Chương 1
GIỚI THIỆU CÁC CASE STUDY

1.1 Giới thiệu

Trong quyển sách này có sử dụng hai case study: *CarMatch* để minh hoạ và làm ví dụ thực hành còn *VolBank* dành cho ví dụ thực hành và bài tập tự làm.

1.2 CarMatch

1.2.1 Tổng quan về CarMatch

CarMatch là một *công ty hội viên* (franchise company) được thành lập với chức năng khuyến khích mọi người dùng chung xe hơi. Ở nhiều thành phố, giao thông ùn tắc đang đe doạ đến chất lượng cuộc sống cũng như gây ô nhiễm môi trường đáng kể, bao gồm cả việc thải khí CO_2 vào bầu khí quyển. Nhiều quốc gia đã đồng ý thực hiện hiệp ước quốc tế giảm bớt lượng khí thải Carbon nhằm ngăn chặn tình trạng trái đất nóng dần lên, *CarMatch* là một giải pháp cho tình trạng này. Ở một số khu vực, phương tiện giao thông công cộng không còn được sử dụng nhiều do số lượng xe hơi riêng ngày càng tăng, và cơ sở hạ tầng của hệ thống giao thông công cộng không đáp ứng được nhu cầu đi lại của những người không sử dụng xe riêng. Việc lên kế hoạch chia sẻ xe để dùng chung là một giải pháp tạm thời giúp giảm bớt lượng lưu thông mà không cần phải đầu tư lập tức các cơ sở hạ tầng cho các phương tiện giao thông công cộng.

CarMatch tìm kiếm giải pháp khuyến khích việc dùng chung xe hơi và đưa ra dịch vụ chia sẻ xe cho những người sống và làm việc gần nhau. Trong khi nhiều người làm việc chung dùng chung phương tiện, thì những người làm việc gần nhau lại khó tìm ra người thích hợp để đi chung xe. Trong vài công ty lớn, ngay cả khi cùng làm việc một nơi người ta có thể cũng không biết nhau.

Cấu trúc của *CarMatch* gồm ba lớp: lớp *hoạt động toàn cầu* mang tính phi lợi nhuận, công ty *điều hành trung tâm* ở mỗi quốc gia và lớp *hoạt động địa phương* của các hội viên. *CarMatch* trung tâm sẽ cung cấp dịch vụ cho chính quyền và các tổ chức có nghĩa vụ pháp lý làm giảm lượng

lưu thông tại các nước hoặc các bang. Nó cũng quảng cáo các dịch vụ đến với công chúng. Những người tham gia phải trả một khoảng lệ phí, gọi là *phí thành viên*, số tiền này sẽ được hoàn lại nếu *CarMatch* địa phương không thể tìm được những người có nhu cầu đi chung với họ hoặc không thể cung cấp phương tiện cho họ. *CarMatch* địa phương sẽ thảo một bản thoả thuận mẫu giữa những người tham gia để đảm bảo số tiền trao tay cho chi phí xăng được coi như thu nhập chịu thuế và khuyên họ mua bảo hiểm đặc trưng cho việc dùng chung xe. *CarMatch* địa phương sẽ đóng vai trò đại lý cho các công ty bán hợp đồng bảo hiểm.

Nhân viên của *CarMatch* địa phương sẽ học một khoá huấn luyện toàn diện về công tác tư vấn (vì họ phải giúp đỡ công ty và chính quyền địa phương, tình trạng pháp lý ở quốc gia hay bang của họ), về những yêu cầu bảo hiểm, cân nhắc sự an toàn và cách điều hành hệ thống *CarMatch*. Ở một số quốc gia, ngành bảo hiểm yêu cầu các nhân viên *CarMatch* cũng phải đáp ứng được các qui định của ngành.

CarMatch dự định sẽ thu lợi nhuận từ phí thành viên, phí tư vấn và hoa hồng bán bảo hiểm. *CarMatch* trung tâm lấy một phần lợi nhuận, *CarMatch* địa phương giữ phần còn lại. Khi hệ thống *tính giá giao thông* (road-price) dựa trên việc liên lạc bằng sóng vô tuyến giữa xe cộ và hệ thống tiếp sóng ngày càng trở nên phổ biến, *CarMatch* địa phương sẽ bán và cài đặt các thiết bị, làm việc với cơ quan chức năng thu phí cầu đường và hệ thống tính giá giao thông để thương lượng chiết khấu cho các thành viên trên cơ sở cam kết giảm nhu cầu lưu thông.

1.2.2 Hỗ trợ khách hàng

CarMatch cần có một hệ thống máy tính cho mỗi *CarMatch* địa phương, để khi có một dịch vụ mới là có sự hỗ trợ của máy tính ngay từ đầu. Mỗi quốc gia phải có ít nhất một web-server. Các web-server này cung cấp cho *CarMatch* địa phương các thông tin cập nhật thường xuyên và các dịch vụ môi giới bảo hiểm, cũng như cho phép các thành viên đăng ký trực tuyến. Thông tin về thành viên sẽ được sẽ được tải xuống hệ thống của *CarMatch* địa phương trong khu vực tương ứng. Khi không có *CarMatch* địa phương thích hợp, *CarMatch* trung tâm sẽ cố gắng đáp ứng nhu cầu cho các thành viên.

1.2.3 Các yêu cầu của CarMatch

Các yêu cầu sau đây là những yêu cầu về hệ thống mà *CarMatch* địa phương sử dụng (hệ thống trung tâm là chủ đề của một quá trình phát triển độc lập khác).

1. Phát triển một hệ thống lưu trữ thông tin về các thành viên *CarMatch*
   1.1 Lưu chi tiết của các khách hàng tiềm năng, dù họ cung cấp phương tiện hoặc tìm phương tiện để đi chung hoặc cả hai, lưu vị trí địa lý nhà họ ở và địa chỉ nơi họ làm việc.
   1.2 Chuyển thông tin chi tiết của khách hàng từ web-server nếu họ đăng ký trực tuyến.
   1.3 Cung cấp một giao diện cho hệ thống giao dịch bằng thẻ tín dụng và *hệ thống chuyển ngân tự động ABTS* (Automated Bank Transfer Sysytem) để xử lý việc trả và hoàn phí thành viên.
2. Ghép thành viên này với những thành viên khác để dùng chung xe.
   2.1 Dựa vào vị trí địa lý và thời gian đi lại để sắp xếp những người có thể dùng chung xe.
   2.2 Lưu chi tiết của những sắp xếp thành công.
3. Ghi lại việc bán bảo hiểm
   3.1 Lưu chi tiết những hợp đồng bán cho các thành viên, các xử lý gia hạn.
   3.2 Ghi nhận hoa hồng thu được từ các hợp đồng này.
4. Lưu chi tiết thông tin của khách hàng trong phạm vi hoạt động
   4.1 Bảo trì danh sách địa chỉ mail của khách hàng tiềm năng.
   4.2 Lưu chi tiết thông tin khách hàng cần tư vấn.
   4.3 Lưu lại những cuộc hẹn gặp của nhân viên (và các tiếp xúc khác) đối với khách hàng cần tư vấn.
5. Hệ thống phải có khả năng mở rộng để hợp nhất thông tin về phí cầu đường, hệ thống tính phí giao thông và các thiết bị đã bán ra và cài đặt cho các thành viên.

1.3 VolBank

1.3.1 Tổng quan về VolBank

VolBank là một tổ chức phi lợi nhuận nhằm liên kết những tình nguyện viên với những cá nhân hay các tổ chức cần sự giúp đỡ. Mục tiêu chính là nhằm tăng trách nhiệm của công dân đối với cộng đồng thông qua các hoạt động tình nguyện ở địa phương của mình. Để thực hiện được việc này cần phải lưu một danh sách thông tin về các hoạt động tình nguyện hiện có và một danh sách thông tin về những tình nguyện viên, để có thể phân công hợp lý. Phương châm của *VolBank* là làm cách nào để có thể đưa những người có kỹ năng đến với các nhu cầu công việc phù hợp nhất. Vì vậy các tình nguyện viên cần phải đăng ký các kỹ năng mình có, và người nhận giúp đỡ nêu lên yêu cầu các kỹ năng mình cần được giúp đỡ. Chẳng hạn như, Pete Duffield tình nguyện giúp việc quét sơn và

trang trí. Anh ta được gửi đến tới một khu vực địa phương sau trường học dành cho trẻ dưới mười tuổi vì ở đó đang cần sơn lại. Những đứa trẻ ở đây dành thời gian vui đùa tại một nhà dưỡng lão địa phương. Một trong những người ở đó, bà Hernandez, sẽ dành thời gian cho những ai muốn luyện tập đàm thoại tiếng Tây Ban Nha và Pete Duffield tiếp xúc với bà ta để ôn lại tiếng Tây Ban Nha trước kỳ nghỉ ở Mexico.

Tên *VolBank* mang ý nghĩa rằng người ta có thể *gởi* thời gian mà họ rỗi cho người khác, cũng như các kỹ năng mà họ có để giúp đỡ người khác. Các thông tin của *VolBank* được cung cấp rộng rãi trên các phương tiện như đài phát thanh, đài truyền hình, quảng cáo và Internet. *VolBank* cũng đồng thời kết hợp với các tổ chức tình nguyện địa phương để có thể mở rộng hệ thống mạng lưới các công tác tình nguyện và các tình nguyện viên.

Các tình nguyện viên có thể đăng ký các kỹ năng của mình với *VolBank* bằng cách gọi điện thoại đến một người tổ chức nào đó, hoặc trực tiếp thông qua một tổ chức tình nguyện địa phương hoặc điền các thông tin chi tiết của mình vào trang web. Khi đăng ký họ có thể *gởi* thời gian của mình theo cùng một cách như khi đăng ký. Nếu người tình nguyện đăng ký qua các tổ chức tình nguyện địa phương thì thông tin cũng được đưa tới một người tổ chức nào đó để lưu lại tương tự như trường hợp liên hệ trực tiếp bằng điện thoại.

Các tổ chức tình nguyện và các cá nhân (bao gồm cả các tình nguyện viên) có thể đăng ký các nhu cầu cần được giúp đỡ bằng cách liên hệ với một người thuộc tổ chức tình nguyện. Người này sẽ cố sắp xếp các tình nguyện viên phù hợp với các nhu cầu đó. Thường sẽ xảy ra hai trường hợp: một tình nguyện viên có thể đáp ứng được nhiều nhu cầu, hoặc một nhu cầu có thể có nhiều tình nguyện viên phù hợp. Việc sắp xếp được thực hiện dựa trên các thông tin về vị trí địa lý, sử dụng mã số vùng điện thoại hoặc mã bưu điện và bằng cách đối sánh các kỹ năng.

Khi tình nguyện viên được giao việc, họ được thông báo chi tiết và nếu cần, những thông tin về họ sẽ được chuyển đến tổ chức tình nguyện hay cá nhân yêu cầu được giúp đỡ. Có một điều rõ ràng là những tình nguyện viên không tự nhiên được chấp nhận. Với một số công việc, chẳng hạn như làm việc với trẻ em, sẽ có các thủ tục kiểm tra kỹ hơn thậm chí có cả sự tham gia của cảnh sát và các tổ chức xã hội. Đó là trách nhiệm của những tổ chức đề nghị giúp đỡ.

1.3.2 Hỗ trợ khách hàng

VolBank cần một hệ thống máy tính hỗ trợ việc liên kết các tình nguyện viên với các nhu cầu. Hệ thống này kết nối các web-server của *VolBank*.

Các tổ chức thành viên sẽ được thông báo mỗi khi một liên kết giữa nhu cầu và tình nguyện viên được hoàn tất. Việc thông báo này được thực hiện bằng email hoặc fax. Tình nguyện viên được thông báo bằng thư.

1.3.3 Các yêu cầu của VolBank

Các yêu cầu dưới đây dành cho hệ thống quản lý việc đăng ký, thực hiện việc liên kết và thông báo cho những người tham gia. Web-server là một hệ thống độc lập.

1. Phát triển một hệ thống quản lý việc đăng ký của các thành viên và quỹ thời gian của họ.
   1.1 Ghi nhận thông tin chi tiết của các tình nguyện viên, kể cả kỹ năng và địa chỉ của mỗi người.
   1.2 Ghi nhận thông tin về quỹ thời gian mà tình nguyện viên đăng ký.
   1.3 Chuyển từ web-server các thông tin chi tiết của tình nguyện viên cũng như thời gian đăng ký của họ.
2. Quản lý các bản ghi về các nhu cầu cần tình nguyện giúp đỡ
   2.1 Lưu chi tiết thành viên của các tổ chức tình nguyện
   2.2 Lưu các nhu cầu cần giúp đỡ của các tổ chức tình nguyện
   2.3 Lưu các nhu cầu cần giúp đỡ của các cá nhân
3. Kết hợp các tình nguyên viên với các nhu cầu và ghi nhận lại kết quả
   3.1 Kết hợp một tình nguyện viên với các hoạt động tình nguyện thích hợp.
   3.2 Kết hợp một hoạt động tình nguyện với các tình nguyện viên thích hợp trong cùng một khu vực.
   3.3 Lưu các kết hợp giữa tình nguyện viên và hoạt động tình nguyện.
   3.4 Thông báo cho tình nguyện viên biết các kết hợp đó.
   3.5 Thông báo cho các tổ chức tình nguyện biết các kết hợp đó.
   3.6 Ghi nhận thành công của mỗi kết hợp và lập ra một bản cam kết cho từng kết hợp đạt được.
4. Lập các báo cáo thống kê về số lượng tình nguyện viên và nhu cầu, và tổng quỹ thời gian mà tình nguyện viên bỏ ra cũng như tổng thời gian cần giúp đỡ.

Chương 2
CƠ BẢN VỀ UML

2.1 Giới thiệu

Ngôn ngữ mô hình hợp nhất *UML* (Unified Modeling Language) là một *ngôn ngữ trực quan* cung cấp cho các nhà phân tích thiết kế các hệ thống hướng đối tượng một cách *hình dung ra* các hệ thống phần mềm, *mô hình hoá* các tổ chức nghiệp vụ sử dụng các hệ thống phần mềm này; cũng như *xây dựng* chúng và *làm tài liệu* về chúng. Công ty phần mềm *Rational* và *OMG* (Object Management Group) đã cùng nhau đưa ra ba biểu đồ các ký hiệu hướng đối tượng có ý nghĩa, kết hợp với các khía cạnh của nhiều ký hiệu khác, tạo ra một ngôn ngữ mô hình chuẩn nhằm biểu diễn cách thực hành tốt nhất trong ngành công nghiệp phát triển phần mềm. UML vẫn đang tiến triển như là một chuẩn, và trở thành một chuẩn quốc tế được *tổ chức tiêu chuẩn quốc tế ISO* (International Standard Organization) chấp nhận.

Chương này sẽ giải thích lịch sử của UML, mô tả hiện trạng và hướng phát triển trong tương lai của nó. Ngoài ra, chương này cũng giải thích cấu trúc của UML và cách làm tài liệu về UML.

2.2 Nguồn gốc của UML

Kỹ thuật phát triển phần mềm hướng đối tượng đã trải qua 3 giai đoạn:

1. Các ngôn ngữ lập trình hướng đối tượng được phát triển và bắt đầu được sử dụng.
2. Các kỹ thuật phân tích và thiết kế hướng đối tượng được tạo ra nhằm giúp đỡ công việc mô hình hoá nghiệp vụ, phân tích các yêu cầu và thiết kế các hệ thống phần mềm. Những kỹ thuật này ngày càng được phát triển rộng rãi.
3. UML được thiết kế nhằm kết hợp các đặc điểm tốt nhất của một số các kỹ thuật và ký hiệu trong phân tích thiết kế để tạo ra một tiêu chuẩn công nghiệp.

2.2.1 Các ngôn ngữ lập trình

Simula-67 được xem như là ngôn ngữ hướng đối tượng đầu tiên. *Simula 1*, được phát triển đầu tiên vào những năm đầu của thập niên 1960, là

ngôn ngữ dùng để mô phỏng các biến cố rời rạc. Một hệ thống mô phỏng được sử dụng để phân tích và tiên đoán các hành vi của một hệ thống vật lý, chẳng hạn như một hệ thống giao thông, một phản ứng hoá học hay một dây chuyền lắp ráp. Việc mô phỏng các biến cố rời rạc sẽ mô phỏng hệ thống thực dưới dạng các trạng thái rời rạc nhằm đáp ứng các biến cố xảy ra vào một thời điểm cụ thể. Đây là sự khác biệt giữa mô phỏng rời rạc và mô phỏng liên tục (trạng thái đang tiến triển liên tục). Mô hình hoá một giao lộ là một mô phỏng biến cố rời rạc: phương tiện giao thông đến và đèn giao thông đổi trạng thái vào các thời điểm xác định. Một phản ứng hoá học thường được mô hình như là một mô phỏng liên tục: các hoá chất phản ứng với nhau liên tục và tốc độ của phản ứng phụ thuộc vào sự thay đổi của các yếu tố như nhiệt độ và áp suất.

Simula-67 được *Kristen Nygaard* và *Ole-Johan Dahl* thuộc đại học *Oslo* và trung tâm máy tính *Morwegian* phát triển vào năm 1967. Nó được xây dựng dựa trên *Simula 1* và là một ngôn ngữ lập trình phổ thông. Năm 1986, ngôn ngữ này được biết đến như là ngôn ngữ *Simula* và được gọi như thế cho đến ngày nay. *Simula* giới thiệu nhiều đặc tính của ngôn ngữ lập trình hướng đối tượng, chẳng hạn như *lớp* (class) và sự *thừa kế* (inheritance).

Ngôn ngữ lập trình hướng đối tượng tường minh đầu tiên là *Smalltalk*, được phát triển bởi *Alan Kay* ở trường đại học *Utah* và sau đó là *Adele Goldberg* và *Daniel Ingalls* ở *Serox PARC* (trung tâm nghiên cứu Palo Alto) vào những năm 1970. Bản phát hành *Smalltalk80* được dùng phổ biến rộng rãi vào thập niên 1980. *Smalltalk* giới thiệu ý tưởng về cách giao tiếp giữa các đối tượng qua cách truyền thông điệp và về các thuộc tính được đóng gói bên trong các đối tượng. Các thuộc tính này có thể được truy cập từ các đối tượng khác bằng cách truyền thông điệp.

Sau *Smalltalk*, nhiều ngôn ngữ lập trình hướng đối tượng ra đời như: *Objective C*, *C++*, *Eiffel* và *CLOS* (Common Lisp Object System). Kể từ phiên bản năm 1996, *Java* đã tạo được sự quan tâm lớn đối với sự phát triển hướng đối tượng. Gần đây Microsoft đưa ra ngôn ngữ *C#* (C-sharp) xem như là sự kết hợp tốt nhất của Java và C++. Giữa *Simula* và *C#*, nhiều ngôn ngữ trình hướng đối tượng được phát triển đã được phát triển và tiếp tục được phát triển, nhưng, cùng với sự phát triển của Internet, *Java* đã làm cho việc phát triển hướng đối tượng trở nên phổ biến hơn.

