumber(int number){
                this.number = number;
        }
        public void interpret(Context context) {
                context.push(this.number);
        }
}

public class TerminalExpressionPlus implements Expression {
        public void interpret(Context context) {
                //Cong 2 phan tu phia tren dinh Stack
                context.push(context.pop() + context.pop());
        }
}

public class TerminalExpressionMutil implements Expression{
        public void interpret(Context context) {
                //Nhan 2 phan tu phia tren dinh Stack
                context.push(context.pop() * context.pop());
        }
}
//
import java.util.ArrayList;
public class NonterminalExpression implements Expression {
        private ArrayList<Expression> expressions;//tham chieu den
mang Exoression con
        public ArrayList<Expression> getExpressions() {
                return expressions;
        }
        public void setExpressions(ArrayList<Expression>
expressions) {
                this.expressions = expressions;
        }

        public void interpret(Context context) {
                if (expressions != null){
                        int size = expressions.size();
                        for (Expression e : expressions){
                                e.interpret(context);
                        }
                }
        }
}
//
import java.util.*;
```

```
public class Client {
        public static void main(String[] agrs){
                Context context = new Context();
        // 3 3 *
        ArrayList<Expression> treeLevel1 = new
ArrayList<Expression>();
        treeLevel1.add(new TerminalExpressionNumber(3));
        treeLevel1.add(new TerminalExpressionNumber(3));
        treeLevel1.add(new TerminalExpressionMutil());
        // 5 (3 3 *) +
        ArrayList<Expression> treeLevel2 = new
ArrayList<Expression>();
        treeLevel2.add(new TerminalExpressionNumber(5));
        Expression nonexpLevel1 = new NonterminalExpression();

((NonterminalExpression)nonexpLevel1).setExpressions(treeLevel1);
        treeLevel2.add(nonexpLevel1);
        treeLevel2.add(new TerminalExpressionPlus());
        // (5 (3 3 *) +) 6 +
        ArrayList<Expression> treeLevel3 = new
ArrayList<Expression>();
        Expression nonexpLevel2 = new NonterminalExpression();

((NonterminalExpression)nonexpLevel2).setExpressions(treeLevel2);
        treeLevel3.add(nonexpLevel2);
        treeLevel3.add(new TerminalExpressionNumber(6));
        treeLevel3.add(new TerminalExpressionPlus());