2.2.2 Phân tích và thiết kế

Sau khi *Smalltalk* xuất hiện vài năm, các sách về phân tích và thiết kế hướng đối tượng bắt đầu xuất hiện. Một số sách gắn liền với một ngôn ngữ cụ thể (chẳng hạn *Objective C* và *C++*), số còn lại có mục đích tổng

quát hơn. Trong số đó, chúng ta phải kể công trình của *Shaler & Melllor* (1988) và *Coud & Yourdon* (1990 & 1991). Phát triển sau là *Booch* (1991), nhóm của *Rumbaugh, Blaha, Premerlani, Eddy & Lorencesen* (1991), và không lâu sau đó là *Jacobson, Christerson Jonsson & Overgaard* (1992). Còn nhiều sách khác nhưng các cuốn sách của các tác giả nói trên được biết và được sử dụng hầu khắp.

Các tác giả khác nhau đã dùng các ký hiệu khác nhau để biểu diễn *lớp*, *đối tượng* và *mối quan hệ* giữa chúng. Thường thì họ lại dùng một thành phần ký hiệu giống nhau để biểu diễn cho những thứ khác nhau. Ví dụ, *Coad và Yourdon* dùng hình tam giác để biểu diễn cho quan hệ kết hợp *whole-part* (xem phần phụ lục), trong khi đó *Rumbaugh* và các cộng sự của mình lại dùng hình tam giác để biểu diễn sự thừa kế. Họ cũng cung cấp một phương pháp phân tích thiết kế, bao gồm các các giai đoạn tiến hành, các công việc phải làm và các đặc tả cần thiết. Chúng đều được định nghĩa rõ ràng, nhiều hay ít.

Đầu thập kỷ 1990 được đặc trưng bởi sự phát triển đa dạng của các ký hiệu và phương pháp, một số tác giả gọi đây là cuộc *"chiến tranh phương pháp"* (Method War). Từ 1989 đến 1994, ngôn ngữ mô hình từ mức chưa tới 10 ngôn ngữ đã phát triển được hơn 50 ngôn ngữ. Khoảng giữa thập niên 1990, tình hình đã thay đổi. Các phương pháp của ba tác giả chính, *Rumbaugh Booch* và *Jacobson*, đã nổi bật hẳn lên. *Rumbaugh* sửa đổi công trình kỹ thuật mô hình hướng đối tượng *OMT* (Object Modelling Technique) thành OMT-2. *Booch* phát hành ấn bản thứ hai của mình có tên là *Booch'93*, các phương pháp của *Jacobson* được biết đến dưới tựa đề Công Nghệ Phần Mềm Hướng Đối Tượng *OOSE* (Object-Oriented Software Engineering) hay *Objectory*, tên công ty của *Jacobson*.

2.2.3 Sự xuất hiện của UML

Cả ba phương pháp của ba người cũng bắt đầu trở nên tương tự nhau vì họ đã kết hợp các đặc tính tốt nhất của các phương pháp khác. Vào năm 1994, *Rumbaugh* và *Booch* đã kết hợp làm việc chung với nhau trong công ty phần mềm *Rational* để thống nhất hai phương pháp của họ. Tháng 10 năm 1995, họ đã đưa ra bản phác thảo *phương pháp hợp nhất* (Unified Method) phiên bản 0.8. Vào mùa thu năm 1995, *Jacobson* cùng với công ty của mình đã gia nhập vào *Rational*, và cả ba bắt đầu phát triển cả *UML* cũng như *qui trình phát triển phần mềm hợp nhất USDP* (Unified Software Development Process), phần lớn dựa trên phương pháp *Objectory*.

Vào tháng 6 và tháng 10 năm 1996, phiên bản 0.9 và 0.91 đã được phát hành và nhận được sự phản hồi từ các tổ chức quan tâm đến việc phát

triển một ngôn ngữ mô hình hướng đối tượng chuẩn. Vào thời điểm này OMG đã đưa ra một *Yêu Cầu Hợp Nhất RFP* (Request for Proposal) cho một ngôn ngữ mô hình hướng đối tượng chuẩn. *Rational* đã nhận thấy rằng có nhu cầu liên đới giữa tiến trình hợp nhất với sự hình thành mối liên kết giữa các thành viên UML với các tổ chức như IBM, HP, Microsoft và Oracle – những tổ chức sẵn sàng cung cấp tài nguyên cho sự phát triển xa hơn của UML như là một phản hồi đến OMG.

Tháng 1 năm 1997 các thành viên UML và một số tổ chức khác đã đệ trình các đề nghị đến OMG. Sau đó họ cùng nhau đưa ra UML phiên bản 1.1. Vào tháng 11 năm 1997, UML phiên bản 1.1 nằm trong danh sách các kỹ thuật được chấp nhận của OMG, và OMG chịu trách nhiệm về tương lai của UML.

OMG đã thành lập Nhóm Xét Duyệt *RTF* (Revision Task Force) do *Cris Kobryn* của *MCI Systemhouse* phụ trách, RTF chịu trách nhiệm cải tiến UML – xử lý lỗi lập trình, điều chỉnh các sai sót, giải quyết các mâu thuẫn và các khái niệm còn nhập nhằng. Tháng 6 năm 1998, RTF đưa ra một phiên bản sửa đổi (phiên bản 1.2) và tháng 6 năm 1999, đưa ra phiên bản hoàn chỉnh (phiên bản 1.3).

2.3 UML ngày nay

OMG thành lập ra RTF để phát triển UML. Kế hoạch phát triển UML trong tương lai được giải thích trong mục 2.5. Phiên bản 1.3 được sưu liệu trong *đặc tả UML* (UML Specification – Object Management Group, 1999a). Nội dung bao gồm

Chương 1	Tóm tắt về UML
Chương 2	Ngữ nghĩa UML
Chương 3	Hướng dẫn ký hiệu UML
Chương 4	Sơ lược chuẩn UML
Chương 5	Định nghĩa giao diện UML CORBA
Chương 6	Đặc tả UML XMI DTD
Chương 7	Đặc tả ngôn ngữ ràng buộc đối tượng
Phụ lục A	Các thành phần chuẩn của UML
Phụ lục B	Chú thích thuật ngữ mô hình UML

Bảng 2.1: Nội dung của đặc tả UML

Đặc tả UML không phải là tài liệu được viết cho người dùng bình thường, nó được viết cho các đối tượng sau đây tham khảo: các thành viên của

OMG, các tổ chức, các công ty tạo ra các công cụ CASE, các tác giả của các cuốn sách và những người làm công tác đào tạo – nói chung đây là những người muốn tìm hiểu chi tiết về UML. Đặc biệt, chương 5 và chương 6 được viết cho những người xây dựng các cộng cụ CASE. *Kiến trúc môi giới yêu cầu đối tượng chung CORBA* (Common Object Request Broker Architecture) dùng *ngôn ngữ định nghĩa giao diện IDL* (Interface Definition Language) để xác định *nội dung kho dữ liệu* phù hợp với việc *khởi tạo, lưu trữ và thao tác* trên các mô hình UML. *Công cụ đặc tả DTD* (khai báo kiểu tài liệu – Document Type Declaration) *trong XMI* (bộ chuyển đổi siêu dữ liệu XML – XML Metadata Interchange) dùng *ngôn ngữ đánh dấu mở rộng XML* (eXtensible Markup Language) cung cấp một đặc tả về cách thức mà *dữ liệu về các mô hình UML* có thể được *chuyển đổi giữa các ứng dụng* như thế nào. Chương này sẽ lập tài liệu cho hai phác thảo chuẩn UML (các cách áp dụng UML cho các dự án phát triển khác nhau): một cho qui trình phát triển phần mềm và một cho việc mô hình hoá nghiệp vụ.

2.4 UML là gì?

UML là ngôn ngữ trực quan được dùng trong qui trình phát triển các hệ thống phần mềm. Nó là một *ngôn ngữ đặc tả hình thức* (formal specification language). Chúng ta cần chú ý đến thuật ngữ *"ngôn ngữ"*. Ngôn ngữ ở đây không phải là ngôn ngữ giống với ngôn ngữ tự nhiên của con người hay ngôn ngữ lập trình. Tuy nhiên nó cũng có một tập các qui luật xác định cách sử dụng.

Các ngôn ngữ lập trình có một tập các *phần tử* và một tập các *qui luật* cho phép chúng ta tổ hợp các phần tử lại với nhau để tạo ra các chương trình hợp lệ. Các ngôn ngữ đặc tả hình thức giống như UML cũng có một tập các phần tử và một tập các qui luật riêng. Với UML, hầu hết các phần tử của nó là các đối tượng đồ hoạ như đường thẳng, hình chữ nhật, hình oval,... Chúng thường được đặt nhãn để cung cấp thêm thông tin. Tuy nhiên các phần tử đồ hoạ của UML chỉ biểu diễn các phần cần mô hình dưới dạng hình ảnh. Ta cũng có thể tạo ra một mô hình UML dưới dạng thuần dữ liệu (giống như CORBA và XMI DTD làm). Tuy nhiên cách biểu diễn bằng hình ảnh vẫn giúp mô hình dễ hiểu và trực quan hơn.

Các qui luật trong UML được mô tả trong *đặc tả UML*. Có 3 loại qui luật: *cú pháp trừu tượng, luật well-formedness* (luật hình thức) *và ngữ nghĩa*. Trong đó cú pháp trừu tượng được biểu diễn như các biểu đồ và ngôn ngữ tự nhiên, *luật well-formedness* nằm trong *ngôn ngữ ràng buộc đối tượng OCL* (Object Constraint Language). Luật được biểu diễn như biểu đồ sẽ

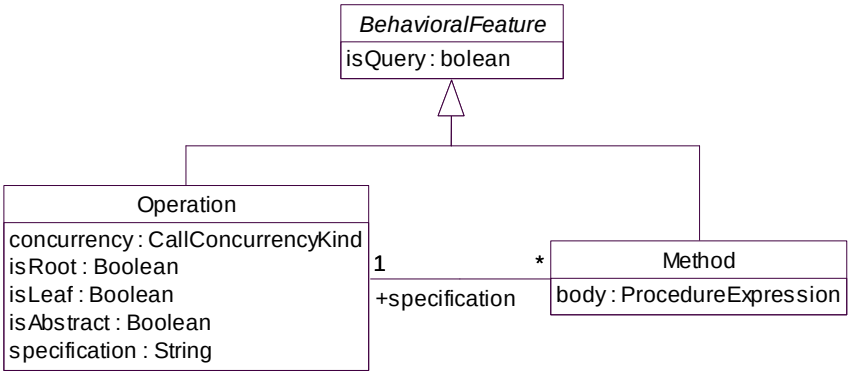
dùng một tập ký hiệu con của UML để xác định cách kết hợp giữa các phần tử. Đây là một đặc điểm quan trọng của UML, nhưng chúng ta không cần phải tìm hiểu chi tiết. Đặc điểm này được gọi là *kiến trúc siêu mô hình 4 tầng* (Four-Layer Metamodel Architecture) của UML.

2.4.1 Kiến trúc siêu mô hình 4 tầng

Xét UML qua 4 lớp. Mỗi lớp là một trừu tượng của lớp bên dưới và được định nghĩa bằng thuật ngữ của lớp nằm trên. Lớp thấp nhất chứa các *đối tượng người dùng*. Chúng là các thể hiện đối tượng trong hệ thống như *<Insurance_Policy_2123434>* (hợp đồng bảo hiểm số 2123434), *21.34* (số thực 21.34), *set_premium* (phí bảo hiểm) hay *<Insurance_Quote_Server_213>* (dịch vụ bảo hiểm 213). Lớp kế chứa các *mô hình*. Chúng là các khái niệm mô hình dùng để định nghĩa các đối tượng người dùng trong một lĩnh vực mô hình cụ thể như *InsurancePolicy* (hợp đồng bảo hiểm), *monthlyPremium* (phí bảo hiểm hàng tháng), *setPremium* (đặt phí bảo hiểm) hay *InsuranceQuoteServer* (dịch vụ bảo hiểm). Hầu hết công việc phân tích và thiết kế hệ thống phụ thuộc vào việc xác định các phần tử của mô hình là gì. Lớp thứ ba là lớp *siêu mô hình* (metamodel). *Metamodel* dùng để định nghĩa các phần tử của mô hình, là *mô hình mô tả mô hình*. Các phần tử của *metamodel* là các thành phần của UML như *lớp* (class), *thuộc tính* (attribute), *thao tác* (operation), *thành phần* (component). Lớp trên cùng là *meta-metamodel* của *OMF* (OMG Meta Object Facility), xác định ngôn ngữ định nghĩa metamodel. Các phần tử của nó là *MetaClass, MetaAttribute và MetaOperation*. Giống như metamodel, được áp dụng để đưa ra các mô hình của nhiều lĩnh vực khác nhau (như điều khiển không lưu, ngân hàng, tình nguyện viên, thư viện, dùng chung xe hơi, rô bô, viễn thông...), meta-metamodel cũng có thể được dùng để xác định nhiều metamodel khác nhau nếu chúng ta định nghĩa các ngôn ngữ đặc tả trực quan khác. *Để hiểu cú pháp trừu tượng của UML, ta phải nắm rõ tầng metamodel.*

2.4.2 Cú pháp trừu tượng

Cú pháp trừu tượng của UML được xác định bằng cách dùng ký hiệu của metamodel. Ký hiệu này là một tập con của ký hiệu UML, chúng ta dùng *biểu đồ lớp* (class diagram) để xác định các phần tử của các mô hình UML và mối quan hệ giữa chúng. Hình 2.1 biểu diễn một phần của biểu đồ này biểu diễn cú pháp trừu tượng của *gói nòng cốt* (UML core package).



Hình 2.1: Một phần cú pháp trừu tượng của gói nòng cốt

Biểu đồ này cho thấy *Operation* (thao tác) và *Method* (phương thức) là các metaclass và là lớp con của metaclass trừu tượng *BehavioralFeature*, ngoài ra *Operation* là một *specification* (đặc tả) của *Method*. Trong khi *specification* là một trong số các thuộc tính của *Operation,* thì *body* là thuộc tính duy nhất của *Method*. Thuộc tính *specification* xác định *chữ ký* (signature) của thao tác (gồm giá trị trả về, tên và các tham số) còn thuộc tính *body* là một thủ tục xác định phương thức thực hiện thao tác. Một thể hiện nào đó của một thao tác là một thể hiện của *Operation*

Trong đặc tả UML, bằng ngôn ngữ tự nhiên cũng có thể đặc tả phương thức, các thuộc tính và các quan hệ kết hợp của nó như sau:

Phương thức (method)

Phương thức là cài đặt của thao tác. Nó xác định giải thuật hoặc thủ tục ảnh hưởng đến kết quả của thao tác. Trong metamodel, phương thức là mô tả của *hành vi được đặt tên* trong một *Classifier* (như class, actor, use case, data type, component,... – xem phụ lục) và thực hiện một (trong trường hợp trực tiếp) hoặc một tập (trong trường hợp gián tiếp) các thao tác của *Classifier*.

Thuộc tính

body	Cài đặt phương thức

Quan hệ kết hợp

Specification	Chỉ rõ thao tác mà phương thức cài đặt. Thao tác phải thuộc *Classifier* (hoặc hậu duệ) sở hữu phương thức.

Đặc tả các metaclass khác trong metamodel được làm tương tự.

2.4.3 Luật well-formedness

Luật *well-formedness* áp dụng cho các thể hiện của metaclass. Chúng cung cấp các luật mà các thể hiện của metaclass phải tuân theo như một tập các bất biến *invariant*. Invariant là các ràng buộc không được phá vỡ, chúng phải luôn được thoả để mô hình có ý nghĩa. Các invariant được ghi rõ trong OCL. Ví dụ, đặc tả cho các trạng thái của metaclass *Class* là *nếu lớp là cụ thể (không trừu tượng), thì mọi thao tác của nó nên có một phương thức thực hiện*. Ràng buộc này được viết trong OCL như sau:

> ***not*** *self.isAbstract* ***implies*** *self.allOperation ® forAll(op | self.allMethods ® exists (m | m.specification ® includes(op)))*

Các ràng buộc này trên metamodel có thể được áp dụng để kiểm tra mô hình phải tuân theo các luật của UML.

Lưu ý OCL được sử dụng trong UML để làm tài liệu cho các ràng buộc. Ví dụ, nếu có một luật của hệ thống *CarMatch* được phát biểu như sau: *mọi lộ trình (journey) được liên kết đến một bản thoả thuận cũng phải thuộc về một người chia sẻ xe khác*, phát biểu này có thể được làm tài liệu trong OCL như sau:

> **context** *SharingAgreement* **inv**:
> *self.IsSharedIn ® forAll*(*i*, *j* | (*i.makes = j.makes*)
> **implies** *i = j*)

Nghĩa là, trong ngữ cảnh của lớp *SharingAgreement* (*IsSharedIn* là liên kết giữa bản thoả thuận với các lộ trình, còn *makes* là liên kết giữa lộ trình với người chia sẻ xe), *nếu hai lộ trình i và j cùng liên kết với cùng một thành viên thì hai lộ trình này là một*. Nói cách khác, hai lộ trình khác nhau cùng thuộc về một thành viên thì không được chia sẻ trong cùng một bản thoả thuận.

2.4.4 Ngữ nghĩa

Ngữ nghĩa của UML được sưu liệu bằng tiếng Anh. Ví dụ sau đây là mô tả của metaclass *Operation*

> *Operation* là một cấu trúc khái niệm, trong khi đó *Method* là một cấu trúc cài đặt. Các đặc trưng thông thường của chúng được mô tả trong *BehavioralFeature*, và xác định ngữ nghĩa của *Operation*. Cấu trúc *Method* được định nghĩa trong lớp con tương ứng của *BehavioralFeature*.

Tuy nhiên, hầu hết các thông tin về các thao tác được tìm thấy trong định nghĩa của *Class*, và bạn cần đọc toàn bộ để hiểu được ngữ nghĩa một cách đầy đủ.