        for(Expression e : treeLevel3){
                e.interpret(context);
        }

        if (context != null)
                System.out.print("Ket qua: " + context.pop());
        }
}
```

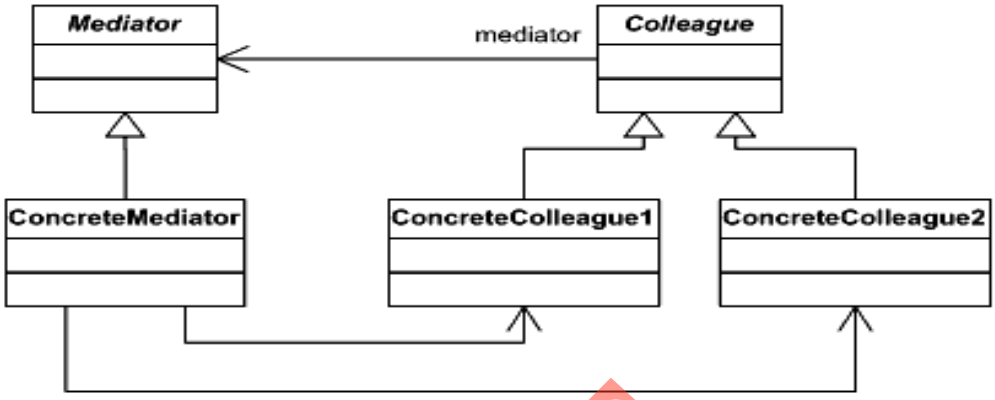

10.5 MẪU MEDIATOR

10.5.1 Đặt vấn đề

Cách thiết kế hướng đối tượng khuyến khích việc phân bố các hành vi giữa các đối tượng. Tuy nhiên, việc phân chia như vậy có thể dẫn đến một cấu trúc có nhiều liên kết giữa các đối tượng và trong trường hợp xấu nhất là tất cả các đối tượng đều có liên kết trực tiếp với nhau. Mẫu Mediator được dùng để đóng gói cách thức tương tác của một tập hợp các đối tượng và như vậy giảm bớt liên kết và cho phép thay đổi cách thức tương tác giữa các đối tượng một cách linh hoạt.

Mẫu Mediator có ưu điểm là giảm việc chia lớp con và liên kết giữa các đối tượng. Nó cũng làm đơn giản hoá giao tiếp giữa đối tượng bằng cách thay các tương tác nhiều - nhiều bằng tương tác 1- nhiều giữa nó và các đối tượng khác. Đồng thời nó trừu tượng hoá cách thức cộng tác giữa các đối tượng và tạo ra sự tách biệt rõ ràng hơn giữa các tương tác ngoài và các đặc tính bên trong của đối tượng. Nó sử dụng điều khiển trung tâm bằng cách thay sự phức tạp trong tương tác bằng sự phức tạp tại mediator.

10.5.2 Cấu trúc mẫu



Hình 10.9: Mẫu Mediator

Cấu trúc mẫu Mediator được cho trong Hình 10.9, trong đó:

- *Mediator* (IChatroom): định nghĩa một giao tiếp cho các đối tượng cộng tác. *ConcreteMediator* (Chatroom) Xây dựng các hành vi tương tác giữa các đối tượng *colleague* và xác định các đối tượng *colleague*
- *Colleague classes* (Participant) mỗi lớp *Colleague* đều xác định đối tượng *Mediator* tương ứng. Mỗi đối tượng *colleague* trao đổi với đối tượng *mediator* khi muốn trao đổi với *colleague* khác.

10.5.3 Tình huống áp dụng

Mẫu mediator có thể áp dụng trong các trường hợp sau:

- Một nhóm các đối tượng trao đổi thông tin một cách rõ ràng nhưng khá phức tạp và điều này có thể dẫn đến hệ thống có các kết nối phi cấu trúc và khó hiểu.
- Việc sử dụng lại một đối tượng gặp khó khăn vì nó liên kết với quá nhiều đối tượng khác. Mẫu này cho phép tuỳ biến một ứng xử được phân tán trong vài lớp mà không phải phân nhiều lớp con.
- Mediator khác với façade ở chỗ façade trừu tượng một hệ thống con của các đối tượng để cung cấp một giao diện tiện ích hơn. Giao thức của nó theo một hướng duy nhất đó là các đối tượng Facade tạo ra các yêu cầu của các lớp hệ thống con nhưng không có chiều ngược lại. Ngược lại, Mediator cho phép kết hợp các hành vi mà các đối tượng cộng tác không thể cung cấp.
- Các cộng tác có thể giao tiếp với Mediator bằng cách sử dụng mẫu Observer.

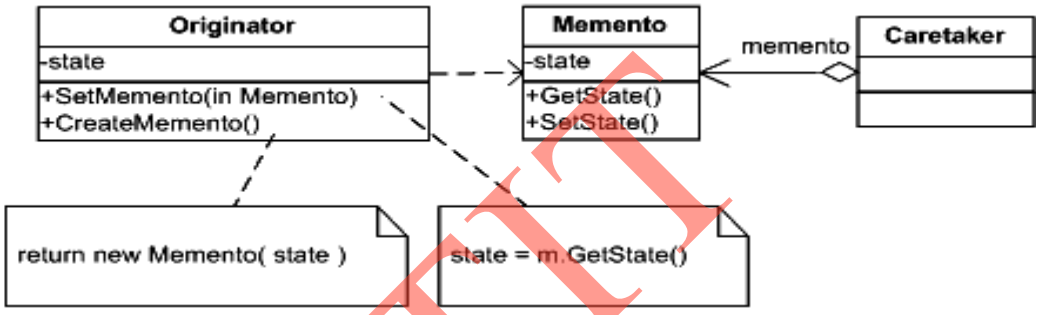
10.6 MẪU MEMENTO

10.6.1 Đặt vấn đề

Đôi lúc, việc lưu lại trạng thái của một đối tượng là cần thiết. Ví dụ khi xây dựng cơ chế checkpoints và undo để phục hồi hệ thống khỏi trạng thái lỗi. Tuy nhiên, các đối tượng thường phải che dấu một vài hoặc tất cả các thông tin trạng thái của mình, làm cho chúng không thể được truy cập từ bên ngoài. Vấn đề này có thể giải quyết với mẫu Memento.

Memento là đối tượng có chức năng lưu lại trạng thái nội tại của một đối tượng khác. Cơ chế undo sẽ yêu cầu một memento ban đầu khi nó cần khôi phục lại trạng thái ban đầu của đối tượng. Chỉ có đối tượng ban đầu mới có quyền truy xuất và lưu trữ thông tin vào memento do nó trong suốt đối với các đối tượng còn lại. Như vậy, Memento là mẫu thiết kế có thể lưu lại trạng thái của một đối tượng để khôi phục lại sau này mà không vi phạm nguyên tắc đóng gói.

10.6.2 Cấu trúc mẫu



Hình 10.10: Mẫu Memento

Trong đó

- *Memento*: lưu trữ trạng thái của đối tượng *Originator* và bảo vệ, chống truy cập từ các đối tượng khác *Originator*.
- *Originator*: Tạo memento chứa bản chụp trạng thái của mình và sử dụng memento để khôi phục về trạng thái cũ.
- *Caretaker*: Có trách nhiệm quản lý các memento, nó không thay đổi hoặc truy xuất nội dung của memento.

10.6.3 Tình huống áp dụng

- Cần lưu lại trạng thái nội bộ của một đối tượng để có thể khôi phục trạng thái đó sau này. Việc xây dựng giao diện trực tiếp để truy xuất thông tin trạng thái sẽ làm lộ diện cấu trúc và phá hỏng tính đóng gói của đối tượng.
- Các Command có thể sử dụng các memento để duy trì trạng thái cho các thao tác có khả năng phục hồi được. Các Memento có thể được sử dụng cho vòng lặp sớm hơn.
- Trong các thiết kế hỗ trợ khôi phục trạng thái khác, Originator có chức năng lưu trữ các phiên bản trạng thái của mình và do đó phải thi hành các chức năng quản lý lưu

trữ. Việc sử dụng memento có thể gây ra chi phí lớn nếu Originator có nhiều thông tin trạng thái cần lưu trữ hoặc nếu việc ghi lại và khôi phục trạng thái diễn ra với tần suất lớn. Việc đảm bảo chỉ originator mới có quyền truy cập memento là tương đối khó xây dựng ở một số ngôn ngữ lập trình.

- Chi phí ẩn của việc lưu trữ memento: caretaker có nhiệm vụ quản lý cũng như xoá bỏ các memento mà nó yêu cầu tạo ra. Tuy nhiên nó không biết được kích thước của memento là bao nhiêu và do đó có thể tốn nhiều không gian bộ nhớ khi lưu trữ memento.

10.7 MẪU OBSERVER

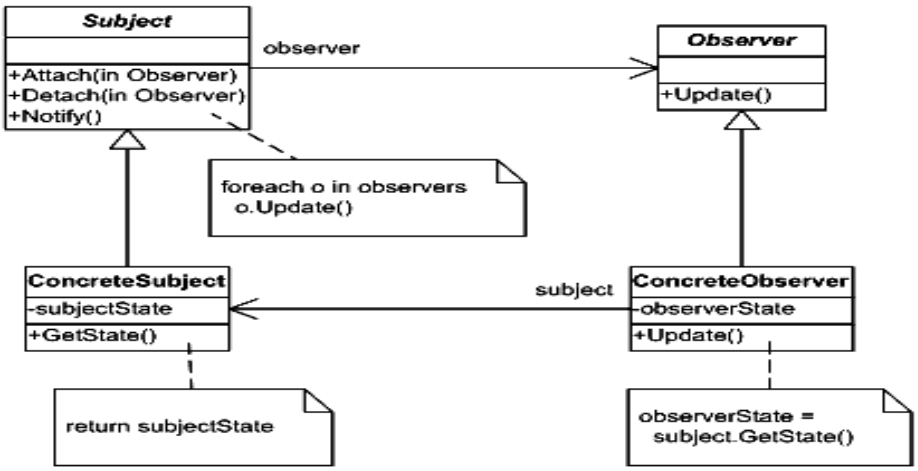
10.7.1 Đặt vấn đề

Mẫu Observer ích lợi trong thiết kế mô hình giao tiếp giữa một đối tượng và một tập đối tượng phụ thuộc nhau. Nó cho phép đồng bộ hóa trạng thái của tập đối tượng với đối tượng kia. và định nghĩa phụ thuộc 1- nhiều giữa các đối tượng để khi một đối tượng thay đổi trạng thái thì tất cả các phụ thuộc của nó được nhận biết và cập nhật tự động.

10.7.2 Cấu trúc mẫu

Cấu trúc mẫu được thể hiện trong Hình 10.11, trong đó

- Subject: hiểu về các Observer của nó. Một số lượng bất kỳ Observer có thể theo dấu một chủ thể nào đó và cung cấp một giao diện cho việc gắn và tách các đối tượng Observer
- ConcreteSubject: Lưu trữ trạng thái của ConcreteObserver cần quan tâm và gửi tín hiệu đến các observer của nó khi trạng thái của nó thay đổi.
- Observer: Định nghĩa một giao diện cập nhật cho các đối tượng mà sẽ nhận tín hiệu của sự thay đổi tại chủ thể.
- ConcreteObserver: Duy trì một tham chiếu tới một đối tượng ConcreteSubject và lưu trữ các trạng thái cố định. Nó cài đặt giao diện cập nhật của Observer để giữ các trạng thái cố định của nó.



Hình 10.11: Mẫu Observer

10.7.3 Tình huống áp dụng

- Theo dõi thay đổi của doanh thu bán hàng ví dụ cho như quản ly báo cáo của doanh nghiệp với nhiều bộ phận.

10.8 MẪU STATE

10.8.1 Đặt vấn đề

Trạng thái của một đối tượng có thể được xác định như điều kiện tại thời điểm đã cho phụ thuộc vào các giá trị của thuộc tính nó. Tập các phương thức cài đặt bởi một lớp tạo nên hành vi của một thể hiện hay đối tượng của nó. Bất kỳ một thay đổi nào của các giá trị thuộc tính thì ta nói trạng thái của đối tượng đã thay đổi. Ví dụ khi người sử dụng chọn kiểu font hay màu trong bộ soạn thảo HTML thì tính chất của đối tượng soạn thảo thay đổi theo và có thể xem như thay đổi trạng thái bên trong của nó.

Mẫu state có thể sử dụng trong thiết kế cấu trúc của lớp để cho các thể hiện của lớp có thể tồn tại ở các trạng thái khác nhau và thể hiện hành vi khác nhau tùy theo trạng thái. Lớp như thế được gọi là lớp ngữ cảnh Context và đối tượng tương ứng của nó có thể thay đổi hành vi khi có sự thay đổi trạng thái bên trong và cũng được xem như đối tượng có trạng thái.

10.8.2 Cấu trúc mẫu

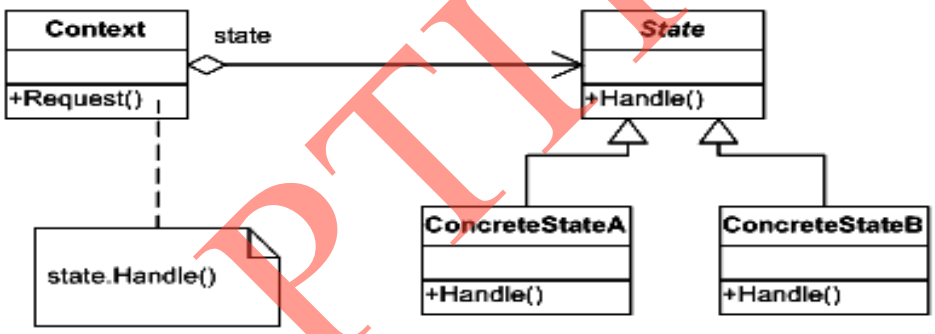


Figure 10.12: Cấu trúc mẫu state

Trong đó

- Context định nghĩa giao diện mà đối tượng khách quan tâm, nó duy trì một thể hiện của một lớp ConcreteState để định nghĩa trạng thái hiện tại
- State: Định nghĩa một giao diện cho việc đóng gói hành vi kết hợp với trạng thái đặc biệt của Context.
- ConcreteState (RedState, SilverState, GoldState) là cài đặt của State, mỗi lớp con cài đặt một hành vi kết hợp với một trạng thái của Context.

10.8.3 Tình huống áp dụng

Mẫu State thường được kết hợp với mẫu Flyweight để giải thích khi nào cũng như cách các đối tượng State có thể được phân tách. Các đối tượng State thường là các Singleton.

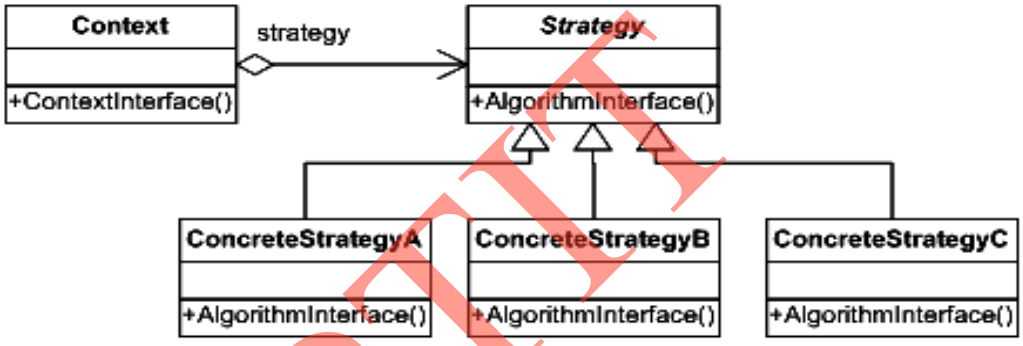
10.9 MẪU STRATEGY

10.9.1 Đặt vấn đề

Strategy là mẫu thiết kế được sử dụng khi một đối tượng client cần chọn một thuật toán thích hợp từ một tập thuật toán có liên quan với nhau. Mẫu này đề nghị cài đặt mỗi thuật toán trong một lớp tách biệt. Mỗi thuật toán đóng gói trong một lớp tách biệt gọi là một *chiến lược* strategy và đối tượng sử dụng *đối tượng chiến lược* gọi là *đối tượng ngữ cảnh* context object. Như vậy, với những đối tượng chiến lược khác nhau, việc thay đổi hành vi của đối tượng ngữ cảnh đơn giản là thay đổi đối tượng chiến lược của nó thành đối tượng cài đặt thuật toán mong muốn.

Để giúp cho đối tượng ngữ cảnh truy nhập các đối tượng chiến lược khác nhau, thì tất cả đối tượng chiến lược phải được thiết kế với cùng giao tiếp. Trong ngôn ngữ lập trình java, ta có thể thiết kế các đối tượng chiến lược như một cài đặt của giao tiếp chung hay như lớp con của lớp trừu tượng chung.

10.9.2 Cấu trúc mẫu



Hình 10.13: Cấu trúc mẫu Strategy

Cấu trúc mẫu thể hiện trong Hình 10.13. Trong đó