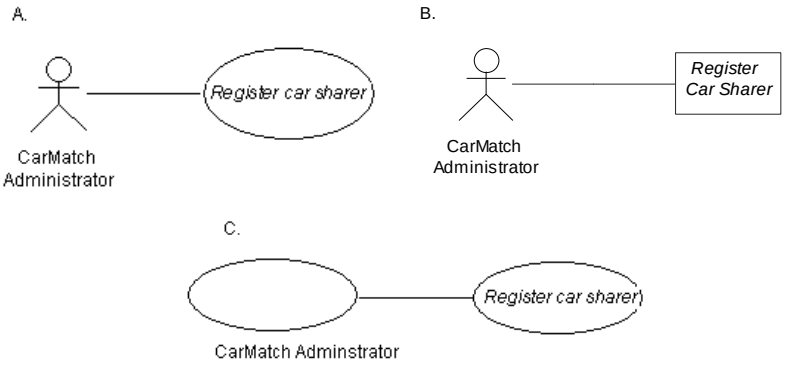
2.4.5 Hướng dẫn ký hiệu

Đây là phần lớn nhất của đặc tả UML và là phần hữu ích nhất cho những ai có ý định áp dụng UML. Trong khi cú pháp trừu tượng được mô tả bằng ký hiệu của metamodel, các luật well-formedness được mô tả bằng OCL, thì ký hiệu của UML được mô tả bằng tiếng Anh với sự hỗ trợ của các biểu đồ. Tuy nhiên, do sử dụng tiếng Anh, ký hiệu vẫn còn mơ hồ ở một vài chỗ, vì vậy đôi khi chúng ta cần phải tham khảo thêm các phần khác của đặc tả.

Phần này bao gồm một mục nói về các thành phần chung của ký hiệu UML. Trong đó sẽ giải thích các biểu đồ bao gồm cả ngữ nghĩa cùng các ký hiệu, ánh xạ giữa các ký hiệu và metamodel, các ví dụ sử dụng, những nguyên tắc đặc trưng và các tuỳ chọn biểu diễn. Các tuỳ chọn biểu diễn là các cách biểu diễn dùng các khía cạnh khác của ký hiệu vốn không phải là một phần ký hiệu, chẳng hạn như việc sử dụng màu sắc để phân biệt các loại thông điệp khác nhau.

Ví dụ 2.1

Một biểu đồ use case được tạo ra từ các hình oval (biểu diễn *use case*) và hình người (biểu diễn *user*). Chúng được liên kết với nhau bằng các đường thẳng để chỉ rõ user nào dùng use case nào. Trong các biểu đồ sau, biểu đồ nào là hợp lệ?



Trả lời:

Chỉ có A là hợp lệ còn B và C thì không (B biểu diễn use case bằng hình chữ nhật và C biểu diễn user bằng hình oval).

Ví dụ 2.2

Sau đây là hai qui luật của *metalanguage* được dùng để định nghĩa UML. Tên của các *stereotype* (khuôn dạng – xem phụ lục) nằm giữa << và >> và được bắt đầu bằng chữ thường (ví dụ <<*type*>>). Ký tự đầu tiên của các

từ, danh từ hoặc tính từ, nối thêm vào được viết hoa (ví dụ <<*ownedElement*>>, <<*allContents*>>).

Các stereotype nào sau đây tuân theo qui luật cú pháp trên?

<<*Type*>> <<*new type*>> <<*newType*>> <<*longGreenDottedLine*>>

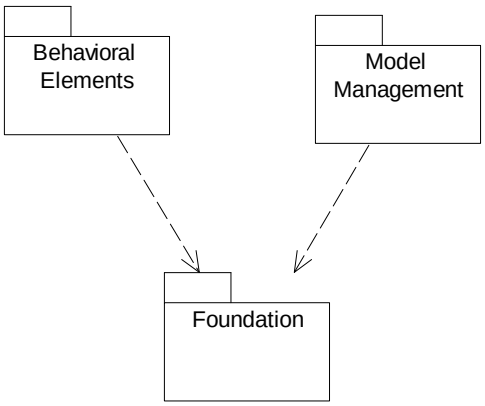
Trả lời:

<<*Type*>> không thoả bởi vì nó bắt đầu bằng ký tự hoa, <<*new type*>> cũng không thoả bởi nó chứa khoảng trắng ở giữa. Hai stereotype còn lại thoả cả hai qui luật.

2.4.6 Quản lý mô hình (model management)

2.4.6.1 Package

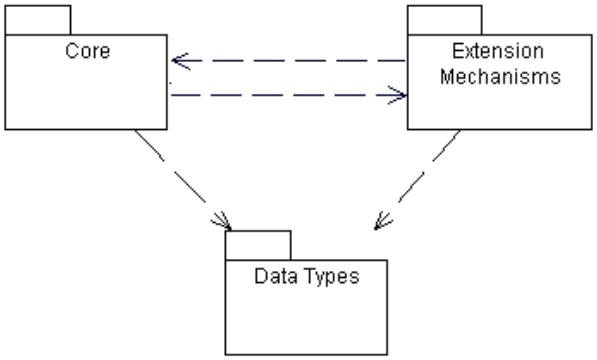
UML được tổ chức thành các *package* (gói). Mỗi package chứa một số biểu đồ tạo nên UML. Các metaclass được nhóm vào các package tuỳ vào mức độ liên kết giữa chúng. Hình 2.2 vẽ các package ở mức cao nhất và các phụ thuộc giữa chúng.



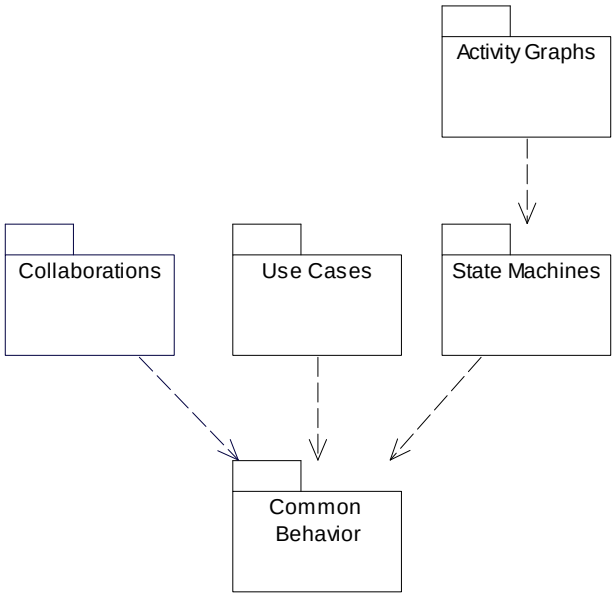
Hình 2.2: Các package *mức cao nhất của UML*

Package *Model Management* (gói quản lý mô hình) chứa các metaclass để tổ chức các mô hình thành các package. Các package được dùng trong dự án để tổ chức các biểu đồ khác nhau thành các nhóm chặt chẽ. Những package như thế thuận lợi về mặt tổ chức và không cần phải khớp với các hệ thống con trong một hệ thống hoàn chỉnh.

Hai package còn lại *Foundation* (gói cơ sở) và *Behaviour Element* (gói phần tử hành vi) được tách thành những package con như trong hình 2.3 và 2.4.

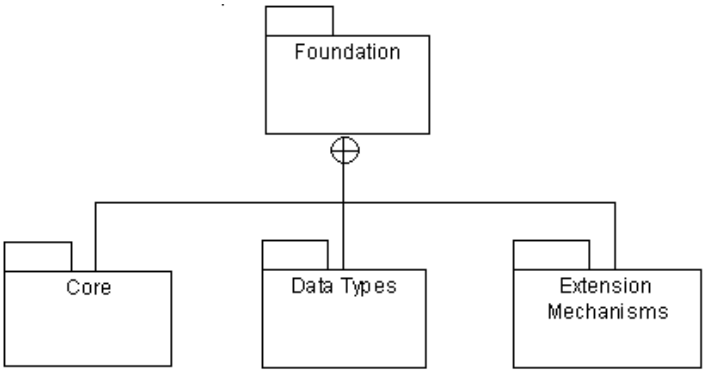


Hình 2.3: Các package Foundation *của UML*

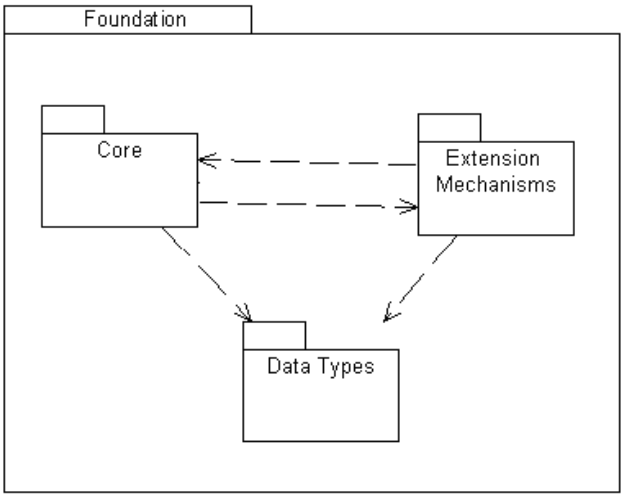


Hình 2.4: Các package Behaviour Element *của UML*

Hình 2.5 minh họa một package chứa các package khác, dùng cấu trúc cây với một dấu cộng trong một vòng tròn được vẽ ngay cuối đường thẳng gắn với đối tượng chứa. Kiểu chứa này có thể được biểu diễn bằng cách đặt các package con bên trong package cha như trong hình 2.6, trong trường hợp này tên của package cha được hiển thị trong *thẻ* thay vì được hiển thị trong thân.



Hình 2.5: Ký hiệu dạng cây

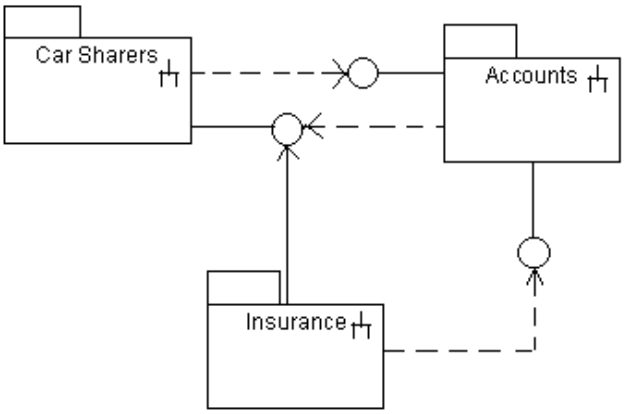


Hình 2.6: Ký hiệu khác

Quan hệ giữa các package được stereotype như <<*import*>> hoặc <<*access*>>. Như đã nói, package cung cấp cơ chế tổ chức các *phần tử mô hình*, ta có thể dùng package biểu diễn các *view* (khung nhìn) khác nhau của dự án, bao gồm *Use Case View* (khung nhìn use case), *Logical View* (khung nhìn logic) và *Component View* (khung nhìn thành phần).

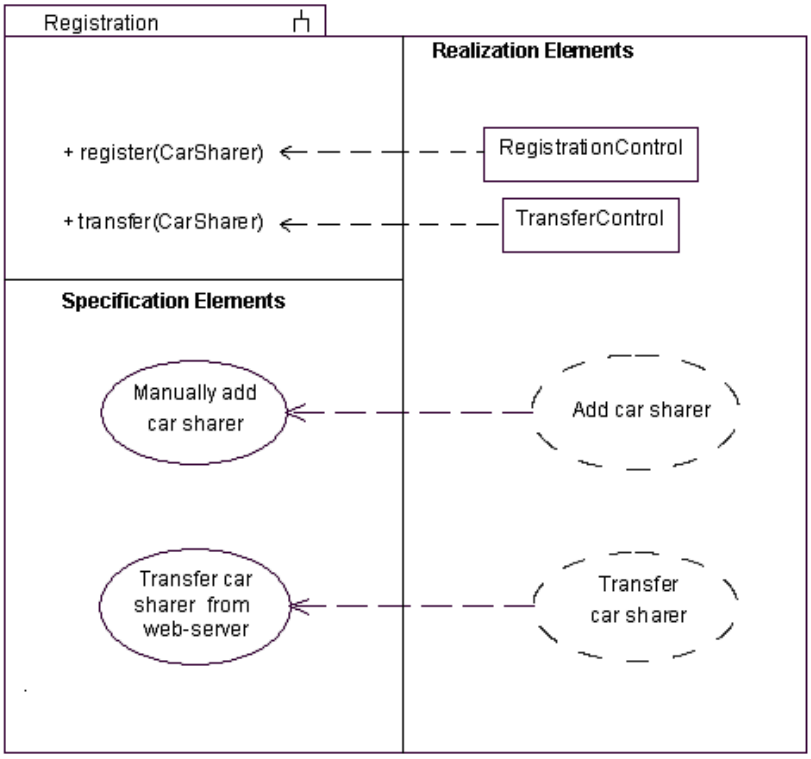
2.4.6.2 Hệ thống con (subsystem)

Các hệ thống con biểu diễn các đơn vị của hệ thống vật lý, chúng có thể được tổ chức trong các package, như trong hình 2.7. Chúng được stereotype bằng biểu tượng hình *chạc cây* hoặc <<*subsystem*>>. Trong hình 2.7 ta còn quan sát thấy các quan hệ phụ thuộc giữa các package.



Hình 2.7: Các hệ thống con CarMatch

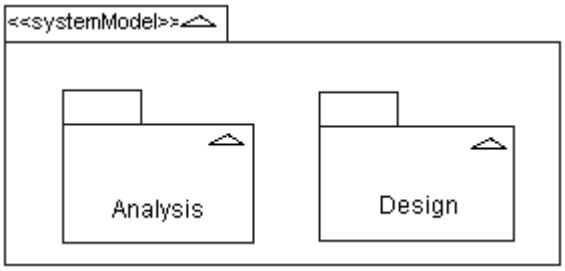
Biểu tượng hệ thống con có thể được phóng lớn và được chia thành ba phần: *Operations* mô tả các thao tác, *Specification Elements* mô tả các phần tử đặc tả và *Realization Elements* mô tả các phần tử hiện thực. Số phần đủ hay thiếu là hoàn toàn linh động. Hình 2.8 minh họa cách dùng.



Hình 2.8: Hệ thống con với các thành phần

2.4.6.3 Mô hình (model)

Mô hình là sự trừu tượng của một hệ thống vật lý với một mục đích nào đó. Các mô hình tiêu biểu được sử dụng cho các giai đoạn khác nhau của qui trình phát triển hệ thống. Mô hình mức cao nhất của một hệ thống có thể được stereotype là *<<systemModel>>* và chứa các mô hình khác, như trong hình 2.9. Có thể thêm vào mô hình một hình tam giác nhỏ (làm biểu tượng) để phân biệt với package và hệ thống con. Cũng có thể dùng stereotype *<<model>>*.



Hình 2.9: Mô hình hệ thống chứa các mô hình khác

2.4.7 Cái gì không phải là UML

Đến đây chúng ta đã biết rằng UML là một ngôn ngữ trực quan và chúng ta cũng đã khảo sát cấu trúc của nó. Bây giờ chúng ta sẽ tiến hành khảo sát các quan điểm không đúng về UML.

UML không phải là một ngôn ngữ lập trình, nghĩa là bạn không thể dùng UML để viết chương trình. Một số công cụ CASE có thể dùng mô hình UML để phát sinh ra mã nguồn dưới dạng một số ngôn ngữ, nhưng rất ít. Thường thì người phát triển vẫn phải viết mã để cài đặt các phương thức.

UML không phải là một công cụ CASE. Có rất nhiều CASE cài đặt chuẩn UML đến một qui mô nào đó rộng hơn hoặc hẹp hơn. Một phần đặc tả UML được viết cho các nhà phát triển CASE để cài đặt chuẩn này và để hoán đổi các mô hình giữa các ứng dụng sử dụng XMI DTD.

UML không phải là một phương pháp, phương pháp luận hay qui trình phát triển phần mềm. Ký hiệu UML được áp dụng trong các dự án nhằm sử dụng những hướng tiếp cận khác nhau cho qui trình phát triển phần mềm và nhằm tách chu kỳ phát triển của hệ thống thành những các hoạt động, các tác vụ, các giai đoạn và các bước khác nhau. Một điều đã xảy ra là, từ khi phát hành chuẩn UML, có một số phương pháp phân tích và thiết kế đã chuẩn hoá các ký hiệu của chúng bằng cách dùng UML. Quan điểm của các tác giả về qui trình nào là đúng, để áp dụng

phát triển các hệ thống, vẫn chưa ngã ngũ, nhưng ít nhất họ đã có cùng quan điểm về cách biểu diễn các mô hình hệ thống bằng ký hiệu trực quan. Một kết hợp chặt chẽ với UML là *USDP* (Unified Software Development Process). Trong các chương sau, chúng ta sẽ giải thích những nơi nào kỹ thuật mô hình hóa phù hợp với chu trình sống của USDP.

2.5 Tương lai của UML

RTF, thuộc tổ chức OMG, chịu trách nhiệm phát triển UML. Kế hoạch phát triển được trình bày trong bài báo *Communications of the ACM* của tác giả *Cirs Kobryn*, chủ tịch RTF. Ngoài ra thông tin về sự phát triển trong tương lai của UML cũng được đăng trên web site của RTF.

Phiên bản hiện tại, năm 2003, là 2.0. RTF đã làm việc với phiên bản 1.4, là bản sửa chữa của bản 1.3, dựa trên thông tin phản hồi từ người dùng. Người dùng đưa ý kiến của họ lên web site của RTF, RTF sẽ tham khảo các thông tin này.

Những yêu cầu cho phiên bản 2.0 được đưa ra vào tháng 9 năm 2000,và dự định tung ra vào 2001. Đây là sự sửa đổi quan trọng và sẽ bao gồm những thay đổi quan trọng cho đặc tả. Những phần đáng quan tâm trong phiên bản 2.0 là

- **Kiến trúc** – Thêm đặc tả nghiêm ngặt cho metamodel, tách biệt rõ ràng hơn cái gì là bản chất của UML và cái gì được định nghĩa trong những *profile* (xem mục sau) chuẩn.
- **Khả năng mở rộng** – Cải thiện cơ chế mở rộng UML cho những lĩnh vực riêng.
- **Các thành phần** – Hỗ trợ tốt hơn cho việc phát triển *dựa trên thành phần* thông qua ký hiệu và ngữ nghĩa của biểu đồ thành phần.
- **Mối kết hợp** – Định nghĩa tốt hơn về ngữ nghĩa của quan hệ kết hợp, bao gồm phụ thuộc *refine* (tinh chế) và *trace* (theo vết).
- **Các biểu đồ trạng thái và hành động** – Tách biệt ngữ nghĩa của 2 loại biểu đồ sao cho biểu đồ hành động có thể được định nghĩa độc lập với biểu đồ trạng thái.
- **Quản lý mô hình** – Tốt hơn về ký hiệu và ngữ nghĩa cho việc quản lý mô hình.
- **Cơ chế tổng quát** – Hỗ trợ kiểm soát mô hình và chuyển đổi biểu đồ.

Ngoài ra, người dùng cũng yêu cầu thay đổi cách đưa ra *đặc tả UML*. Với hơn 800 trang, cho thấy đây là một tài liệu cồng kềnh, người ta đã dự định sẽ tách những đặc tả về mô hình vật lý (những tiện ích CORBA và XMI) thành tài liệu riêng.

Biến cố quan trọng khác trong tương lai phát triển của UML là việc OMG dự định nộp UML cho Tổ chức tiêu chuẩn quốc tế (ISO) để chứng nhận chuẩn quốc tế cho công nghệ này.

2.6 Các *profile* của UML và khả năng mở rộng

Phiên bản 1.4 và 2.0 tiếp tục phát triển ý tưởng của UML. Một trong các đặc điểm của UML là nó có thể được tuỳ biến cho nhiều loại dự án khác nhau. Sở dĩ làm được điều này là nhờ có các cơ chế *constraint* (ràng buộc), *stereotype* (khuôn dạng) và *tagged value* (giá trị bổ sung), các cơ chế này được mô tả chi tiết trong chương 14. Các cơ chế mở rộng này có thể được đóng gói cùng nhau thành các *profile* cho các lĩnh vực khác nhau. (Lưu ý thuật ngữ *process extension* đôi khi được dùng thay cho *profile*, trong đặc tả UML thì *profile* được dùng). Có hai profile trong đặc tả UML là profile *phát triển phần mềm SDP* (Software Development Profile) và profile *mô hình nghiệp vụ BMP* (Business Modelling Profile). OMG đã đưa ra các yêu cầu RFP cho các profile mà có thể được áp dụng trong các lĩnh vực khác. Trong đó một cho CORBA và một cho tính toán phân bố.

Profile có ý nghĩa quan trong nhất là profile cho việc *lập lịch làm việc, hiệu suất công việc và thời gian làm việc*. Công việc này được thực hiện bởi nhóm *RADWG* (Real-time Analysis and Design Working Group) của ORG với hạn cuối vào tháng 6/2001. Những mở rộng hiện tại để phát triển những ứng dụng *thời gian thực* đã có trong chuỗi sản phẩm của công ty phần mềm *Rational* dưới dạng *Rational Rose RealTime*. CASE này sử dụng stereotype để biểu diễn những ký hiệu từ phương pháp *ROOM* (Real-Time Object-Oriented Modeling) (Selic,Gullekson & Ward, 1994). Tuy nhiên, các sản phẩm CASE khác lại sử dụng các ký hiệu khác đối với việc thiết kế các hệ thống thời gian thực, và có một *nhu cầu chuẩn hóa* cấp bách.

2.7 Tại sao dùng UML?

Mục này sẽ thảo luận tại sao nên dùng UML. Những ai mới làm quen với phân tích thiết kế, trước hết hãy thảo luận các lý do sử dụng tiếp cận trực quan. Sau đó chúng ta tìm hiểu các lợi ích mà UML đem lại trước khi đưa ra vài dự án có dùng UML để làm ví dụ.

2.7.1 Tại sao dùng các biểu đồ trong phân tích thiết kế?

Khi thiết kế các loại sản phẩm chúng ta đều dùng hình ảnh hoặc biểu đồ để trợ giúp. Các nhà thiết kế thời trang, các kỹ sư, các kiến trúc sư và các nhà phân tích thiết kế - tất cả đều sử dụng biểu đồ để hình dung công việc thiết kế của họ. Các phân tích và thiết kế viên dùng các biểu đồ để hình dung hệ thống phần mềm mà họ đang thiết kế, mặc dù sự thật là sản phẩm của qui trình thiết kế (tức là các chương trình máy tính) vốn không trực quan. Vậy việc sử dụng biểu đồ mang lại cho qui trình thiết kế những thuận lợi gì?

Có hai lợi ích chính khi dùng các biểu đồ để tạo ra một thiết kế:

- Trừu tượng hoá các thuộc tính của bản thiết kế.
- Chỉ ra các mối quan hệ giữa các thành phần của bản thiết kế.

Khi kiến trúc sư thiết kế một toà nhà, anh ta sẽ đưa ra một số bản vẽ với nhiều mục đích khác nhau. Bao gồm các biểu đồ cho một *cái nhìn về toàn bộ toà nhà với rất ít chi tiết* nhằm chỉ ra các đặc điểm đặc biệt của bản thiết kế, chẳng hạn như vị trí của ống nước; các biểu đồ vẽ ra *chi tiết bản thiết kế*, chẳng hạn như hình dạng của các vật bằng gỗ hoặc màu và vân của bề mặt bên ngoài. Không có biểu đồ nào có thể biểu hiện hết mọi phương diện của một đối tượng phức tạp, chẳng hạn như một toà nhà, và không một ai có thể xử lý hết mọi thông tin trong bản thiết kế toà nhà trong một lần. Điều này cũng giống với các hệ thống phần mềm: chúng rất phức tạp, và người thiết kế sẽ biểu diễn các khía cạnh khác nhau của bản thiết kế bằng các biểu đồ khác nhau. *Mỗi biểu đồ chỉ biểu diễn một hoặc nhiều khía cạnh đặc trưng từ bản thiết kế tổng quát.* Mặc dù thế, mỗi biểu đồ không thể biểu diễn từng chi tiết của các khía cạnh của bản thiết kế. Một biểu đồ ống nước trong một toà nhà có thể chỉ sử dụng các đường thẳng để biểu diễn cho các ống nước thay vì tìm cách vẽ độ rộng của ống nước theo tỉ lệ. Tương tự như vậy, một biểu đồ biểu diễn giao tiếp giữa các thành phần khác nhau trong một hệ thống phần mềm có thể dùng các đường thẳng để biểu diễn cho giao tiếp mà không cần tìm cách biểu diễn cách thức mà giao tiếp xảy ra.

Sử dụng biểu đồ để đơn giản hoá hệ thống và để biểu diễn các đặc điểm chính nào đó được gọi là *sự trừu tượng*. Trừu tượng hoá là một cơ chế được dùng để biểu diễn một sự vật phức tạp trở nên đơn giản hơn bằng cách dùng một số loại mô hình. Hơn nữa, nếu sự *trừu tượng* được biểu diễn ở mức vật lý, chẳng hạn như một biểu đồ trên giấy hoặc một đối tượng vật lý, thì chúng ta sẽ dùng thuật ngữ *mô hình*.

Trong phân tích thiết kế hệ thống, các mô hình được tạo ra để trừu tượng hoá các đặc điểm quan trọng của các hệ thống thế giới thực. Một

lớp UML mô tả một khách hàng chỉ gồm những đặc điểm của khách hàng liên quan đến hệ thống thông tin nghiệp vụ. Một lớp UML mô hình hành vi của chiếc máy bay trong hệ thống điều khiển không lưu sẽ mô hình chỉ các đặc điểm mà hệ thống điều khiển thời gian thực quan tâm. Trong cả hai trường hợp, người phân tích hoặc thiết kế quyết định đặc điểm nào là cần và đặc điểm nào là không cần.

Ví dụ 2.3

Trong hầu hết các hệ thống nghiệp vụ, các đặc điểm của khách hàng cần được quan tâm bao gồm *tên*, *địa chỉ*, *số điện thoại*, *số fax* và *địa chỉ email*. Màu tóc, cân nặng và chiều cao không phải là các đặc điểm cần thiết. Tuy nhiên, nếu hệ thống nghiệp vụ có liên quan đến câu lạc bộ làm ốm thì cân nặng sẽ là một đặc điểm cần được mô hình. *Trừu tượng hóa các đặc điểm thích hợp và xây dựng mô hình chính xác là kỹ năng của người phân tích*.

Các nhà phân tích và thiết kế hệ thống dùng các biểu đồ cho tất cả các mục đích và lý do vừa nêu. Hệ thống máy tính là một *sản phẩm* phức tạp được tạo thành từ phần mềm và phần cứng; các biểu đồ cung cấp cách mô hình các hệ thống này, chúng được cấu trúc với nhau như thế nào và chúng sẽ làm việc ra sao.

Các quan hệ giữa các thành phần của một bản thiết kế cũng có thể được vẽ bằng hình, hoặc bằng những văn bản hỗ trợ đính kèm theo mô hình. Trong kiến trúc, quan hệ giữa các thành phần có thể bao gồm cả sự cần thiết mô hình các *quan hệ về mặt cấu trúc* giữa sàn nhà và các thành phần khác của toà nhà (ví dụ như tường và rầm nhà). Khi mô hình bất cứ chủ đề gì, các mối quan hệ như vậy cũng quan trọng như chính bản thân các thành phần có liên quan với nhau. Các quan hệ trong mô hình có thể bao gồm:

- *Quan hệ cấu trúc* (structural relationship): sự phụ thuộc lẫn nhau giữa các thành phần của mô hình.
- *Quan hệ tổ chức* (organizational relationship): các thành phần của hệ thống phải cùng được đóng gói với nhau trong hệ thống cuối cùng để làm việc.
- *Quan hệ thời gian* (temporal relationship): minh họa cho chuỗi biến cố theo thời gian giữa các thành phần của mô hình.
- *Quan hệ nhân quả* (cause and effect relationship): chỉ ra các điều kiện tiên quyết phải có, trước khi một điều gì đó làm việc.

- *Quan hệ tiến hóa* (evolutionary relationship) giữa các mô hình theo thời gian: chỉ ra cách một thành phần được dẫn xuất từ thành phần khác trong suốt chu kỳ sống của dự án.

UML có tất cả các loại quan hệ trên. Danh sách sau đây sẽ cung cấp các ví dụ tương ứng cho từng loại quan hệ.

- Quan hệ cấu trúc: sự kết hợp giữa các lớp.
- Quan hệ tổ chức: package như là cách tổ chức các thành phần của mô hình
- Quan hệ thời gian: trình tự theo thời gian của các thông điệp trong một tương tác của biểu đồ tuần tự.
- Quan hệ nhân quả: các trạng thái trong biểu đồ trạng thái.
- Quan hệ tiến hoá: các đường phụ thuộc giữa các biểu đồ trong mô hình thiết kế và mô hình phân tích.

Ví dụ 2.4

Trong một câu lạc bộ làm ốm, các số cân của khách hàng được ghi lại trong một vài dịp nào đó và được tích luỹ thành một tập các số đo về cân nặng. Có một quan hệ cấu trúc giữa mỗi khách hàng và một tập các số đo cân nặng của khách hàng đó. Quan hệ này có thể trừu tượng hoá thành quan hệ giữa lớp *Customer* (khách hàng) và lớp *WeightMeasurement* (số đo cân nặng). Chúng ta cũng muốn mô hình mối quan hệ nguyên nhân kết quả, tỉ như nếu khách hàng sụt hơn một số cân nào đó thì khách hàng được cấp một chứng nhận.

2.7.2 Tại sao dùng đặc tả UML?

Trong dự án phát triển hệ thống hướng đối tượng, UML là một ngôn ngữ mô hình được ưu tiên cho việc phân tích và thiết kế một sản phẩm. (Chú ý rằng không phải tất cả những dự án phát triển đều sử dụng những ngôn ngữ hướng đối tượng, mặc dù có quảng cáo thổi phồng. Đối với những dự án sử dụng những ngôn ngữ thủ tục hoặc những ngôn ngữ hàm và đối với những dự án được cài đặt bằng cách dùng cơ sở dữ liệu quan hệ, thì các mô hình trong UML có thể khó chuyển thành cài đặt).

Lý do mạnh mẽ nhất để sử dụng UML bởi vì nó đã trở thành chuẩn thực tế đối với mô hình hướng đối tượng. Nếu cần thu hút một nhóm các nhà phát triển hoặc cần chuyển thông tin trong mô hình cho những người khác, thì UML là sự chọn lựa hiển nhiên vì nó dễ dàng giao tiếp giữa các bên tham gia.

Lý do thứ hai UML là ngôn ngữ mô hình hợp nhất. Nó là sản phẩm kết hợp từ các ý tưởng của ba nhà dẫn đầu trong việc lập mô hình hướng đối

tượng và kết hợp chúng thành một ký hiệu duy nhất. Từ các phiên bản đầu tiên, một số tổ chức có liên quan đến việc phát triển UML cũng cố gắng hợp tác các đặc điểm tốt nhất của các ngôn ngữ lập mô hình khác, vì thế UML có thể được xem là một ngôn ngữ tốt nhất trong lĩnh vực này. Tuy nhiên có một điều nguy hiểm ở đây là việc cố gắng hợp nhất nhiều quan điểm trên mô hình hướng đối tượng làm cho UML trở nên phình ra với nhiều ký hiệu thừa thải. RTF của OMG đã cố gắng tránh điều này bằng cách giữ cho phần cốt lõi của UML đơn giản, đồng thời dùng *profile* và các cơ chế mở rộng khác để mở rộng nó.

Đây là lý do thứ ba để sử dụng UML. Các *profile* đặc biệt đã tồn tại để giúp người sử dụng áp dụng cho một số vấn đề đặc trưng, hiện một số loại *profile* khác cũng đang được xây dựng. Nếu một vấn đề nào đó chưa có *profile* tương ứng, thì ta có thể mở rộng ký hiệu để áp dụng. Công việc của Jim Conallen trong việc áp dụng UML để mô hình hoá các hệ thống dựa trên web là một ví dụ điển hình, trong chương 14 chúng ta sẽ thảo luận chi tiết hơn vấn đề này.

2.8 *USDP* (Unified Software Development Process)

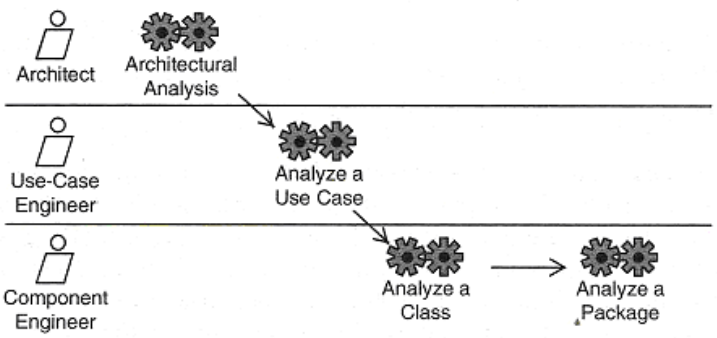
UML dùng để xác định hệ thống dưới dạng hình thức, nghĩa là nó không định nghĩa qui trình phân tích, thiết kế và cài đặt hệ thống. Những người phát triển UML cũng đã tạo ra một đặc tả của một qui trình phát triển phần mềm, nhằm giải thích cho cách suy nghĩ của họ về việc người phát triển dùng UML để phát triển hệ thống. Qui trình phát triển phần mềm này gọi là *qui trình phát triển phần mềm hợp nhất USDP* (Unified Software Development Process), hay *qui trình hợp nhất Rational RUP* (Rational Unified Process) hay gọi đơn giản là *qui trình hợp nhất UP* (Unified Process). Trong quyển sách này, chúng tôi gọi là UP.

UP bao gồm con người, dự án, sản phẩm, qui trình và công cụ. *Con người* là đối tượng tham gia, là người phát triển *dự án* nhằm tạo ra *sản phẩm* phần mềm theo một *qui trình* và dùng *công cụ* tự động để hỗ trợ. UP là một qui trình *tiếp cận theo tình huống sử dụng* (use-case-driven). Nghĩa là các yêu cầu của người sử dụng được mô tả trong các *use case*, là một chuỗi các hành động được thực hiện bởi hệ thống nhằm cung cấp một cái gì đó cho người dùng. Các *use case* này được người phát triển sử dụng như nền tảng của chuỗi công việc để tạo ra các mô hình thiết kế và mô hình cài đặt (kỹ thuật tạo use case sẽ được mô tả trong chương 3). UP cũng là qui trình *architecture-centric*, *iterative*, và *incremental* (tập trung kiến trúc, lặp và tăng trưởng). *Architecture-centric* nghĩa là kiến trúc của hệ thống phải được phát triển nhằm đáp ứng các yêu cầu của các use case chính, trong giới hạn của chuẩn phần cứng mà hệ thống sẽ chạy,

của cấu trúc của hệ thống và hệ thống con. *Iterative* nghĩa là dự án sẽ được chia thành các dự án nhỏ trong mỗi bước lặp. Mỗi dự án nhỏ sẽ được phân tích, thiết kế, cài đặt và kiểm tra. Mỗi phần như vậy là một *increment* và hệ thống sẽ được xây dựng dựa trên các *increment* này. Các bước lặp không phải đều giống nhau: các công việc trong mỗi bước lặp sẽ thay đổi trong toàn bộ chu trình. Trong UP, chu kỳ sống được phân ra thành 4 giai đoạn: *khởi tạo* (Inception), *phát triển* (Elaboration), *xây dựng* (Construction) và *chuyển tiếp* (Transition). Toàn bộ dự án có thể bao gồm nhiều chu kỳ, mỗi chu kỳ bao gồm 4 giai đoạn trên, và mỗi giai đoạn này lại bao gồm nhiều bước lặp.

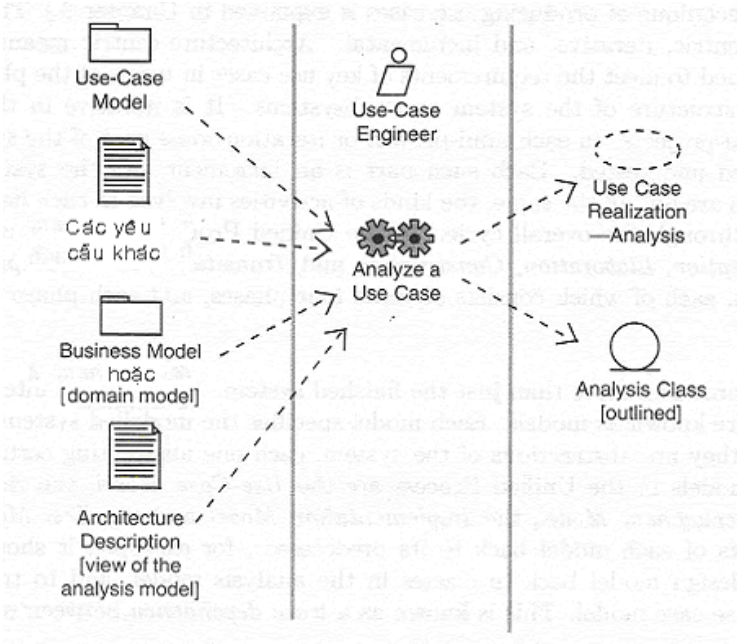
UP không chỉ tạo ra một hệ thống hoàn chỉnh mà còn tạo ra một số sản phẩm trung gian, chúng là các mô hình. Mỗi mô hình xác định một hệ thống được mô hình hoá theo một quan điểm cụ thể. Các mô hình chính trong UP là *Use-Case Model*, *Analysis Model*, *Design Model*, *Deployment Model*, *Implement Model* và *Test Model*. Có thể lần theo từng phần của mỗi mô hình ngược lên các mô hình trước. Ví dụ như có thể lần theo các class trong mô hình thiết kế để biết thông tin của các class này trong mô hình phân tích, và biết được các yêu cầu trong mô hình Use Case. Tính chất này gọi là *phụ thuộc vết* (trace dependency) giữa các mô hình.