- Strategy: Khai báo một giao diện thông thường cho tất cả các thuật toán được hỗ trợ.
- Context sử dụng giao diện này để gọi các thuật toán được định nghĩa bởi một ConcreteStrategy.
- ConcreteStrategy (ví dụ như QuickSort, ShellSort, MergeSort): Cài đặt thuật toán sử dụng giao diện Strategy và được cấu hình với một đối tượng ConcreteStrategy. Nó duy trì một tham chiếu tới một đối tượng Stategy.

10.9.3 Tình huống áp dụng

Các đối tượng Strategy thường tạo ra các Flyweight tốt hơn.

10.10 MẪU TEMPLATE METHOD

10.10.1 Đặt vấn đề

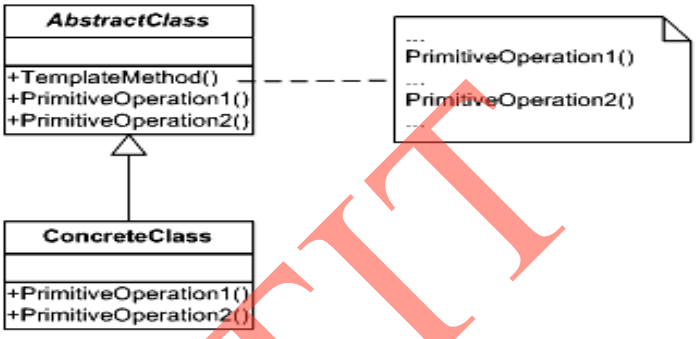
Phương thức khung (Template Method) có thể được sử dụng trong những tình huống khi có một thuật toán mà một số bước của nó có thể cài đặt theo nhiều cách khác nhau. Mẫu

này cho rằng khi đó nên đưa sườn thuật toán thành một phương thức tách biệt gọi là *phương thức khung* trong một lớp gọi là *lớp khung* (Template class) và cài đặt các phần còn lại của thuật toán trong những lớp con của lớp này. Với ngôn ngữ java, lớp khung Template có thể thiết kế theo hai cách: lớp trừu tượng hay lớp cụ thể.

10.10.2 Cấu trúc mẫu

Cấu trúc mẫu được cho trong Hình 10.14. Trong đó:

- AbstractClass: Định nghĩa các thao tác nguyên thủy trừu tượng, các thao tác này định nghĩa các lớp con cụ thể để thực hiện các bước của một thuật toán. Nó cài đặt một templateMethod() để định nghĩa khung của một thuật toán. templateMethod() này gọi các thao tác nguyên thủy cũng như các thao tác được định nghĩa trong AbstractClass hoặc một số các đối tượng khác.
- ConcreteClass: Thực thi các thao tác nguyên thủy để thực hiện các bước đã chỉ ra trong các lớp con của thuật toán.



Hình 10.14: Mẫu Template method

10.10.3 Tình huống áp dụng

Phương thức khung nên sử dụng trong các trường hợp sau đây:

- Thực hiện các phần cố định của một thuật toán khi đặt nó vào các lớp con để thực hiện hành vi có thể thay đổi.
- Khi các lớp hành vi thông thường cần được phân tách và khoanh vùng trong một lớp thông thường để tránh sự giống nhau về mã.
- Điều khiển mở rộng các lớp con. Ta có thể định nghĩa một phương thức khung, phương thức này gọi các thao tác cụ thể tại các điểm đặc biệt, bằng cách đó cho phép các mở rộng chỉ tại các điểm đó.
- Các phương thức khung sử dụng tính kế thừa để thay đổi các bộ phận của một thuật toán. Các chiến lược Strategy sử dụng sự ủy nhiệm để thay đổi hoàn toàn một thuật toán.

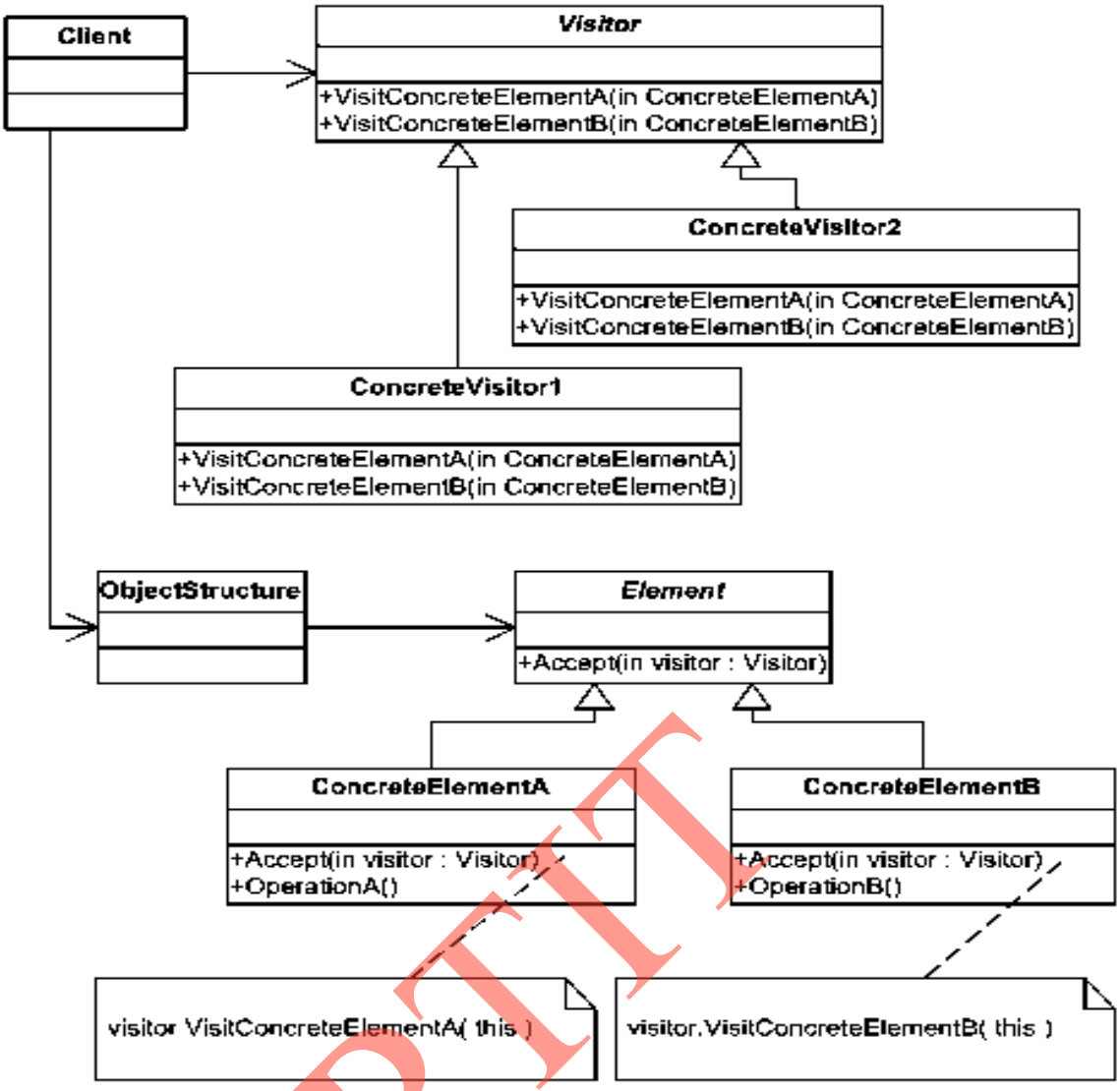
10.11 MẪU VISITOR

10.11.1 Đặt vấn đề

Visitor là mẫu thiết kế xác định khung của một giải thuật theo một số bước của các phân cấp lớp.

10.11.2 Cấu trúc mẫu

- *Visitor:* Đưa ra một thao tác Visit cho mỗi lớp của ConcreteElement trong cấu trúc đối tượng. Tên và dấu hiệu của các thao tác này nhận dạng lớp gửi yêu cầu Visit tới visitor, nó cho phép visitor quyết định lớp cụ thể nào của thành phần được thăm. Sau đó visitor có thể truy nhập thành phần trực tiếp thông qua giao diện đặc biệt của nó.
- *ConcreteVisitor:* Thực hiện mỗi thao tác được đưa ra bởi Visitor. Mỗi thao tác thực hiện một phần của giải thuật định nghĩa cho lớp phù hợp của đối tượng trong cấu trúc. ConcreteVisitor cung cấp ngữ cảnh cho giải thuật và lưu trữ trạng thái cục bộ của nó.
- *Element:* Định nghĩa một phương thức Accept, phương thức này sử dụng một visitor như là một đối số.
- *ConcreteElement:* Cài đặt phương thức Accept, phương thức này sử dung visitor như là một đối số.
- *ObjectStructure:* Có thể đếm các thành phần của nó để cung cấp một giao diện mức cao cho phép visitor thăm các thành phần của nó như một composite hoặc một sưu tập như danh sách hay tập hợp.



Hình 10.15: Mẫu Visitor

10.11.3 Tình huống áp dụng

Mẫu Visitor có thể được sử dụng khi:

- Một cấu trúc đối tượng chứa đựng nhiều lớp của các đối tượng với các giao diện khác nhau, và việc thực hiện các phương thức trên các đối tượng này đòi hỏi các lớp cụ thể của chúng.
- Nhiều phương thức khác nhau không có mối liên hệ nào cần được thực hiện trên các đối tượng trong một cấu trúc đối tượng, và ta muốn tránh "làm hỏng" các lớp của chúng khi thực hiện các thao tác đó. Visitor cho phép ta giữ các thao tác có mối liên hệ với nhau bằng cách định nghĩa chúng trong một lớp. Khi một cấu trúc đối tượng được chia sẻ bởi nhiều ứng dụng, sử dụng Visitor để đặt các thao tác này vào trong các ứng dụng cần thiết.

- Các lớp định nghĩa các cấu trúc đối tượng hiếm khi thay đổi, nhưng ta muốn định nghĩa các thao tác mới trên các cấu trúc. Thay đổi các lớp cấu trúc yêu cầu định nghĩa lại giao diện cho tất cả các visitor.
- Các Visitor có thể được sử dụng để cung cấp một thao tác trên một cấu trúc đối tượng được định nghĩa bởi mẫu Composite.