Trong qui trình UP khía cạnh được định nghĩa dưới dạng các *hoạt động* (activity) nhằm chuyển đổi các yêu cầu của người dùng thành hệ thống làm việc. Các hoạt động này được nhóm lại với nhau thành các *dòng công việc* (workflow), mỗi dòng công việc được biểu diễn dưới dạng hình ảnh bằng cách dùng *biểu đồ hoạt động* (activity diagram). Hình 2.10 minh họa *dòng công việc phân tích* (analysis workflow), trong đó các tác giả của UP đã sử dụng một dạng stereotype của biểu đồ hoạt động để biểu diễn dòng công việc.



Hình 2.10: Analysis workflow

Workflow cũng xác định các *worker* để thực hiện các hoạt động. *Worker* không là người cụ thể mà là một vai trò trừu tượng. Có thể có nhiều người cùng thực hiện vai trò của *worker*, hoặc nhiều vai trò worker được thực hiện bởi cùng một người. Mỗi hoạt động được chia thành các *bước* chi tiết, được đặc tả đầu vào là các mô hình, các kết quả của dự án khác, và đầu ra (artefact) là các kết quả do hoạt động tạo ra. Hình 2.11 minh họa hoạt động *phân tích một use case* (Analyze a Use Case).



Hình 2.11: Thông tin vào ra của hoạt động Analyze a Use Case

2.9 Tìm thêm thông tin ở đâu?

Để tìm hiểu thêm về UML có thể tham khảo *đặc tả UML* (OMG 1999a). Tuy nhiên, nó được viết như là tài liệu tham khảo cho chuyên gia với hơn 800 trang.

Addison-Wesley đã xuất bản một số sách về UML, nhiều cuốn được viết chung với công ty *Rational*. Các sách này bao gồm sách của các tác giả đầu tiên của UML là *Grady Booch, Ivar Jacobson và Jim Rumbaugh*: một cuốn hướng dẫn tham khảo bám sát các đặc tả (Rumbaugh, Jacobson & Booch. 1999), một cuốn hướng dẫn sử dụng với một *case study* (Booch, Rumbaugh & Jacobson. 1999), và một cuốn hướng dẫn về UP (Jacobson, Booch & Rumbaugh. 1999). Ba tài liệu UML này khoảng 1500 trang dành cho các độc giả đã quen với cách thực hiện một dự án phân tích thiết kế hệ thống.

McGraw-Hill đã xuất bản một tài liệu giáo khoa về phân tích và thiết kế hệ thống sử dụng các ký hiệu của UML (Bennett, McRobb & Farmer, 1999).

Thông tin về phiên bản hiện tại và bản phát triển của UML hiện có trên web site của công ty *Rational* (www.rational.com), trên web site của OMG (www.omg.org). Site của OMG có liên kết tới các diễn đàn của UML và các trang do thành viên của RTF (*Revision Task Force*) chịu trách nhiệm cũng như các tài nguyên khác của UML. Web site của công ty *Rational* thì liên kết với các *case study* của các tổ chức sử dụng UML và *Rational Rose* phát triển các dự án.

Tài liệu của *Cris Kobyrn* (Kobyrn, 1999) trong *Communications of the ACM* cung cấp một cái nhìn toàn cảnh của sự phát triển UML, mặc dù nó hơi cũ.

Câu hỏi ôn tập

2.1 Ba tác giả gia nhập công ty phần mềm *Rational* để phát triển UML là ai?

2.2 Tổ chức nào hiện nay chịu trách nhiệm về chuẩn UML?

2.3 Mục đích của UML XMI DTD là gì?

2.4 Ba loại qui luật nào được sử dụng để định nghĩa UML?

2.5 Bốn tầng của kiến trúc metamodel trong UML là gì?

2.6 Các luật well-formedness được đặc tả như thế nào?

2.7 Các package được sử dụng để làm gì trong UML?

2.8 Nhóm nào trong OMG chịu trách nhiệm về tương lai UML?

2.9 Các *profile* UML được sử dụng để làm gì?

2.10 Cái nào sau đây là một khái niệm hay một sự vật trừu tượng?

- a Một bản đồ được bạn phác thảo bằng vài đường thẳng trên một mẩu giấy để chỉ đường đến nhà của bạn.
- b Một bản đồ đường phố của London, Anh quốc.
- c Thành phố London, Anh quốc.
- d Một biểu đồ lớp của UML.

2.11 Cái nào sau đây là một mô hình?

- a Một biểu đồ lớp của UML.
- b Một tập các biểu đồ lớp mô tả các lớp trong hệ thống phần mềm.
- c Một bản sao xe hơi thể thao bằng đất sét tỉ lệ 1:100 được sử dụng để kiểm tra đặc tính khí động lực

d Vật nguyên mẫu bằng kích thước thật của một xe hơi thể thao mới.

2.12 Đưa ra ba lý do sử dụng UML.

2.13 Bốn giai đoạn của chu kỳ sống của UP là gì?

2.14 Giải thích các mối quan hệ giữa *dòng công việc*, *hoạt động* và các *bước* trong UP.

Bài tập bổ sung

2.1 Nghiên cứu một trong những ký hiệu của nguyên mẫu đầu tiên của UML (của Rimbaugh, Booch hoặc của Jacobson). Chọn một biểu đồ UML và biểu đồ tương đương rồi liệt kê các điểm giống và khác nhau giữa chúng.

2.2 Nghiên cứu ba ký hiệu nguyên mẫu đầu tiên của UML (của Rimbaugh, Booch hoặc Jacobson). Chọn một loại biểu đồ và so sánh giữa chúng.

2.3 Một trong những nguyên tắc trong thiết kế tương tác giữa người và máy là tính *affordance*. Nghĩa là hình dạng của một đối tượng nên gợi lên cho ta thấy được mục đích của nó. Dựa trên nghiên cứu của bạn trong hai câu hỏi trước, bạn có nghĩ rằng những biểu tượng của biểu đồ được sử dụng trong UML có đặc tính *affordance* rõ rệt, hoặc chúng chỉ là một tập những ký hiệu tuỳ ý? Tìm các ví dụ cho cả hai phạm trù này.

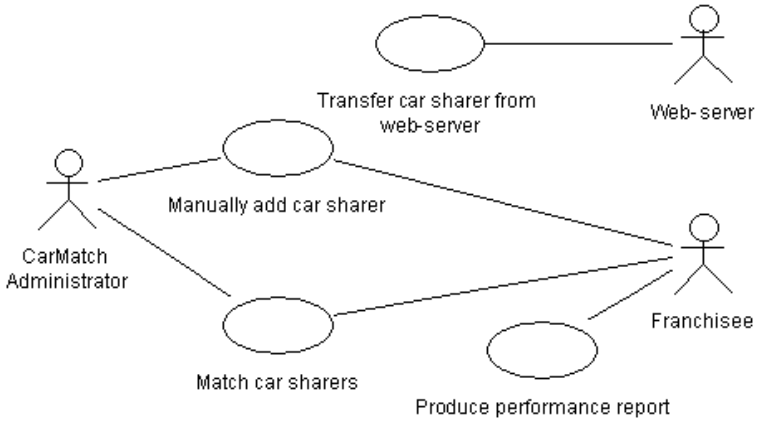
2.4 Tìm một số *case study* sử dụng UML. (Có một số trên website của công ty phần mềm Rational, mặc dù hiện giờ chúng được khuyến khích dùng công cụ CASE của Rational Rose). Xác định các lợi ích khi sử dụng UML.

Chương 3
USE CASE

3.1 Giới thiệu

Use case cung cấp một bức tranh toàn cảnh về những gì đang xảy ra trong hệ thống hiện tại hoặc những gì sẽ xảy ra trong hệ thống mới, do đó có rất nhiều dự án tin học được bắt đầu từ các use case. *Biểu đồ use case* (use case diagram) rất đơn giản với rất ít ký hiệu. Đây là một phương tiện giao tiếp hữu hiệu với người dùng về hệ thống, về những gì hệ thống được dự định sẽ làm. *Use case* cũng có thể được dùng làm cơ sở cho các *đặc tả kiểm tra* (test specification) sau này.

Biểu đồ use case đưa ra các *use case* (tình huống sử dụng), các *actor* (tác nhân) và các *association* (quan hệ kết hợp) giữa chúng. Hình 3.1 cho thấy sự đơn giản của một biểu đồ use case. *Use case* biểu diễn chuỗi hành động mà hệ thống thực hiện, *actor* biểu diễn người hoặc hệ thống khác tương tác với hệ thống đang được mô hình hoá. Các biểu đồ use case được hỗ trợ bởi các *đặc tả hành vi* (behaviour specification), nhằm định nghĩa các tương tác bên trong một use case nào đó.



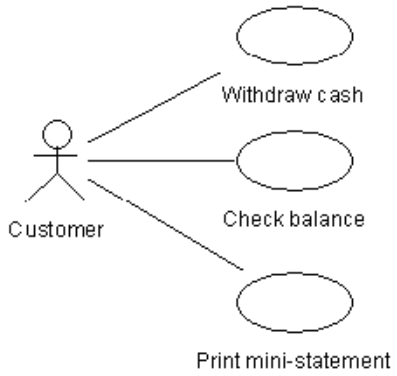
Hình 3.1: Biểu đồ use case *trong hệ thống* CarMatch

Các use case đã được Jacobson đưa vào UML (1992), ban đầu được thực hiện trong công ty Ericsson, Thụy Điển. Trong tiếp cận của Jacobson,

use case là điểm bắt đầu cho việc phát triển một hệ thống mới. Trong thuật ngữ UML, gói *use case* là một gói con của gói *Behavioural Element* (phần tử hành vi). Nghĩa là nó được sử dụng để xác định hành vi của một số thực thể, chẳng hạn một hệ thống hoặc một hệ thống con. Use case không xác định chi tiết cách mà hành vi được thực hiện như thế nào, chi tiết này sẽ được thảo luận trong các mô hình khác của qui trình thiết kế hệ thống. Thông thường, cách thực hiện một use case được định nghĩa trong một hoặc nhiều *biểu đồ cộng tác* (collaboration diagram) nhằm biểu diễn sự tương tác giữa các đối tượng cùng hoạt động.

Ví dụ 3.1:

Trong hệ thống ngân hàng, các use case định nghĩa tương tác giữa khách hàng và *máy rút tiền tự động ATM* (Automated Teller Machines). Hình 3.2 là một biểu đồ use case đơn giản. *Customer* biểu diễn *lớp* (class) của tất cả khách hàng sử dụng. Khi bạn dùng ATM để rút tiền, thì bạn là một *thể hiện* (instance) của *Customer* đang dùng một *thể hiện* nào đó của use case *withdraw cash*. Một người khác cũng đến rút tiền, anh ta là một thể hiện khác của *Customer*. Bạn có thể rút tiền thành công, còn anh bạn này thì không. Anh ta nhận ra rằng anh ta không còn đủ tiền để rút, và *withdraw cash* xử lý tình huống này khác với của bạn, đó là từ chối yêu cầu. Một số người khác có thể dùng một thể hiện khác của use case *Check balance* hoặc use case *Print mini-statement*.



Hình 3.2: Biểu đồ use case *của hệ thống con* ATM

Thuật ngữ *scenario* (kịch bản) thường được dùng để ám chỉ đến các diễn biến khác nhau mà các thể hiện của cùng một use case sẽ thực hiện. Biểu đồ use case chỉ đặt tên cho các use case, các tài liệu hoặc các biểu đồ kèm theo sẽ chỉ ra các *scenario*.

Ví dụ 3.2

Các *scenario* của use case *withdraw cash* có thể là:

- Thẻ của khách hàng không được nhận biết và bị từ chối.
- Khách hàng nhập mã số (PIN) sai và được yêu cầu nhập lại.
- Khách hàng nhập PIN sai ba lần và card sẽ bị AMT giữ lại.
- Khách hàng nhập số lượng tiền rút không hợp lệ.
- ATM cố gắng kết nối với hệ thống của ngân hàng nhưng không được do nó bị trục trặc hoặc mạng có vấn đề.
- ATM không đủ tiền mặt để đáp ứng yêu cầu của khách hàng.
- Tài khoản của khách hàng không đủ tiền để đáp ứng yêu cầu.
- Khách hàng huỷ bỏ việc rút tiền.

Bạn có thể nghĩ ra thêm các *scenario* khác, use case *withdraw cash* sẽ biểu diễn tất cả. Cuối cùng, use case này phải được xác định thật chi tiết, đủ để các đối tượng tạo ra hệ thống xử lý một cách chính xác. Ở giai đoạn đầu, chúng ta nên xác định các diễn biến đơn giản (khách hàng rút thành công lượng tiền yêu cầu) và các *scenario* gần gũi nhất. Như bạn sẽ thấy trong phần 3.3 sau đây, các *mô tả use case* thường được dùng để định nghĩa các *scenario* khác nhau.

3.2 Mục đích của biểu đồ use case

Các use case được tạo ra ở giai đoạn đầu của một dự án. Tạo ra các biểu đồ use case và các tài liệu là công việc thiên về kỹ thuật phân tích hơn kỹ thuật thiết kế. Các use case cũng có thể được dùng ở các giai đoạn sau của qui trình phát triển dự án, ví dụ để đặc tả các tình huống kiểm tra. Các use case được dùng để mô hình hoá cái gì xảy ra trong hệ thống hiện tại, hoặc có thể được dùng để mô hình hoá hệ thống sắp được phát triển.

Các mục đích sử dụng chính:

- Dùng để mô hình hoá các chuỗi hành động mà hệ thống sẽ thực hiện, và nhằm cung cấp một kết quả có ý nghĩa cho một người nào đó hoặc một hệ thống bên ngoài (gọi chung là actor).
- Cung cấp một cái nhìn tổng thể về những gì mà hệ thống sẽ làm và ai sẽ dùng nó.
- Đưa ra cơ sở để xác định giao tiếp người-máy đối với hệ thống.
- Dùng để mô hình hoá các *scenario* cho một use case.
- Để người dùng cuối có thể hiểu được và có thể giao tiếp với hệ thống ở mức tổng thể.
- Làm cơ sở cho việc phác thảo ra các đặc tả kiểm tra.

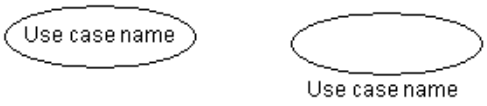
3.3 Ký hiệu

Use case mô tả một chuỗi các hành động mà hệ thống sẽ thực hiện để đạt được kết quả có ý nghĩa đối với một tác nhân. Mô tả có thể được biểu diễn bằng văn bản có cấu trúc (chẳng hạn như mã giả), hoặc thông qua một *đặc tả hành vi* được biểu diễn bởi một liên kết đến một biểu đồ khác (chẳng hạn như biểu đồ cộng tác). Cái nhìn ở mức cao của các use case sẽ được biểu diễn bởi biểu đồ use case.

3.3.1 Các ký hiệu cơ bản: *use case, actor* và *relationship*

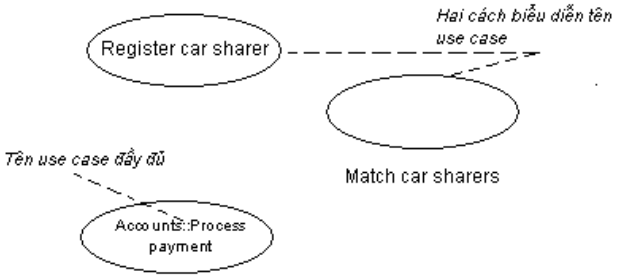
Use case, được biểu diễn bằng đồ thị trong *biểu đồ use case*, cho phép phân tích viên hình dung từng use case trong ngữ cảnh của các use case khác, và cho thấy các mối quan hệ của nó với các actor, với các use case khác.

Trong biểu đồ, use case được biểu diễn bằng hình ellipse (hình 3.3). Tên use case được đặt bên trong hoặc bên dưới. Chúng ta phải biểu diễn nhất quán, nghĩa là trong cùng một biểu đồ không nên biểu diễn use case bằng cả hai cách.



Hình 3.3: Ký hiệu use case

Tên use case là một chuỗi gồm các ký tự, các con số và hầu hết các dấu phân cách, ngoại trừ dấu hai chấm bởi dấu hai chấm được dùng để phân tách tên của use case với tên của package. Qui ước đặt tên cho use case như sau: trước hết là một động từ và sau đó là danh từ hoặc cụm danh từ mô tả hành vi, chẳng hạn: *Regsiter car sharer* (đăng ký thành viên), *Match car sharers* (kết hợp các thành viên) hay *Record sharing agreement* (ghi lại bản thoả thuận) - xem hình 3.4.

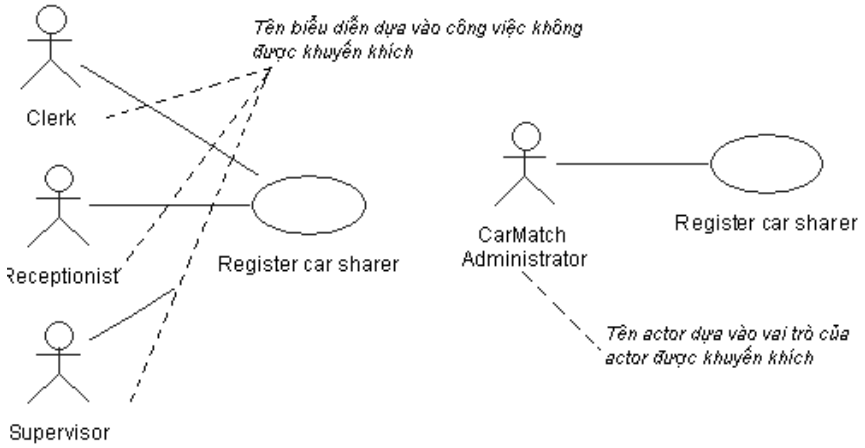


Hình 3.4: Các tên use case *hợp lệ*

Actor là người hoặc hệ thống tương tác với các use case. Thường actor là người dùng hệ thống. Trong biểu đồ use case, mỗi actor được vẽ bằng một biểu tượng hình người với *tên vai trò* (role name) đặt bên dưới (hình 3.5).

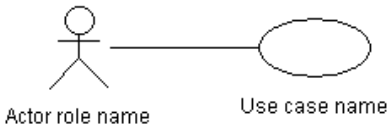
Hình 3.5: Ký hiệu actor

Khi actor là người, thì tên actor là tên vai trò mà actor đảm nhiệm chứ không phải là tên công việc. Trong một văn phòng *CarMatch*, có nhiều người, mỗi người có một công việc khác nhau (thư ký, tiếp tân, giám sát), tất cả những người này đều có thể đăng ký cho khách hàng chia sẻ xe trong hệ thống *CarMatch*. Thay vì vẽ ba actor khác nhau, chúng ta xác định công việc chung của họ và tạo một actor cho vai trò đó, trong trường hợp này ta chọn tên của actor là *CarMatch Administrator*, xem hình 3.6.



Hình 3.6: Cách đặt tên cho actor

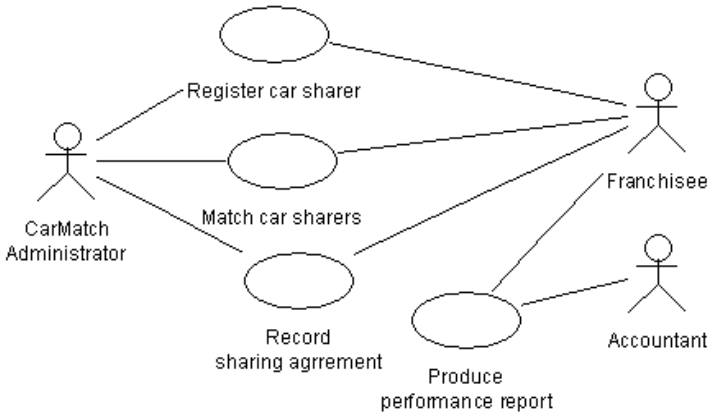
Đoạn thẳng nối actor với use case mô tả mối quan hệ giữa chúng, là mối quan hệ *tương tác* giữa actor và use case, xem hình 3.7.



Hình 3.7: Quan hệ giữa actor *và* use case

Quan hệ này cho biết có một kết hợp giữa một actor và một use case. Nghĩa là con người hoặc hệ thống, trong vai trò actor này, sẽ giao tiếp với các thể hiện của use case, tham gia vào chuỗi các biến cố được use case biểu diễn. Trong thực hành, use case được cài đặt thành một vài loại chương trình máy tính và các actor sẽ dùng những chương trình này bằng cách nhập thông tin vào, nhận thông tin ra.

Một actor có thể được kết hợp với một hoặc nhiều use case, và một use case có thể được kết hợp với một hoặc nhiều actor. Xem hình 3.8.



Hình 3.8: Actor *và* use case

3.3.2 Đặc tả hành vi (behaviour specification)

Mỗi use case biểu diễn một chuỗi hoạt động đưa ra một số kết quả đối với một hoặc nhiều actor. Chuỗi hoạt động này được sưu liệu trong một đặc tả hành vi. Một công cụ CASE có thể biểu diễn đặc tả này bằng một biểu đồ. Biểu đồ này có thể là *biểu đồ tuần tự* (sequence diagram), *biểu đồ cộng tác* (collaboration diagram), *biểu đồ trạng thái* (statechat diagram) hoặc có thể là văn bản (mã của một ngôn ngữ lập trình). Thông thường thì một đặc tả không hình thức được cung cấp như là một *mô tả use case*. Trong CASE, có thể dùng cả hai cách để mô tả use case: đặc tả không hình thức và đặc tả hình thức bằng cách liên kết đến một biểu đồ.

Một số biểu đồ UML có các quy tắc cú pháp để biểu diễn thông tin văn bản. Mục đích là để nó có thể thực thi mô hình và mô phỏng hệ thống, thậm chí chuyển mô hình thành mã chương trình của một ngôn ngữ nào đó, như C++ hoặc Java. Tuy nhiên các mô tả use case không có bất kỳ quy tắc cú pháp nào cả, bạn có thể mô tả bằng bất kỳ dạng nào bạn thích. Tất nhiên, khi làm việc trong một công ty, hẳn bạn phải tuân thủ các quy tắc của công ty khi lập sưu liệu cho các use case.

Có hai hướng tiếp cận thông dụng để viết các mô tả use case:

- Thứ nhất là viết ngắn gọn một hoặc một vài phát biểu hoặc đoạn văn để mô tả chuỗi hoạt động tiêu biểu của use case.
- Thứ hai là liệt kê thành hai cột, một cột là các hoạt động của tác nhân và cột còn lại là các đáp ứng của hệ thống.

Hình 3.9 mô tả theo tiếp cận thứ nhất cho use case *Register car sharer*, và hình 3.10 mô tả theo tiếp cận thứ hai cũng cho use case này. Có thể lúc đầu bạn dùng tiếp cận thứ nhất, sau khi đã hiểu rõ hơn các yêu cầu sẽ áp dụng tiếp cận thứ hai.

> Người sử dụng nhập *tên, địa chỉ* và *số điện thoại* của thành viên tiềm năng. Với mỗi lộ trình mà người này muốn chia sẻ, nhập *địa chỉ bắt đầu, địa chỉ đích, thời gian bắt đầu* và *thời gian kết thúc* của lộ trình.

Hình 3.9: Mô tả đơn giản cho Register car sharer

	Tác nhân	*Hệ thống*
1	Người dùng nhập tên và địa chỉ của người chia sẻ xe.	Hệ thống xác nhận địa chỉ khớp với cơ sở dữ liệu địa lý (*geographical database*).
2	Người dùng nhập số điện thoại của người chia sẻ xe.	Hệ thống yêu cầu cung cấp các thông tin chi tiết của lộ trình.
3	Người dùng nhập thời gian bắt đầu và địa chỉ xuất phát.	Hệ thống xác nhận địa chỉ xuất phát khớp với cơ sở dữ liệu địa lý.
4	Người dùng nhập thời gian kết thúc và địa chỉ đích.	Hệ thống xác nhận địa chỉ đích khớp với cơ sở dữ liệu địa lý. Hệ thống hỏi người dùng có muốn nhập vào lộ trình mới hay không.
5	Người dùng hoặc nhập lộ trình khác (quay lại bước 3) hoặc lưu và thoát.	Hệ thống hoặc nhắc nhập thêm lộ trình (quay lại bước 3) hoặc lưu người chia sẻ xe, thông tin chi tiết của lộ trình và thoát khỏi use case.

Hình 3.10: Mô tả use case cho Register car sharer *dùng cột*

Constantine (1997) phân loại các mô tả use case bằng một cách khác tuỳ vào chúng biểu diễn một khung nhìn logic hay vật lý. Ông ta phân biệt giữa *bản chất* và *thực tế*. Bản chất ở đây là use case nắm được cốt lõi những gì cần làm, không phụ thuộc vào thiết kế của hệ thống cuối cùng. Hình 3.10 là một mô tả bản chất, còn hình 3.11 là một mô tả thực tế của use case *Register car sharer*.

	Tác nhân	*Hệ thống*
1	Người dùng nhập tên và địa chỉ của người chia sẻ xe trong cửa sổ *Car Sharer entry window*.	Hệ thống xác nhận địa chỉ khớp với cơ sở dữ liệu địa lý.
2	Người dùng nhập số điện thoại của người chia sẻ xe trong cửa sổ *Car Sharer entry window*.	Hệ thống yêu cầu cung cấp thông tin chi tiết của lộ trình bằng cách trình bày hộp thoại *Journey entry dialogue box.*
3	Người dùng nhập thời gian bắt đầu và địa chỉ xuất phát.	Hệ thống xác nhận địa chỉ xuất phát khớp với cơ sở dữ liệu địa lý.
4	Người dùng nhập thời gian kết thúc và địa chỉ đích.	Hệ thống xác nhận địa chỉ đích khớp với cơ sở dữ liệu địa lý. Hệ thống nhắc người dùng nếu muốn nhập lộ trình mới bằng cách hiển thị hai nút *Another* và *Done*
5	Người dùng nhấp vào nút *Another* để nhập lộ trình khác (quay lại bước 3) hoặc nhấp vào nút *Done* để lưu và thoát.	Hệ thống hoặc sẽ xoá hộp thoại nhập lộ trình (quay lại bước 3) hoặc sẽ đóng hộp thoại, lưu thông tin chi tiết của người chia sẻ xe và chi tiết của lộ trình vừa nhập; thoát khỏi thể hiện của use case.

Hình 3.11: Mô tả thực tế *use case* Register car sharer

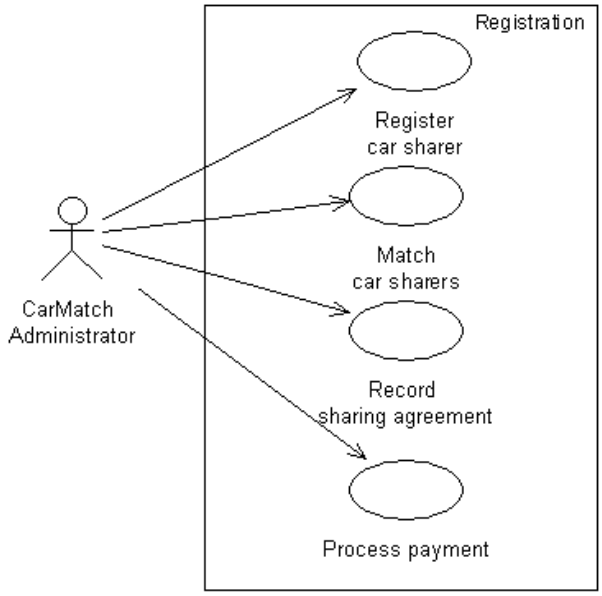
Các use case trong một biểu đồ có thể được đặt trong một hình chữ nhật. Hình chữ nhật này hoặc ánh xạ vào mô hình use case hoàn chỉnh chứa tất cả các use case và actor, hoặc ánh xạ vào hệ thống hay hệ thống con chứa chúng. Hình 3.12 minh họa một phác thảo sớm của hệ thống con *Registration* (thuộc *CarMatch*).

3.3.3 Các kiểu *kết hợp* (association) và *quan hệ* (relationship)

Có 4 kiểu kết hợp và quan hệ trong một biểu đồ use case:

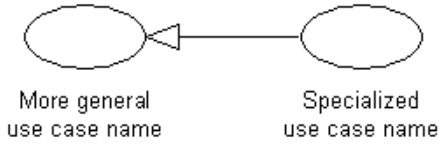
- Kết hợp *generalization* (tổng quát hoá) giữa các use case.
- Kết hợp *generalization* giữa các actor.
- Quan hệ *include* (bao gồm) giữa các use case.
- Quan hệ *extend* (mở rộng) giữa các use case.

Lưu ý trong UML Version 1.1, *include* và *extend* được gọi là *uses* và *extends*.



Hình 3.12: Các use case trong một hệ thống con

3.3.4 Kết hợp *generalization* giữa các use case

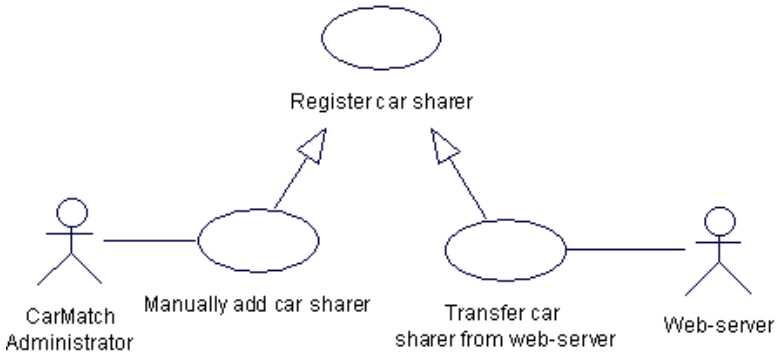


Hình 3.13: Ký hiệu kết hợp generalization *giữa hai use case*

Đôi khi chúng ta gặp trường hợp cùng một use case lại có nhiều phiên bản khác nhau, các phiên bản này có một số hành động giống nhau và một số hành động không giống nhau. Hãy khảo sát hai use case cùng chức năng cho phép thêm thành viên vào hệ thống như sau: *Manually add*

car sharer và *Transfer car sharer from web-server*. Hai use case có cùng một chức năng nhưng cách hoạt động lại không hoàn toàn giống nhau. Ta có thể xem chúng như hai use case đặc biệt của *Register car sharer*, lúc này ta gọi *Register car sharer* là một use case tổng quát. Điều này được biểu diễn trong biểu đồ thông qua việc dùng *generalization*. Kết hợp *genralization* được vẽ bằng một biểu tượng hình tam giác trên đường nối hướng đến use case tổng quát, xem hình 3.13.

Biểu đồ trong hình 3.14 cho biết hai use case *Manally add car sharer* và *Transfer car sharer from web-server* kế thừa một số tính năng từ use case *Register car sharer*, nhưng có một vài điểm khác biệt. Một use case sẽ được liên kết đến một giao diện người dùng để cho phép actor *CarMatch Administrator* nhập vào các thông tin chi tiết; một use case cho nhập dữ liệu trên web-server. Lưu ý rằng tác nhân không nhất thiết là người, mà có thể là một hệ thống khác.



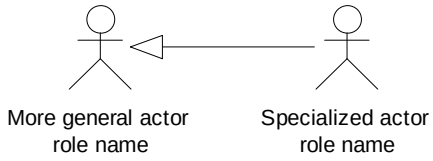
Hình 3.14: Các use case trong kết hợp generalization

Đôi khi một use case tổng quát ở mức cao là use case không bao giờ tồn tại trong hệ thống thực, nó được dùng để định nghĩa các chức năng chung cho các use case cụ thể. Trong trường hợp này chúng ta gọi đó là *asbtract use case* (use case trừu tượng), tên của use case trừu tượng sẽ được in nghiêng. Các use case đặc biệt thường được gọi là *concrete use case* (use case cụ thể). Trong quyển sách này chúng ta gọi nó là *real use case* (use case thực, xem phần 3.3.2).

Generalization thường được cài đặt bằng kỹ thuật *kế thừa* (inheritance), trong chương 5 chúng ta sẽ quay lại vấn đề này.

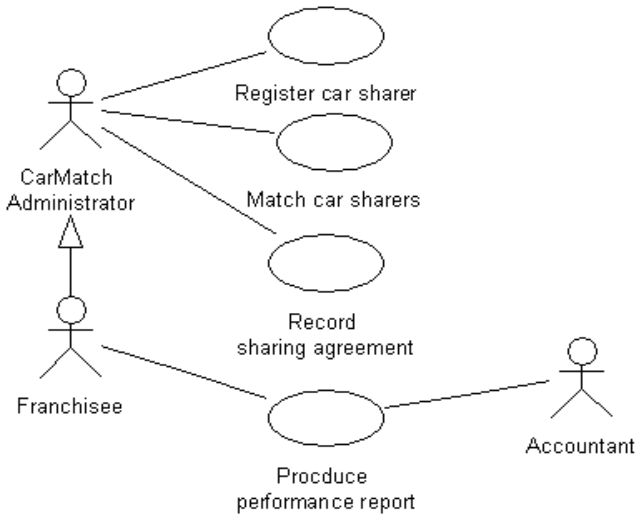
3.3.5 Kết hợp *generalization* giữa các Actor

Giữa các actor cũng tồn tại kết hợp generalization như trong hình 3.15.



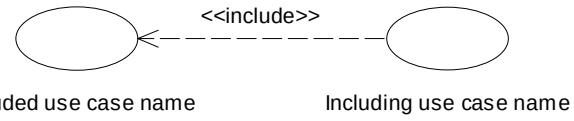
Hình 3.15: Ký hiệu kết hợp generalization *giữa các actor*

Tại CarMatch, *Franchisee* quản lý cả việc đăng ký thành viên mới nhưng cũng là người nhận các báo cáo. Thay vì biểu diễn như trong hình 3.8, ta có thể biểu diễn *Franchisee* là một trường hợp đặc biệt của *CarMatch Administrator*. Nghĩa là *Franchisee* có thể thực hiện mọi hành động của *CarMatch Administrator*, ngoài ra nó còn có riêng một số hoạt động khác, xem hình 3.16.



Hình 3.16: Các actor với kết hợp generalization

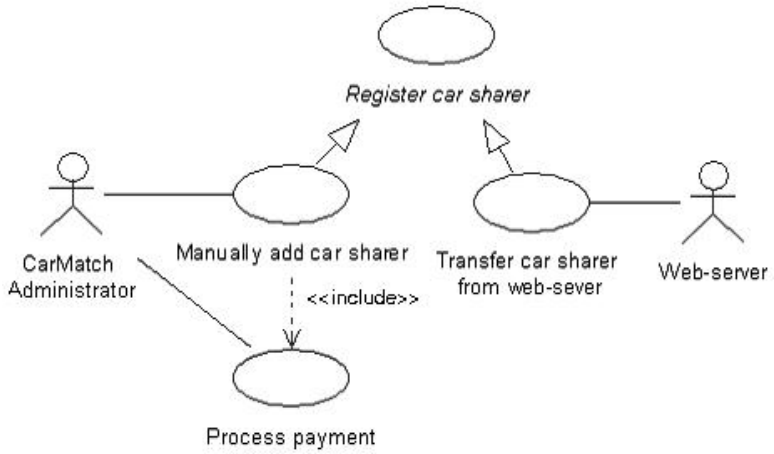
3.3.6 Quan hệ *include* giữa các use case



Hình 3.17: Ký hiệu quan hệ include

Đôi khi một use case có các tính năng của một use case khác, khi đó tồn tại quan hệ *include* (bao gồm) giữa chúng. Quan hệ <<*include*>> được vẽ bằng một mũi tên đứt nét hướng đến use case thứ hai (xem hình 3.17).

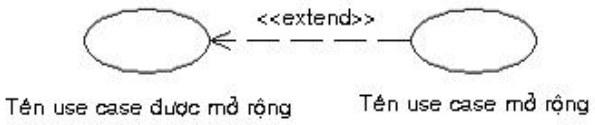
Trong *CarMatch* cũng có loại quan hệ này, đó là *use case* thực hiện việc thanh toán. Use case này có thể tồn tại như một use case đứng tự do để xử lý các thanh toán (thông tin được truyền từ web-server), nhưng khi một thành viên mới được thêm bằng vào *Manually add car sharer* thì phí thành viên luôn được thực hiện đồng thời. Vì vậy use case *Process payments* vừa là một use case riêng vừa là use case được *bao gồm* bên trong *Manually add car sharer*. Hình 3.18 minh họa tình huống này.



Hình 3.18: Các use case với quan hệ include

Quan hệ *include* cũng được dùng khi một use case được *bao gồm* bên trong các use case khác bởi vì nó đóng gói một số tính năng được sử dụng ở một số nơi trong hệ thống. Điều này tránh được tình trạng cùng một chuỗi hành động lại định nghĩa trong nhiều use case. Use case *bao gồm* sẽ tiếp tục chuỗi hành động tại điểm mà nó *include*, sau đó quay lại và tiếp tục quá trình tạm dừng trước đó.