10.12 KẾT LUẬN

Trong chương này chúng ta đã xem xét một số mẫu thiết kế hành vi. Các mẫu thiết kế hành vi dành cho việc thiết kế hiệu quả các phương thức trong lớp. Nhiều ứng dụng sử dụng các mẫu thiết kế này để giải quyết một số vấn đề trong phát triển phần mềm đã được chứng tỏ là hiệu quả. Chương này đã trình bày 11 mẫu thiết kế với cấu trúc mẫu, tình huống áp dụng.

BÀI TẬP

1. Hoàn thiện chương trình để chạy cho các mẫu đã được trình bày trong chương này
2. Hoàn thiện Case study 1 để chương trình có thể chạy được
3. Hoàn thiện Case study 2 để chương trình chạy được. Xem xét thêm vào một số mẫu thiết kế và cài đặt cho hệ thống này.
4. Một đơn đặt hàng của một cửa hàng online có thể là một trong các trạng thái sau:
   - Không gửi đi
   - Đã gửi đi
   - Đã nhận
   - Đã xử lý
   - Đã chuyển hàng
   - Hủy

   a. Hãy xác định bảng chuyển vị trạng thái cho đơn đặt hàng

   b. Thiết kế lớp Order để biểu diễn đơn đặt hàng. Thiết kế hành vi của đơn đặt hàng dưới dạng tập các lớp trạng thái State với lớp cha chung.
5. Thiết kế lớp đăng ký của hệ thống quản lý học theo tín chỉ dựa vào tập các trạng thái đăng ký của sinh viên cùng các trạng thái tương ứng.
6. Một hệ quản lý thư viện cho phép bạn đọc đăng ký mượn qua mạng. Hãy thiết kế lớp đăng ký của hệ thống này dựa vào tập các trạng thái có thể khi bạn đọc đăng ký.
7. Thiết kế lớp đăng ký của hệ thống quản lý tour du lịch dựa vào tập các trạng thái đăng ký tour của khách.

CHƯƠNG 11: CASE STUDY

Chương này trình bày hai Case study để minh họa kết hợp các mẫu thiết kế cho phát triển các ứng dụng:

- Case study 1: Thiết kế cơ sở truy nhập dữ liệu
- Case study 2: Hệ quản lý bán sách trực tuyến BookStore

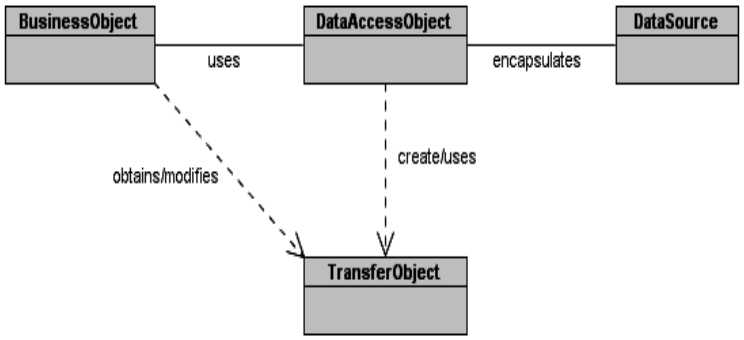
11.1 CASE STUDY 1: THIẾT KẾ CƠ CHẾ TRUY NHẬP DỮ LIỆU

Việc truy cập dữ liệu cần phải thay đổi tùy theo nguồn dữ liệu nghĩa là tùy thuộc vào kiểu của nó như CSDL quan hệ, CSDL hướng đối tượng, file…và cách cài đặt của chúng. Các ứng dụng có thể sử dụng JDBC API để truy cập dữ liệu trong hệ quản trị CSDL. JBDC API cho phép các ứng dụng JDBC sử dụng các lệnh SQL để truy cập dữ liệu. Tuy nhiên, trong môi trường như hệ quản trị CSDL quan hệ, cú pháp và định dạng của các lệnh SQL thực tế cũng biến đổi tùy thuộc vào sản phẩm CSDL cụ thể.

Giải pháp là sử dụng đối tượng truy nhập dữ liệu Data Access Object (DAO) để trừu tượng và đóng gói các truy cập tới nguồn dữ liệu. DAO quản lý kết nối với nguồn dữ liệu để lấy và lưu trữ dữ liệu và cài đặt cơ chế truy cập cần thiết để làm việc với nguồn dữ liệu. Tham khảo chi tiết: http://www.oracle.com/java/dataaccessobject.html

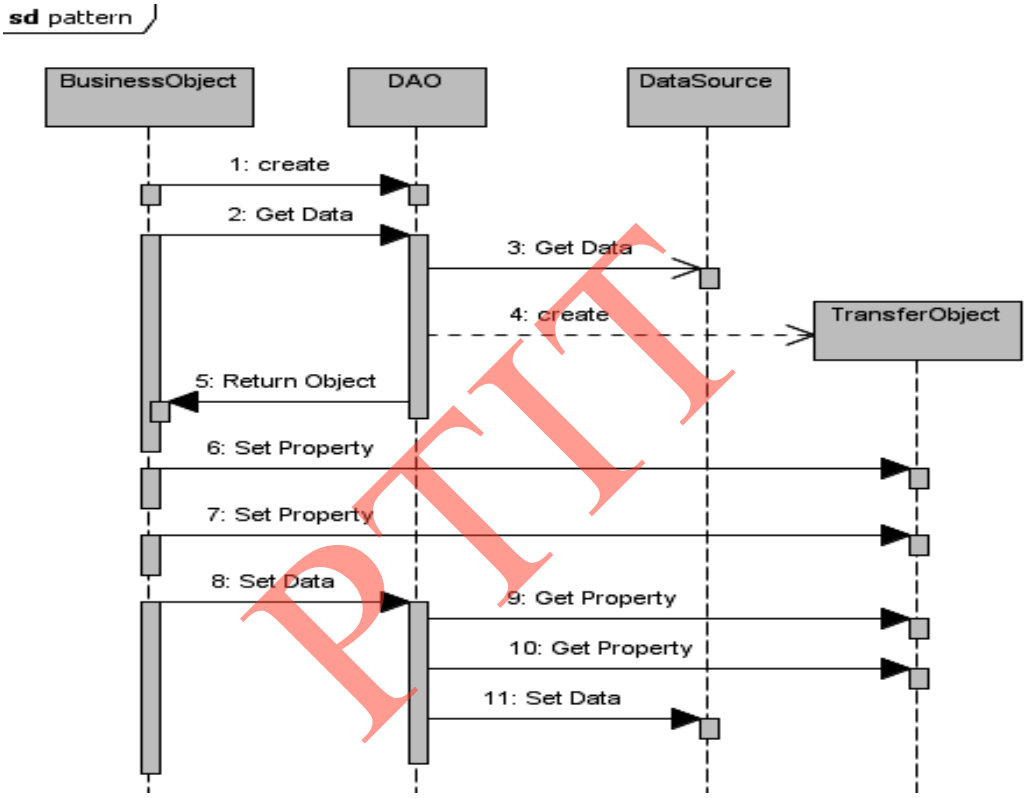
11.1.1 Cấu trúc của DAO

- *BusinessObject:* Biểu diễn phía yêu cầu dữ liệu. Nó là đối tượng yêu cầu truy nhập tới nguồn dữ liệu để lấy và lưu trữ dữ liệu. Nó có thể được cài đặt là các bean hoặc đối tượng Java khác như servlet để truy cập tới nguồn dữ liệu.
- *DataAccessObject*: Là đối tượng chính của mẫu này. Nó trừu tượng hóa các cài đặt việc truy cập dữ liệu bên dưới để BusinessObject có thể truy cập tới nguồn dữ liệu một cách trong suốt. BusinessObject cũng giao phó thao tác nạp và lưu trữ dữ liệu cho DataAccessObject.
- *DataSource*: Biểu diễn cài đặt của nguồn dữ liệu. Nguồn dữ liệu có thể là CSDL như RDBMS, OODBMS, kho chứa XML, hệ thống file…
- *TransferObject:* được dùng như một vật mang dữ liệu. DataAccessObject có thể sử dụng TransferObject để trả dữ liệu về phía yêu cầu. DataAccessObject cũng có thể nhận dữ liệu từ phía yêu cầu trong TransferObject để cập nhật dữ liệu trong nguồn dữ liệu.



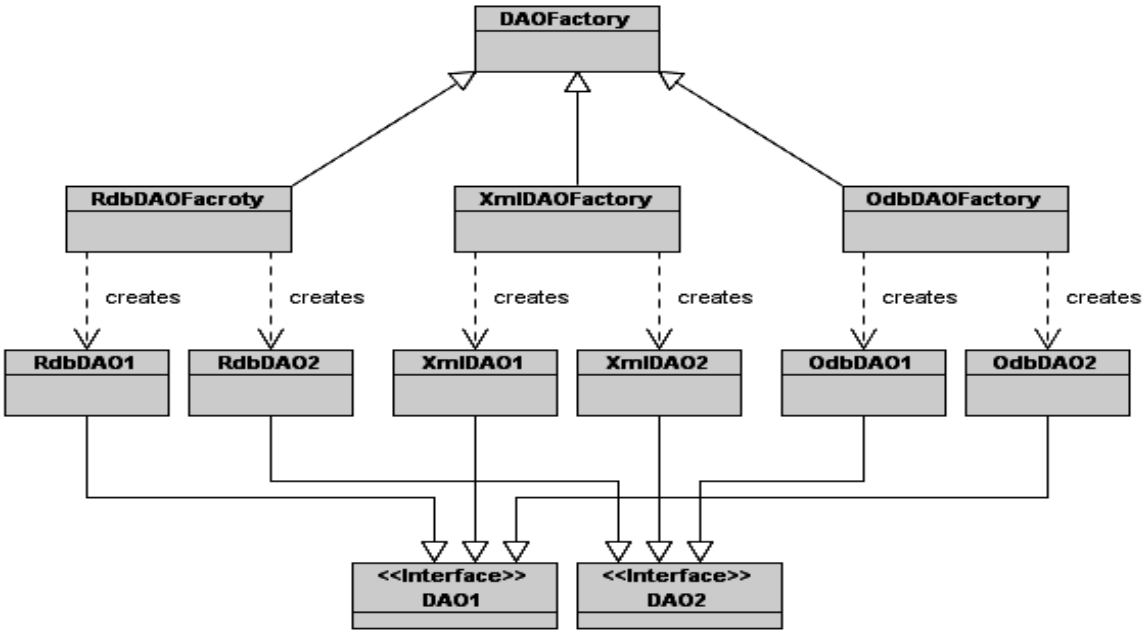
Hình 11.1: Cấu trúc của DAO

Biểu đồ giao tiếp theo kiểu tuần tự được cho trong Hình 11.1.



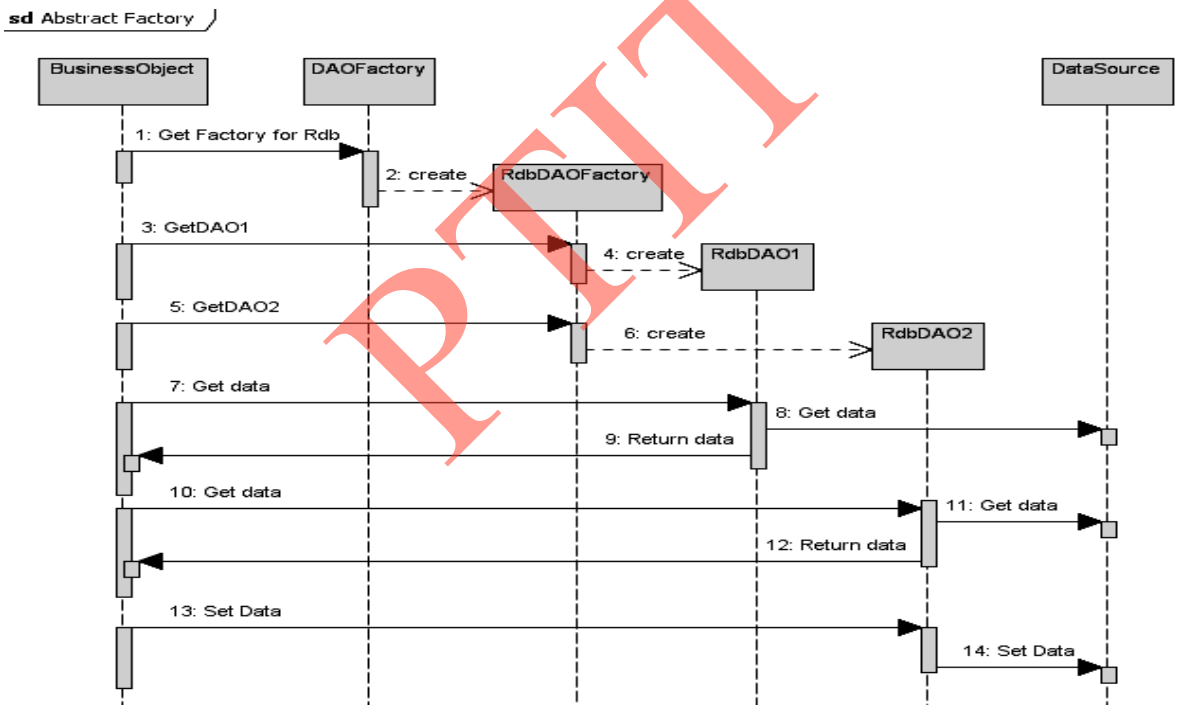
Hình 11.2: Biểu đồ tuần tự của DAO

DAO có thể sử dụng với mẫu Factory Method (Hình 11.3).



Hình 11.3: Biểu đồ lớp của DAO theo chiến lược Factory Method

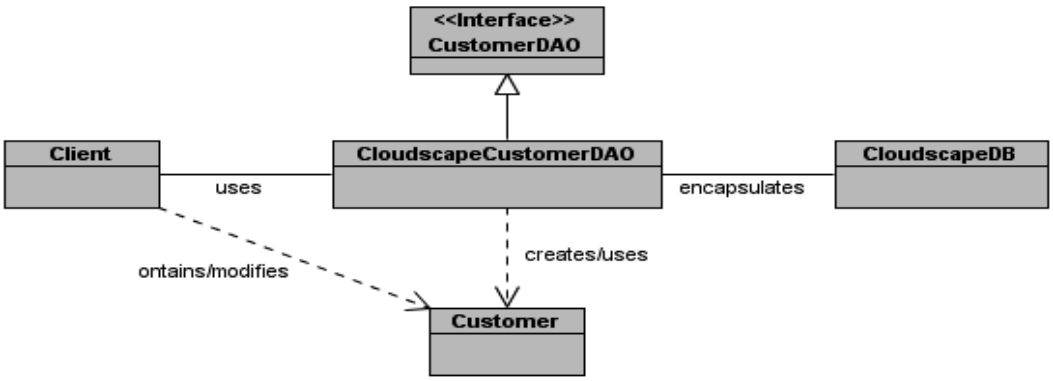
Biểu đồ tuần tự mô tả giao tiếp tuần tự theo chiến lược Factory method (Hình 11.4)



Hình 11.4: Biểu đồ tuần tự của DAO theo chiến lược Factory Method

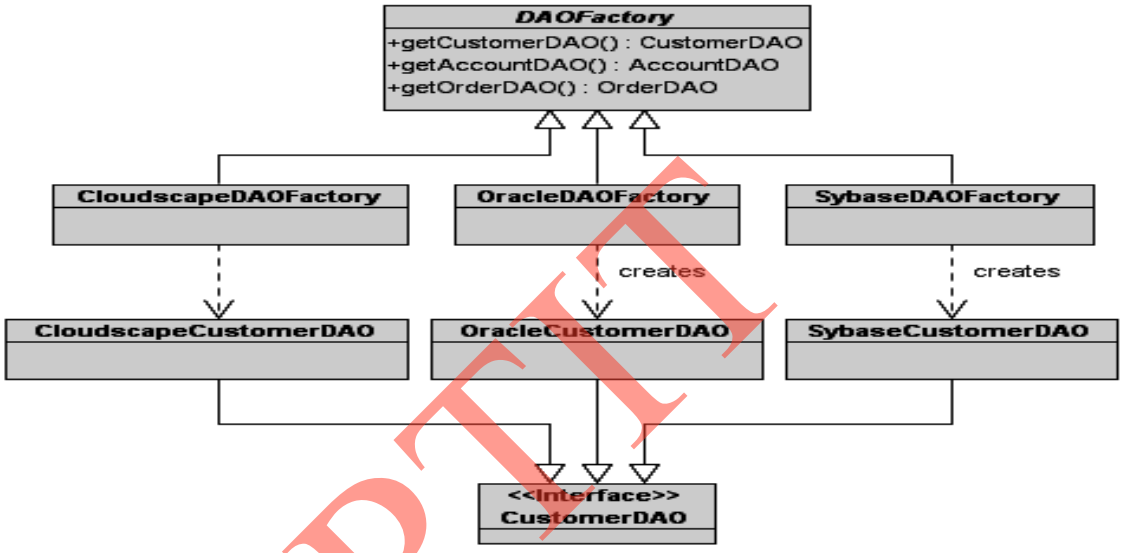
11.1.3 Hệ quản lý dữ liệu khách hàng

Ta xem xét hệ quản lý dữ liệu khách hàng. Biểu đồ lớp của quản lý dữ liệu khách hàng cho trong Hình 11.5



Hình 11.5: Biểu đồ lớp của CustomerDAO

Ta có thể cài đặt bằng cách sử dụng chiến lược Factory Method (Hình 11.6)



Hình 11.6: Biểu đồ lớp của CustomerDAO theo chiến lược Factory Method

Một phần mã nguồn được cho dưới đây