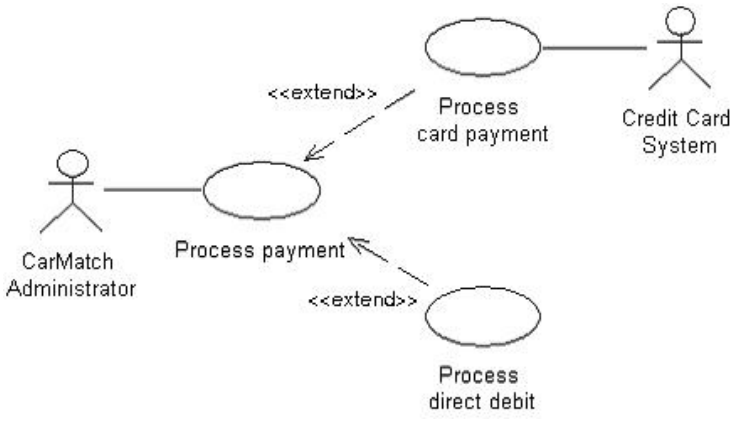
3.3.7 Quan hệ *extend* giữa các use case



Hình 3.19: Ký hiệu của quan hệ extend

Trong khi quan hệ *include* là quan hệ use case này chứa use case khác, thì quan hệ *extend* là quan hệ mở rộng một use case. Quan hệ được vẽ bằng một mũi tên đứt nét hướng đến use case *được* mở rộng. Từ <<*extend*>> được đặt bên cạnh mũi tên quan hệ như trong hình 3.19.

Ví dụ, nếu một thành viên thanh toán lệ phí bằng tiền mặt hoặc bằng ngân phiếu, thì *Process payment* sẽ có đầy đủ chức năng để thực hiện việc thanh toán trên. Tuy nhiên nếu hình thức là thanh toán bằng điện tử (bank direct debit) hoặc bằng thẻ tín dụng (credit card) hoặc thẻ nợ (debit card), thì use case *Process payment* không thể thực hiện được. Lúc này, nó có thể được mở rộng sang use case *Process card payment* hoặc *Process direct debit*, xem ví dụ trong hình 3.20.



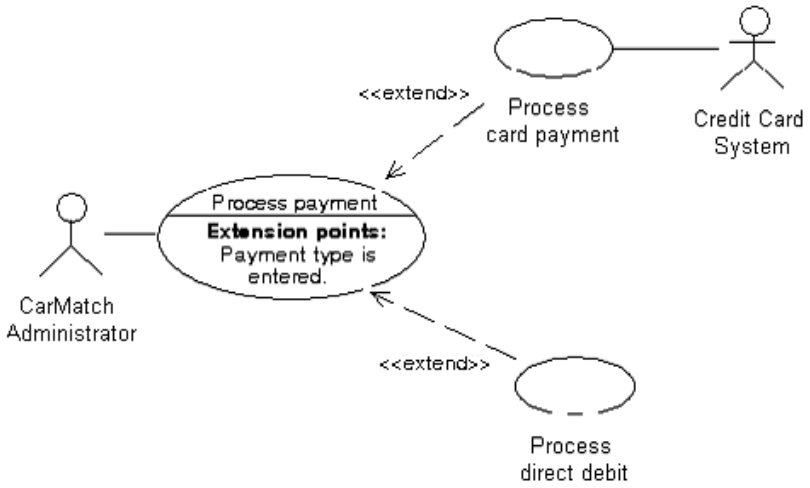
Hình 3.20: Các use case với quan hệ extend

Chi tiết của các điểm trong use case tại đó sự mở rộng xảy ra được mô tả ở phần bên dưới nằm trong biểu tượng use case. Phần này có tên là *Extension points* (các điểm mở rộng), xem hình 3.21.

Trong quan hệ *extend*, một số điều kiện phải thoả thì use case *mở rộng* mới được thực hiện. Với ví dụ trong hình 3.12, có ba điều kiện sau:

- paymentType = *Credit card* hoặc *debit card*
- paymentType = *cash* hoặc *cheque*
- paymentType = *direct debit*

Chỉ cần một trong ba điều kiện trên được thoả, nghĩa là chỉ một trong hai use case mở rộng sẽ được sử dụng. Nếu thành viên thanh toán phí bằng tiền mặt thì cả hai use case đều không được sử dụng.



Hình 3.21: Use case với Extension points

3.4 Làm thế nào để đưa ra các use case?

Người phân tích hệ thống, người đang tạo ra các biểu đồ use case và các mô tả use case, sẽ đưa ra các use case dựa trên các tài liệu nguồn chẳng hạn như các ghi chú và các bản ghi chép lại từ các cuộc phỏng vấn (một cuộc phỏng vấn được ghi lại qua băng từ và sau đó nội dung của băng từ được sao chép lại nguyên văn vào trong một tài liệu xử lý văn bản).

Qui trình tạo ra các use case liên quan đến các bước sau:

- Tìm các actor và các use case.
- Sắp xếp độ ưu tiên cho các use case vừa tìm được.
- Phát triển từng use case (bắt đầu từ use case có độ ưu tiên cao nhất).
- Lập cấu trúc cho mô hình use case.

Hướng tiếp cận này dựa trên các hoạt động trong UP (xem mục 3.7).

3.4.1 Tìm các actor và use case

Actor biểu diễn vai trò do con người đảm nhiệm, hoặc hệ thống khác tương tác với hệ thống đang xét. Để dễ dàng xác định các actor, ta đi tìm câu trả lời cho các câu hỏi sau:

- Ai là người sẽ dùng hệ thống này để *nhập* thông tin?
- Ai là người sẽ dùng hệ thống này để *nhận* thông tin?
- Các hệ thống nào tương tác với hệ thống này?

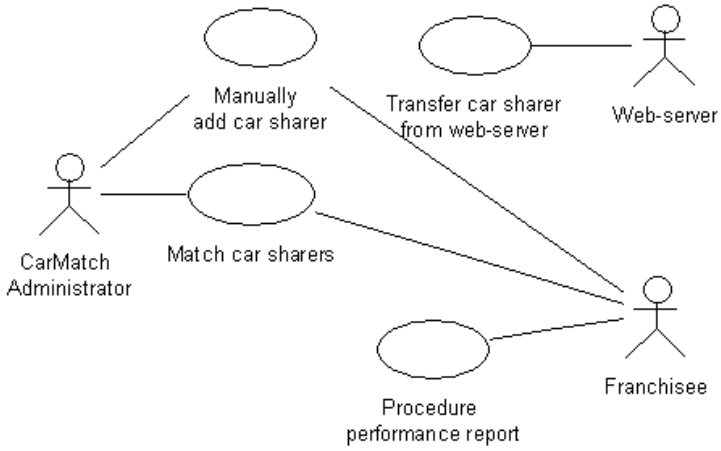
Ví dụ 3.3

Sau đây là một đoạn đối thoại ngắn được trích từ cuộc phỏng vấn với một trong các vị giám đốc sáng lập ra *CarMatch*. Mick Perez là phỏng vấn viên và Janet Hoffner là giám đốc.

> **Mick Perez**: Anh nói rằng những người muốn chia sẻ xe có thể đăng ký trở thành thành viên của *CarMatch* bằng cách gọi điện đến văn phòng và trao đổi với một nhân viên bất kỳ, sau đó các thông tin chi tiết của người này sẽ được nhân viên ấy nhập vào máy tính.
>
> **Janet Hoffner**: Vâng. Hoặc *franchisee* hoặc nhân viên văn phòng sẽ nhận cuộc điện và nhập các thông tin vào máy tính.
>
> **MP**: Ai là nhân viên văn phòng?
>
> **JH**: Có một hoặc hai thư ký, một tiếp tân và một nhân viên giám sát. Họ có một vai trò trong việc quản trị hệ thống.
>
> **MP**: Họ sẽ nhập những gì vào hệ thống?
>
> **JH**: Tên và địa chỉ khách hàng, các chi tiết về lộ trình cần chia sẻ, các sở thích mà họ có, chẳng hạn như không hút thuốc.
>
> **MP**: Đây có phải là cách duy nhất để nhập các thông tin này vào hệ thống hay không?
>
> **JH**: Không. Thông tin có thể được chuyển đến hệ thống từ web-server.
>
> **MP**: Các thông tin này sẽ được sử dụng như thế nào?
>
> **JH**: Có hai cách. Thứ nhất, nó được dùng để tìm kết hợp những thành viên tiềm năng. Thứ hai, nó được sử dụng để tạo ra một bản báo cáo quản lý cho *franchisee* biết số người đăng ký mỗi tuần và cách đăng ký.

Nếu chúng ta trả lời cho ba câu hỏi ở trên, chúng ta có thể xác định được *franchisee*, thư ký, tiếp tân và nhân viên giám sát là những người nhập thông tin vào hệ thống. *Franchisee* cũng có vai trò nhận thông tin từ hệ thống, và *web-server* là một hệ thống khác tương tác với hệ thống này. Cần lưu ý những người muốn chia sẻ xe không tương tác trực tiếp với hệ thống, vì vậy họ không phải là actor. Không cần phân biệt vai trò của thư ký, tiếp tân và nhân viên giám sát. Tất cả họ điều có quyền quản lý qui trình đăng ký của khách hàng, do đó ta có thể dùng *CarMatch Administrator* biểu diễn vai trò chung của họ. *Franchisee* cũng là một actor, nhưng có một vai trò riêng là nhận báo cáo quản lý. Vai trò actor cuối cùng là web-server, nó là một hệ thống khác tương tác với *CarMatch*.

Use case biểu diễn một chuỗi hành động được thực hiện bởi hệ thống, vì vậy để tìm được các use case chúng ta cần tìm kiếm các hành động của hệ thống. Trong ví dụ này có 4 use case khác nhau:



Hình 3.22: Các use case *và* ator *đầu tiên*

- Thêm trực tiếp một khách hàng mới (*Manually add car sharer*).
- Chuyển khách hàng từ web-server (*Transfer car sharer from web-server*).
- Kết hợp những người dùng chung xe (*Match car sharer*).
- Phát sinh báo cáo quản trị (*Produce management report*).

Chúng được minh họa trong hình 3.22. Lưu ý chúng ta không tách các use case này thành các mức chi tiết như mô tả trong ví dụ trên. Ở đây chúng ta nhóm chuỗi hành động lại với nhau thành các *đơn vị có ý nghĩa theo ngữ cảnh nghiệp vụ*. Chúng ta không tạo ra các use case có tên như *Take call* (nhận điện thoại) hoặc *Enter details* (nhập chi tiết). Các mô tả chi tiết nằm trong các đặc tả hành vi.

3.4.2 Sắp xếp các *use case* theo độ ưu tiên

Bước tiếp theo là sắp xếp các use case theo độ ưu tiên. Mục đích của bước này để bảo đảm các use case quan trọng được phát triển trước.

Ví dụ 3.4

Với các use case trong ví dụ trước, việc nhập các thông tin chi tiết của người dùng chung xe vào hệ thống sẽ diễn ra đầu tiên và kết hợp họ lại là việc quan trọng hơn so với việc tạo ra các bản báo cáo quản trị.

3.4.3 Phát triển use case (bắt đầu với các use case có độ ưu tiên cao)

Mục đích của hoạt động này là đưa ra đặc tả chi tiết của mỗi use case. Quá trình này cũng có thể tạo ra một số use case mới. Trong UP có một vai trò *staff* cho hoạt động *Use Case Specifiers*, là người tạo ra đặc tả chi tiết từng bước của các use case. Trong hầu hết các dự án, người phân tích là người thực hiện vai trò này.

Ví dụ 3.5

Dưới đây là một đoạn trích từ cuộc phỏng vấn với Janet Hoffner mô tả điều gì xảy ra khi nhập thành viên mới với nhiều chi tiết hơn.

> **Janet Hoffner**: Với khách hàng mới, dù chúng ta nhập thông tin trực tiếp hay chuyển về từ web-server đi nữa, thì quá trình này tách biệt với việc xử lý thanh toán phí thành viên. Nếu nhập trực tiếp, ta cần xử lý việc thanh toán ngay lúc đó. Còn nếu thông tin được lấy từ web-server, thì quá trình thanh toán phí sẽ được xử lý độc lập sau này. Khi thực hiện thanh toán, thành viên có thể thanh toán bằng ba hình thức: *Direct debit, Credit card* hoặc *Debit card*. Nếu hình thức là *Direct debit*, thì dữ liệu về cuộc thanh toán được lưu lại, và chúng ta sẽ chuyển một loạt những chi tiết thanh toán đến hệ thống ABTS vào cuối tháng.
>
> **Mick Perez**: ABTS là gì vậy?
>
> **JH**: Là viết tắt của *Automated Bank Transfer System*. Hệ thống này xử lý các cuộc thanh toán điện tử. Nếu đó là một cuộc thanh toán bằng thẻ thì nó được xử lý ngay tại chỗ.
>
> **MP**: Còn về quá trình kết hợp thành viên thì sao?
>
> **JH**: Quá trình này dựa vào một vài nhân tố, chủ yếu là nhân tố địa lý...

Tại điểm này, người phân tích sẽ bắt đầu tạo ra các mô tả use case để định nghĩa các use case dưới dạng hình thức. Đoạn ghi chép trong ví dụ 3.4 có thể được dùng để tạo ra mô tả use case như trong hình 3.9. Thông tin từ đoạn ghi chép trên cũng cung cấp cho phân tích viên một số điều:

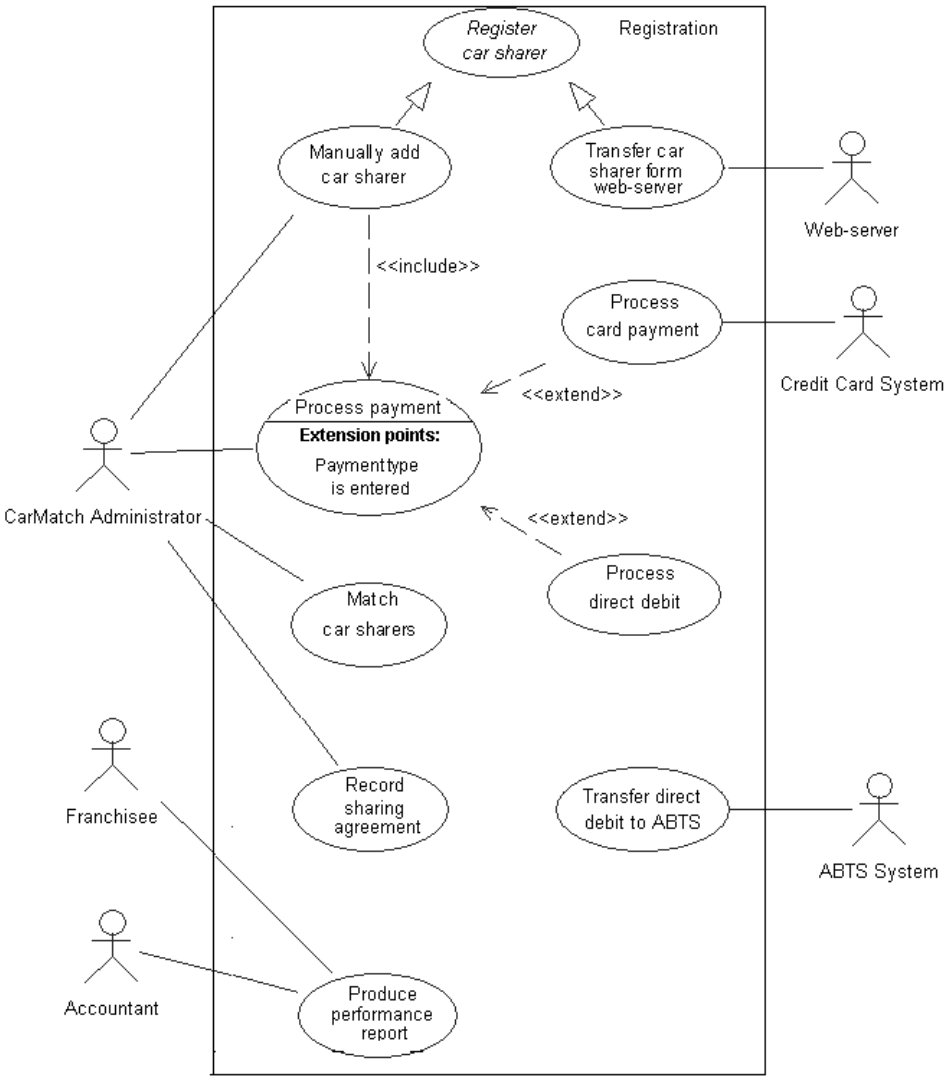
- Nhập trực tiếp và chuyển từ *web-server* là 2 phiên bản của cùng 1 sự việc.
- Khi nhập trực tiếp, các chi tiết thanh toán luôn được xử lý, nhưng điều này cần đến một use case riêng bởi vì thanh toán có thể xảy ra riêng.
- Use case *Process payment* có thể được mở rộng bằng hai cách khác nhau, thanh toán bằng thẻ hoặc ghi nợ ngân hàng.

Chúng sẽ được dùng để bổ sung cho cấu trúc của mô hình use case.

3.4.4 Lập cấu trúc cho mô hình use case

Trong hành động này, cấu trúc sẽ được lập bằng các quan hệ *generalization*, *include* và *extend* và bằng cách nhóm các use case thành các *package*.

Ví dụ 3.6



Hình 3.23: Phiên bản cấu trúc của biểu đồ use case

Có hai cách đăng ký khác nhau. Cả hai thực hiện cùng chức năng chính, nhưng một đáp ứng cho hành động nhập liệu của *CarMatch Administrator* qua giao diện người dùng, trong khi cách thứ hai lại đáp ứng cho dữ liệu về một *CarSharer* mới được chuyển từ web-server. Quan hệ *generalization* có thể được dùng để biểu diễn phần này của mô hình.

Process payment là một use case thực hiện chức năng riêng, nhưng nó cũng được bao hàm trong *Manually add car sharer*. Điều này được biểu diễn bằng quan hệ include. *Process payment* cũng được biểu diễn bằng một trong hai cách, tuỳ vào hình thức thanh toán. Điều này được biểu diễn bằng một quan hệ extend.