```
// Abstract class DAO Factory
public abstract class DAOFactory {
        // List of DAO types supported by the factory
        public static final int CLOUDSCAPE = 1;
        public static final int ORACLE = 2;
        public static final int SYBASE = 3;
        ...
// There will be a method for each DAO that can be
// created. The concrete factories will have to
// implement these methods.
        public abstract CustomerDAO getCustomerDAO();
        public abstract AccountDAO getAccountDAO();
        public abstract OrderDAO getOrderDAO();
        ...
        public static DAOFactory getDAOFactory(
```

```
            int whichFactory) {

          switch (whichFactory) {
            case CLOUDSCAPE:
                return new CloudscapeDAOFactory();
            case ORACLE    :
                return new OracleDAOFactory();
            case SYBASE    :
                return new SybaseDAOFactory();
            ...
            default           :
                return null;
          }
        }
      }
//Hiện thực hóa cài đặt DAOFactory cho Cloudscape

import java.sql.*;
public class CloudscapeDAOFactory extends DAOFactory {
        public static final String DRIVER=
          "COM.cloudscape.core.RmiJdbcDriver";
        public static final String DBURL=
          "jdbc:cloudscape:rmi://localhost:1099/CoreJ2EEDB";

        // method to create Cloudscape connections
        public static Connection createConnection() {
          // Use DRIVER and DBURL to create a connection
          // Recommend connection pool implementation/usage
        }
        public CustomerDAO getCustomerDAO() {
          // CloudscapeCustomerDAO implements CustomerDAO
          return new CloudscapeCustomerDAO();
        }
        public AccountDAO getAccountDAO() {
          // CloudscapeAccountDAO implements AccountDAO
          return new CloudscapeAccountDAO();
        }
        public OrderDAO getOrderDAO() {
          // CloudscapeOrderDAO implements OrderDAO
          return new CloudscapeOrderDAO();
        }
        ...
      }
//Cài đặt DAO Interface cho Customer
public interface CustomerDAO {
        public int insertCustomer(...);
        public boolean deleteCustomer(...);
        public Customer findCustomer(...);
```

```
        public boolean updateCustomer(...);
        public RowSet selectCustomersRS(...);
        public Collection selectCustomersTO(...);
        ...
      }
// Cài đặt Cloudscape DAO cho Customer
// CloudscapeCustomerDAO implementation of the
// CustomerDAO interface. This class can contain all
// Cloudscape specific code and SQL statements.
// The client is thus shielded from knowing
// these implementation details.

import java.sql.*;

public class CloudscapeCustomerDAO implements CustomerDAO {
      public CloudscapeCustomerDAO() {
      // initialization
        }
      // The following methods can use
      // CloudscapeDAOFactory.createConnection()
      // to get a connection as required
      public int insertCustomer(...) {
      // Implement insert customer here.
      // Return newly created customer number
      // or a -1 on error
        }

        public boolean deleteCustomer(...) {
          // Implement delete customer here
          // Return true on success, false on failure
        }

      public Customer findCustomer(...) {
          // Implement find a customer here using supplied
          // argument values as search criteria
          // Return a Transfer Object if found,
          // return null on error or if not found
        }
      public boolean updateCustomer(...) {
          // implement update record here using data
          // from the customerData Transfer Object
          // Return true on success, false on failure or
          // error
        }
      public RowSet selectCustomersRS(...) {
          // implement search customers here using the
          // supplied criteria.
```

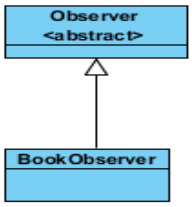
```
          // Return a RowSet.
        }
      public Collection selectCustomersTO(...) {
          // implement search customers here using the
          // supplied criteria.
          // Alternatively, implement to return a Collection
          // of Transfer Objects.
        }
        ...
      }
//Customer Transfer Object
      public class Customer implements java.io.Serializable {
        // member variables
        int CustomerNumber;
        String name;
        String streetAddress;
        String city;
        ...
        // getter and setter methods...
        ...
      }
//Sử dụng DAO và DAOFactory ở client code
      ...
// create the required DAO Factory
      DAOFactory cloudscapeFactory =
        DAOFactory.getDAOFactory(DAOFactory.DAOCLOUDSCAPE);

      // Create a DAO
      CustomerDAO custDAO =
        cloudscapeFactory.getCustomerDAO();

      // create a new customer
      int newCustNo = custDAO.insertCustomer(...);

      // Find a customer object. Get the Transfer Object.
      Customer cust = custDAO.findCustomer(...);

      // modify the values in the Transfer Object.
      cust.setAddress(...);
      cust.setEmail(...);
      // update the customer object using the DAO
      custDAO.updateCustomer(cust);

      // delete a customer object
      custDAO.deleteCustomer(...);
      // select all customers in the same city
      Customer criteria=new Customer();
      criteria.setCity("New York");
```

```
    Collection customersList =
      custDAO.selectCustomersTO(criteria);
    // returns customersList - collection of Customer
    // Transfer Objects. iterate through this collection to
    // get values.
    ...
```

11.2 CASE STUDY 2: HỆ QUẢN LÝ BÁN SÁCH TRỰC TUYẾN BOOKSTORE

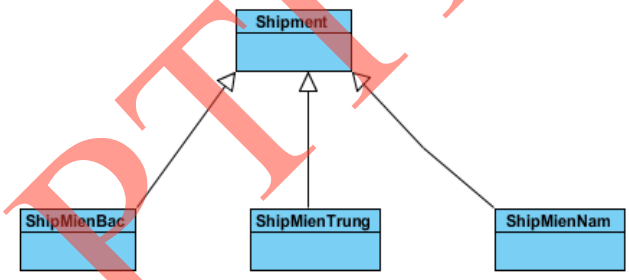
Phần này trình bày áp dụng một số mẫu Façade, Observer và Factory để thiết kế một số chức năng của Hệ Quản lý bán sách Bookstore.

Áp dụng Observer



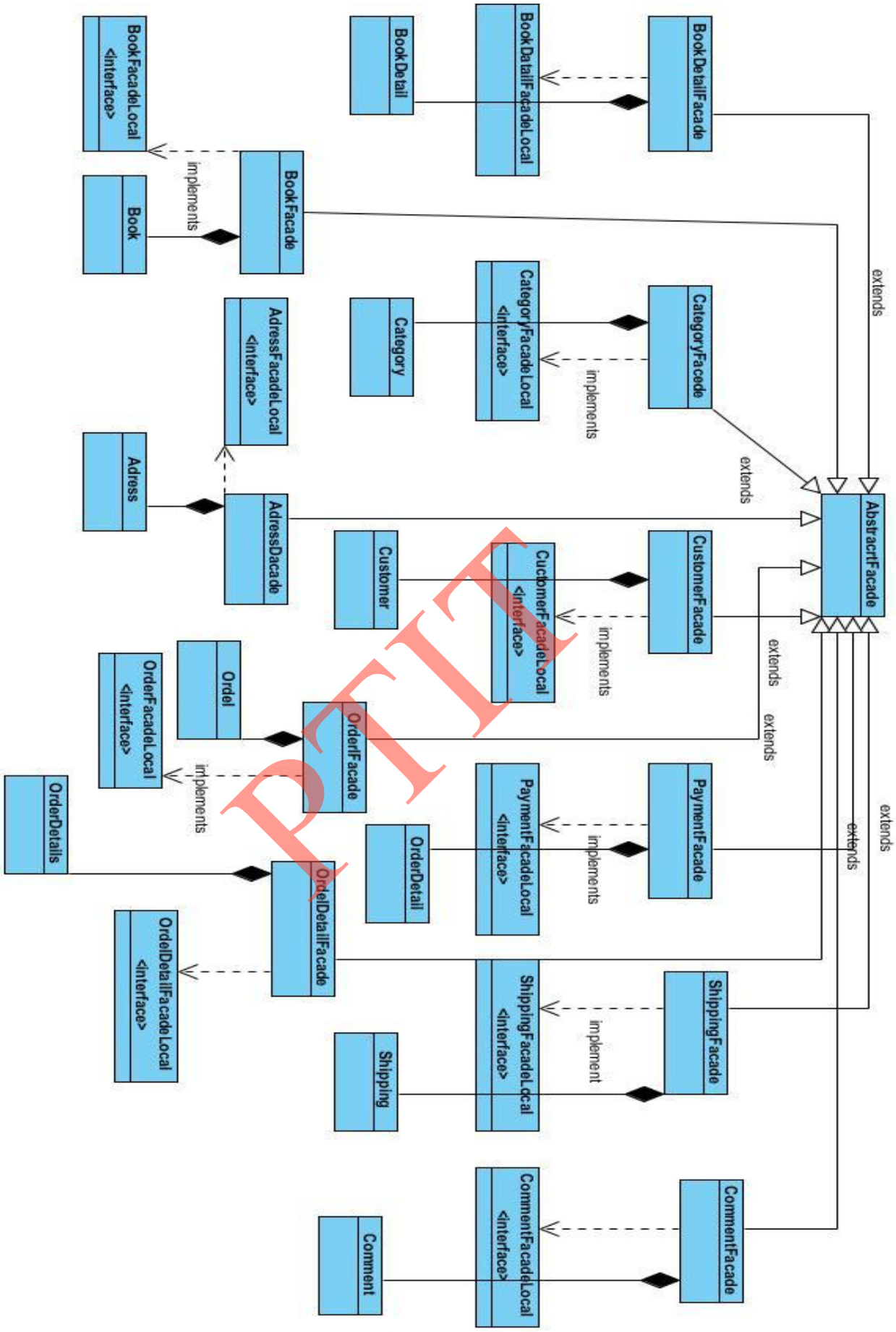
Hình 11.7: Biểu đồ Observer cho Bookstore

Áp dụng Factory



Hình 11.8: Biểu đồ Factory cho Bookstore

Áp dụng Façade



Hình 11.9: Biểu đồ Façade cho Bookstore

MÃ NGUỒN

```
package entity;
import java.io.Serializable;
import java.util.ArrayList;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import java.util.List;
import javax.persistence.CascadeType;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.OneToMany;
import javax.validation.constraints.Null;

@Entity
public class Book implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
    private String name;
    private String author;
    private String state;
    private Boolean status;
    private Integer price;
    private String image;
    private int quanity;
    private String description;
    @OneToMany(mappedBy = "book", cascade= CascadeType.REMOVE)
private List<BookObserver> booksObserver = new
ArrayList<BookObserver>();
```

```
    @ManyToOne
    @JoinColumn(name = "category_id")
    private Category category;

    public void attach(BookObserver bo) {
        if (bo == null) {
            throw new NullPointerException("Null Observer");
        }
        if (!booksObserver.contains(bo)) {
            booksObserver.add(bo);
        }
    }
    private boolean changes = false;

    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    @Override
    public int hashCode() {
        int hash = 0;
        hash += (int) id;
        return hash;
    }
    @Override
    public boolean equals(Object object) {
        // TODO: Warning - this method won't work in the case the
id fields are not

set
        if (!(object instanceof Book)) {
            return false;
```

```
        }
        Book other = (Book) object;
        if (this.id != other.id) {
            return false;
        }
        return true;
    }
    @Override
    public String toString() {
        return "entity.Book[ id=" + id + " ]";
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getAuthor() {
        return author;
    }
    public void setAuthor(String author) {
        this.author = author;
    }
    public List<BookObserver> getBooksObserver() {
        return booksObserver;
    }
public void setBooksObserver(List<BookObserver> booksObserver) {
        this.booksObserver = booksObserver;
    }
    public String getState() {
        return state;
    }
    public void setState(String state) {
        if (this.state == null) {
```

```
            this.state = state;
        } else if (!this.state.equals(state)) {
            this.changes = true;
        }
        this.state = state;
        notifyObservers();
    }
    public void notifyObservers() {
        for (BookObserver observer : booksObserver) {
            observer.update();
        }
    }
    public Boolean isStatus() {
        return status;
    }
    public void setStatus(Boolean status) {
        if (this.status != null) {
            changes = true;
            if (status == false) {
                setState("Het Hang");
            } else {
               setState("Co Hang");
            }
            System.out.println("ok");
        }
        this.status = status;
        System.out.println("not ok");
    }
    public Integer getPrice() {
        return price;
    }
    public void setPrice(Integer price) {
        if (this.price != null) {
            this.changes = true;
```

```
            if (this.price < price) {
                setState("Tang gia " + (price - this.price));
            } else if (this.price == price) {
                changes=false;
            } else {
                setState("Giam gia " + (this.price - price));
            }
        }
        this.price = price;
    }
    public String getImage() {
        return image;
    }
    public void setImage(String image) {
        this.image = image;
    }

    public int getQuanity() {
        return quanity;
    }
    public void setQuanity(int quanity) {
        this.quanity = quanity;
    }
    public String getDescription() {
        return description;
    }
    public void setDescription(String description) {
        this.description = description;
    }
    public Category getCategory() {
        return category;
    }
    public void setCategory(Category category) {
        this.category = category;
```

```
    }
    public boolean isChanges() {
        return changes;
    }
    public void setChange(boolean changes) {
        this.changes = changes;
    }
}
package entity;
import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
@Entity
public class BookObserver implements Observer,Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
    private String message;
     @ManyToOne
     @JoinColumn(name = "book_id")
     private Book book;
     public BookObserver(Book book) {
           this.book = book;
           this.book.attach(this);
         }
    public int getId() {
        return id;
    }

    public void setId(int id) {
```

```
        this.id = id;
    }
    public String getMessage() {
        return message;
    }
    public void setMessage(String message) {
        this.message = message;
    }
    public BookObserver() {
    }
        @Override
        public void update() {
           if(book.isChanges()){
                this.message = book.getState();
           }
        }
        @Override
        public void setBook(Book book) {
           this.book = book;
        }
    public Book getBook() {
        return book;
    }
}
package entity;
import entity.OrderDetail;
import java.util.List;
public class ShipMienBac extends Shipment{

        @Override
        public int calculateShipmentPayment() {
           int gia = 0;
           for(OrderDetail ct : books){
                gia +=ct.getQuanity()*5;
```

```
            }
            return gia;
         }
}
package entity;
import java.util.List;
public class ShipMienTrung  extends Shipment{

         @Override
         public int calculateShipmentPayment() {
            int gia = 0;
            for(OrderDetail ct : books){
                  gia +=ct.getQuanity()*10;
            }
            return gia;
         }
}
package entity;
import java.util.List;
public class ShipMienNam extends Shipment{
         @Override
         public int calculateShipmentPayment() {
            int gia = 0;
            for(OrderDetail ct : books){
                  gia +=ct.getQuanity()*15;
            }
            return gia;
         }
}

package entity;
import entity.OrderDetail;
import java.io.Serializable;
import java.util.List;
```

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
public abstract class Shipment {
    public Shipment() {
    }
    protected List<OrderDetail> books;
    public abstract int calculateShipmentPayment();
    public List<OrderDetail> getBooks() {
        return books;
    }
    public void setBooks(List<OrderDetail> books) {
        this.books = books;
    }
}
package session;
import java.util.List;
import javax.persistence.EntityManager;
public abstract class AbstractFacade<T> {
    private Class<T> entityClass;

    public AbstractFacade(Class<T> entityClass) {
        this.entityClass = entityClass;
    }
    protected abstract EntityManager getEntityManager();
    public void create(T entity) {
        getEntityManager().persist(entity);
    }
    public void edit(T entity) {
        getEntityManager().merge(entity);
    }
    public void remove(T entity) {
```

```
getEntityManager().remove(getEntityManager().merge(entity));
    }
    }

    public T find(Object id) {
        return getEntityManager().find(entityClass, id);
    }
    public List<T> findAll() {
        javax.persistence.criteria.CriteriaQuery cq =
getEntityManager().getCriteriaBuilder().createQuery();
        cq.select(cq.from(entityClass));
        return getEntityManager().createQuery(cq).getResultList();
    }
    public List<T> findRange(int[] range) {
        javax.persistence.criteria.CriteriaQuery cq =
getEntityManager().getCriteriaBuilder().createQuery();
        cq.select(cq.from(entityClass));
        javax.persistence.Query                q                =
getEntityManager().createQuery(cq);
        q.setMaxResults(range[1] - range[0] + 1);
        q.setFirstResult(range[0]);
        return q.getResultList();
    }
    public int count() {
        javax.persistence.criteria.CriteriaQuery cq =
getEntityManager().getCriteriaBuilder().createQuery();
        javax.persistence.criteria.Root<T>           rt           =
cq.from(entityClass);

cq.select(getEntityManager().getCriteriaBuilder().count(rt));
        javax.persistence.Query                q                =
getEntityManager().createQuery(cq);
        return ((Long) q.getSingleResult()).intValue();
    }}
package session;
import entity.BookObserver;
```

```
import java.util.List;
import javax.ejb.Local;
@Local
public interface BookObserverFacadeLocal {
    void create(BookObserver bookObserver);
    void edit(BookObserver bookObserver);
    void remove(BookObserver bookObserver);
    BookObserver find(Object id);
    List<BookObserver> findAll();
    List<BookObserver> findRange(int[] range);
    int count();
}
package session;
import entity.BookObserver;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
@Stateless
public class BookObserverFacade extends
AbstractFacade<BookObserver> implements BookObserverFacadeLocal {
    @PersistenceContext(unitName = "BookStorePU")
    private EntityManager em;
    @Override
    protected EntityManager getEntityManager() {
        return em;
    }
    public BookObserverFacade() {
        super(BookObserver.class);
    }
}
package session;
import entity.OrderDetail;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
```

```
import javax.persistence.PersistenceContext;

@Stateless

public class OrderDetailFacade extends AbstractFacade<OrderDetail>
implements OrderDetailFacadeLocal {

    @PersistenceContext(unitName = "BookStorePU")

    private EntityManager em;

    @Override

    protected EntityManager getEntityManager() {

        return em;

    }

    public OrderDetailFacade() {

        super(OrderDetail.class);

    }

}
```

Bạn đọc tự xây dựng Servlet và trang JSP để thực thi các chức năng của hệ thống.

TÀI LIỆU THAM KHẢO

[1] S. T. Albin, The Art of Software Architecture: Desing Methods and Techniques, John Wiley and Sons, 2003.

[2] Nguyễn Văn Ba, Phát triển hệ thống hướng đối tượng với UML 2.0 *và C++*, NXB Đại học Quốc gia Hà nội, 2005.

[3] Bass L., Clements P., Kazman R., Software Architecture in Practice, Addison-Wesley, 2013

[4] A. Dennis B. H. Wixom and David Tegarden, System Analysis and Design with UML version 2.0: An Object-Oriented Approach, Second Edition, John Wiley & Sons 2005.

[5] M. K. Debbarma et al., A Review and Analysis of Software Complexity Metrics in Structural Testing, International Journal of Computer and Communication Engineering, Vol. 2, No. 2, March 2013. Available at http://www.ijcce.org/papers/154-K271.pdf

[6] Gregor Engels, Object-Oriented Modeling: A Roadmap, http://wwwcs.uni-paderborn.de/cs/ag-engels/Papers/2000/EG00objectorientedModelling.pdf

[7] Hans-Erit, Magnus Penker, Brian Lyons, David Faado, UML2 Toolkit, Wiley Publishing, Inc, 2004

[8] Microsoft, Microsoft application architecture guide, Second Edition, 2009

[9] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design patterns: Elements of Reusable Object Oriented Software, Addition Wesley, 1994

[10] Partha Kuchana, Software architecture design patterns in Java, Auerbach Publications, 2004.

[11] Lin Liao, From Requirements to Architecture: The State of the Art in Software Architecture Design. Availble at: http://www.liaolin.com//Courses/architecture02.pdf

[12] Mike O'Docherty, Object-Oriented Analysis and Design: Understanding System Development with UML 2.0, John Wiley & Sons, 2005.

[13] R. Pressman, Software Engineering: A Practitioner's Approach, McGraw-Hill, 2005

[14] Trần Đình Quế, Phân tích và thiết kế Hệ thống thông tin, Bài giảng cho sinh viên Học viện Công nghệ Bưu chính Viễn thông, 2013

[15] S. Schach, Object-oriented and classical software engineering, Eighth Edition, McGraw-Hill, 2011.

[16] Brett Spell, Pro Java Programming, Second Edition, Apress 2006

[17] Ashish Sharma and D.S. Kushwaha, A Complexity measure based on Requirement Engineering Document, Journal of computer science and engineering, Vol.1, Issue 1, May 2010. Available at http://arxiv.org/ftp/arxiv/papers/1006/1006.2840.pdf

[18] R. Taylor, N. Medvidovic and E. Dashofy, Software Architecture: Foundations, Theory and Practice, Wiley Publisher, 2010.

[19] David P. Tegarden et al., A Software Complexity Model of Object-Oriented Systems, 1992. Available at http://www.acis.pamplin.vt.edu/faculty/tegarden/wrk-pap/DSS.PDF

[20] Joseph S. Valacich, Joey F. George, Jeffrey A. Hoffer, Essentials of systems analysis and design, Fifth Edition, Pub. Pearson, 2011.

[21] A. J. A. Wang and K. Qian. Component Oriented Programming, Wiley, 2005

[22] Java Pattern: http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html

[23] Java Pattern Tutorial: https://www.tutorialspoint.com/design_pattern/index.htm