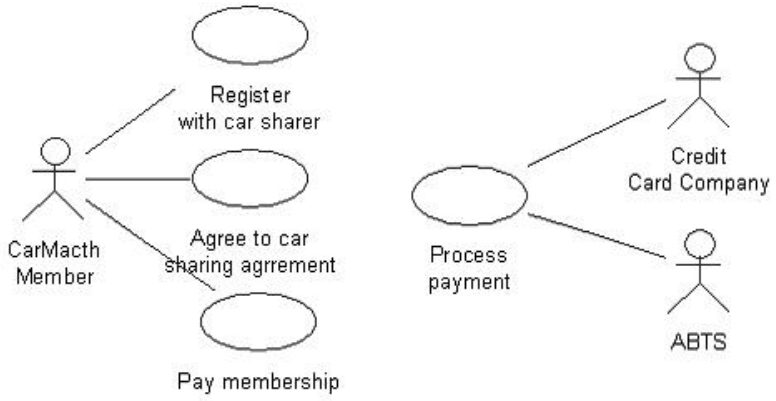
Lưu ý *Franchisee* là một trường hợp đặc biệt của *CarMatch Administrator*, và để biểu diễn chúng ta dùng quan hệ *generalization*.

Hình 3.23 là toàn bộ cấu trúc của biểu đồ use case cho hệ thống con *Registeration*.

3.5 Mô hình hoá nghiệp vụ với use case

UML có các *profile* đặc biệt có thể được dùng để áp dụng ký hiệu của UML cho các khía cạnh riêng của qui trình phát triển hệ thống. Một trong những *profile* được mô tả trong UML 1.3 là *profile* cho mô hình hoá nghiệp vụ. Trong *profile* này, các biểu đồ use case có thể được áp dụng cho việc lập mô hình nghiệp vụ. Các use case trong các ví dụ của chương này đều là các use case bên trong hệ thống có giao tiếp với các actor – người sử dụng trực tiếp hệ thống. Biểu đồ use case cũng biểu diễn các qui trình nghiệp vụ và tương tác giữa chúng với con người và hệ thống ngoài.

Ví dụ 3.7



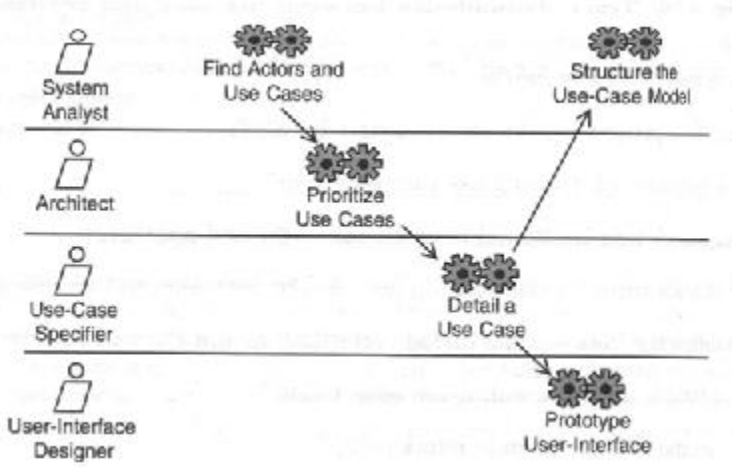
Hình 3.24: Biểu đồ use case profile *nghiệp vụ*

CarMatch như là một nghiệp vụ tương tác với một số tác nhân ngoài. Dựa vào thông tin trong chương 1 chúng ta xác định được các tác nhân này, đó là *CarMatch Member*, *Credit card company* và *ABTS*. Các use case trong mô hình nghiệp vụ là các chức năng mà các tác nhân này sẽ tương tác với chúng. Ví dụ, *CarMatch Member* tương tác với các use case nghiệp vụ *Register with CarMatch*, *Agree to car sharing arrangement* và *Pay membership* (hình 3.24).

3.6 Quan hệ với các biểu đồ khác

Như đã trình bày trong phần 3.3.2, hành vi của một use case có thể được xác định bằng cách sử dụng một biểu đồ UML khác: biểu đồ cộng tác, biểu đồ tuần tự hoặc biểu đồ trạng thái. Trong một công cụ CASE, điều này được xử lý bằng một *siêu liên kết* (hyperlink) từ use case đến biểu đồ liên quan đặc tả hành vi của nó. Ngoài ra, giống như tất cả các mô hình UML khác, mô hình UML được tổ chức thành các package như một phần của khung nhìn *Model Management*.

3.7 Các use case trong UP



Hình 3.25: Requirements Workflow

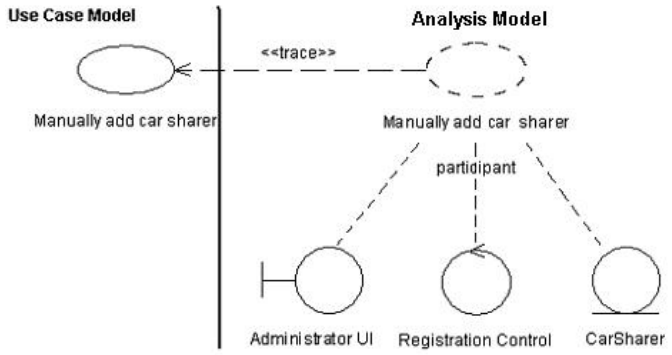
Jacobson và hầu hết các chuyên gia khác (1999) mô tả UP như là *use-case-dirven* (xem mục 2.8) và các use case là trung tâm cho phương pháp trong đó các yêu cầu của người dùng được xác định và làm tài liệu.

Trong chương 2 chúng ta đã biết UP gồm có 5 *workflow*. Mỗi *workflow* gồm nhiều hoạt động có liên quan do các *worker* khác nhau thực hiện. Bằng cách thực hiện các hoạt động trong mỗi *workflow*, các *worker* tạo

ra các mô hình. Trong UP, *workflow* đầu tiên là *Requirements Workflow*, và *Use Case Model* là một trong các đầu ra của nó.

Hình 3.25 trình bày một *workflow* như một biểu đồ hoạt động (xem chương 10) bằng cách dùng các biểu tượng *stereotype* (xem chương 14) để biểu diễn các *worker* và các hoạt động. Trong hình này có một hoạt động chưa được thảo luận đó là *Prototype User-Interface*. Hoạt động này liên quan đến việc tạo ra một phác thảo ban đầu (*prototype*) của giao diện người dùng cho các use case. Nó không dùng ký hiệu UML, nhưng lại là một phần của dòng công việc UP.

Trong UP, use case được *hiện thực* (realize) bởi *cộng tác* (collaboration) với một phụ thuộc *trace*. Phụ thuộc *trace* là một phụ thuộc đặc biệt giữa các phần tử của các mô hình khác nhau cho phép lần theo các yêu cầu cho đến cài đặt cuối cùng. Nó được chỉ ra trong biểu đồ bằng stereotype <<*trace*>>. *Trace* tượng trưng cho một quan hệ giữa các phần tử của hai mô hình hoặc liên quan đến lịch sử hoặc dựa trên một số qui trình đưa ra dẫn xuất. Hình 3.26 biểu diễn phụ thuộc *trace* giữa use case *Manually add carsher* và *hiện thực* của nó. (Trong hình, UI là viết tắt của *User Interface* – giao diện người dùng). Ký hiệu phụ thuộc sẽ được giải thích trong chương 7.



Hình 3.26: Phụ thuộc trace *giữa use case và hiện thực của nó*

Câu hỏi ôn tập

3.1 *Use case* là gì ?

3.2 Ký hiệu của *use case* là gì?

3.3 *Actor* là gì?

3.4 Ký hiệu của *actor* là gì?

3.5 *Screnario* là gì?

3.6 Trong một dự án, giai đoạn nào các *use case* sẽ được sử dụng đầu tiên?

3.7 Trình bày mục đích của biểu đồ use case?

3.8 Giữa các *actor*, những loại kết hợp nào được phép dùng?

3.9 Giữa các *use case*, các loại kết hợp và quan hệ nào được phép dùng?

3.10 Trình bày sự khác biệt giữa quan hệ *include* và quan hệ *extend*.

3.11 Ký hiệu của kết hợp *generalization* là gì?

3.12 Ký hiệu của quan hệ *include* là gì?

3.13 Ký hiệu của quan hệ *extend* là gì?

3.14 *Extension point* là gì?

3.15 Trong UP, các use case được phát triển trong *workflow* nào?

3.16 Các hoạt động UP nào tạo ra các use case?

Bài tập có lời giải

3.1 Trong hệ thống bảo hiểm của *CarMatch*, nhân viên sẽ tìm kiếm các hợp đồng thích hợp cho một thành viên nào đó dựa vào tuổi, nghề nghiệp và nơi sinh sống của người này. Rồi họ sẽ giới thiệu cho thành viên này mua. Những use case nào có liên quan ở đây? Nên gọi chúng là gì?

Lời giải: Có ba hoạt động được thực hiện bởi nhân viên dùng hệ thống này: tìm kiếm các hợp đồng, giới thiệu các hợp đồng cho thành viên và bán hợp đồng. Tên của các use case nên giữ cho đơn giản, có thể là: *Search for Policy*, *Recommend Policy* và *Sell Policy*.

3.2 Trong hệ thống *CarMatch*, sẽ có một *Insurance Supervisor* (nhân viên giám sát bảo hiểm) và một *Insurance Asssistant* (nhân viên trợ lý bảo hiểm) trong mỗi văn phòng chi nhánh (CarMatch địa phương). Vai trò của họ là bán hợp đồng bảo hiểm cho thành viên của *CarMatch*. Họ sẽ dùng chung các use case trong hệ thống. Có bao nhiêu actor liên quan và tên của các actor này nên đặt là gì?

Lời giải: Do cả giám sát và trợ lý đều dùng chung các use case, nên chúng ta không cần tách họ thành hai actor, chỉ cần một actor là đủ. Có thể đặt tên cho actor này là *Insurance Administrator*.

3.3 Sau đây là một đoạn trích từ cuộc phỏng vấn với Janet Hoffner:

> **Janet Hoffner**: Khi chúng ta bán bảo hiểm, việc làm đầu tiên là thu thập các chi tiết của thành viên, bao gồm tuổi tác và nghề nghiệp của họ, địa chỉ và lịch sử bảo hiểm – nghĩa là trong quá khứ người này có bị tai nạn gì hay không.
>
> **Mick Perez**: Lấy các thông tin này từ đâu?

JH: Một số thông tin lấy từ chi tiết thành viên trong hệ thống, và thông qua điện thoại.

MP: Chuyện gì sẽ xảy ra nữa?

JH: Chúng ta sẽ cố gắng tìm ra một hợp đồng thích hợp. Chúng ta sẽ tìm kiếm một hợp đồng tốt nhất cho họ dựa vào thông tin mà chúng ta có. Hệ thống có thể đưa ra nhiều điều khoản hợp đồng, chúng ta có nhiệm vụ giới thiệu những điều khoản nào sát thực với yêu cầu của thành viên.

MP: Anh luôn bán được hợp đồng chứ?

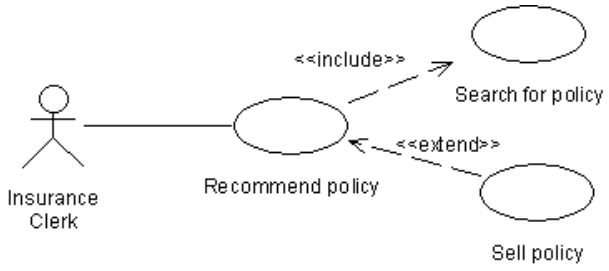
JH: Không, thỉnh thoảng có người quyết định mua hợp đồng, nhưng đôi khi không mua.

MP: Nghĩa là anh có nhiệm vụ đưa ra điều khoản hợp đồng thích hợp với người đó. Anh luôn luôn thực hiện việc tìm kiếm một điều khoản hợp đồng như vậy, và thỉnh thoảng bán được hợp đồng.

JH: Vâng, đúng vậy.

Hãy dùng quan hệ *extend*, *include* để biểu diễn cho các quan hệ giữa ba use case này.

Lời giải: Các yếu tố chính ở đây là toàn bộ tác vụ đều dùng để đưa ra một điều khoản hợp đồng cho một thành viên, trong đó có chức năng tìm kiếm các điều khoản thích hợp. Chúng ta tạo ra use case *Search for policy* với tư cách là một use case độc lập, nó được liên kết với use case *Recommend policy* thông qua quan hệ *include*, theo đó *Recommend policy* luôn luôn bao gồm *Search for policy*. Tuy nhiên, use case *Sell policy* lại hiếm khi xảy ra, cho nên liên kết giữa nó và *Recommend policy* là một quan hệ *extend*, (xem hình 3.27). Chúng ta phải lưu ý đến chiều mũi tên của hai quan hệ này.



Hình 3.27 Các use case cho việc bán bảo hiểm

3.4 Theo bạn, điều kiện tại điểm mở rộng cho *Sell policy* là gì?

Lời giải: Điều kiện sẽ là *đồng ý mua bảo hiểm* (Member agrees to purchase policy). Ở giai đoạn này chúng ta không biết cách cài đặt điều kiện này. Sau này nó có thể được xác định dưới dạng một phần của dữ liệu và được nhập vào trong một *trường* (field) trên màn hình.

3.5 Dưới đây là các yêu cầu khác của hệ thống con *Insurance*. Hãy vẽ biểu đồ use case biểu diễn chúng, cùng với các use case từ các bài tập trên.

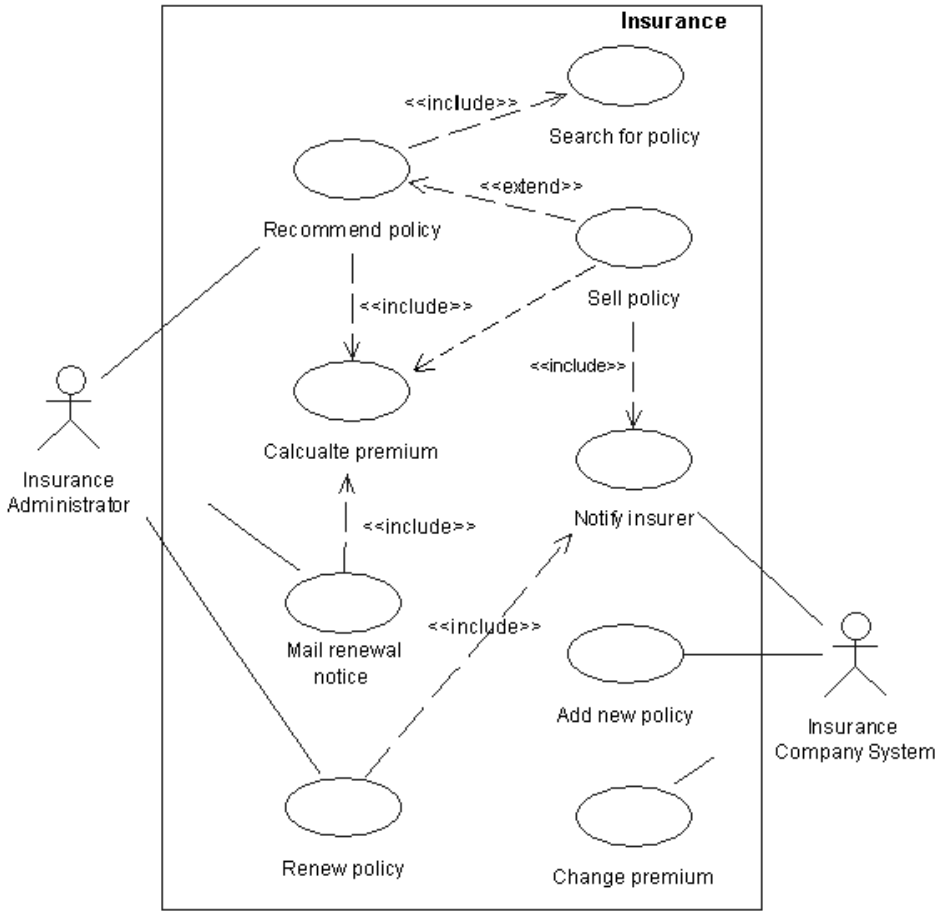
- Báo cho hệ thống công ty bảo hiểm tất cả các hợp đồng đã bán.
- Nhận thông báo từ hệ thống công ty bảo hiểm về các hợp đồng mới và điều kiện thực hiện.
- Nhận thông báo về các thay đổi phí bảo hiểm từ hệ thống công ty bảo hiểm.
- Phát sinh các thông báo gia hạn định kỳ hàng tuần cho tất cả các hợp đồng một tháng trước khi đến hạn.
- Gia hạn một hợp đồng.
- Thông báo cho hệ thống công ty bảo hiểm về tất cả các hợp đồng được gia hạn.
- Tính phí bảo hiểm cho các hợp đồng giới thiệu, đã bán và được gia hạn.
- Lưu ý tất cả việc truyền tải thông tin giữa hệ thống công ty bảo hiểm và *CarMatch* được dự định truyền tự động giữa hai hệ thống máy tính.

Lời giải: Biểu đồ có thể được vẽ như trong hình 3.28. Lưu ý, giải pháp này giả sử việc thông báo cho hệ thống công ty bảo hiểm xảy ra như là một phần của các use case *Sell policy* và *Renew policy* (gia hạn hợp đồng). Trong trường hợp nếu use case *Notify insurer* (thông báo) xảy ra sau đó (một tuần một lần vào ngày thứ sáu), thì các quan hệ *include* từ *Sell policy* và *Renew policy* đến *Notify insurer* không cần phải có.

Bài tập bổ sung

3.6 Ba vấn đề đầu tiên áp dụng cho hệ thống con tư vấn của *CarMatch*. Hãy vẽ một biểu đồ use case biểu diễn actor *Consultant* (vai trò tư vấn) với một kết hợp tới từng use case *Record consultancy visit* (ghi lại cuộc viếng thăm tư vấn) và *Record consultancy expenses* (ghi lại chi phí tư vấn).

3.7 Hãy thêm vào actor *Senior Consultant* (trưởng tư vấn) và một kết hợp *generalization* giữa *Consultant* và *Senior Consultant* vào biểu đồ trên. *Senior Consultant* có thể làm những gì mà *Consultant* có thể làm trên các use case *Initiate consultancy project* (đề xuất dự án tư vấn), *Conclude consultancy project* (ký kết dự án tư vấn) và *Invoice customer* (lập hoá đơn khách hàng). Hãy bổ sung các use case này.



Hình 3.28: Các use case của hệ thống con Insurance

3.8 Use case *Record consultancy visit* đôi khi được mở rộng từ use case *Record consultancy expenses*. Còn *Conclude consultancy project* luôn bao gồm use case *Invoice customer*. Hãy thêm các quan hệ cần thiết vào biểu đồ từ các thông tin trên. (Lưu ý đến hướng của các quan hệ).

3.9 Đọc case study *Volbank* trong chương 1. (Tất cả những vấn đề sau đây sẽ dùng thông tin trong case study này. Vấn đề sau được xây dựng dựa trên các vấn đề trước đó). Hãy liệt kê các actor có liên quan trong hệ thống *Volbank*.

3.10 Liệt kê các use case theo bạn nghĩ có thể có trong hệ thống *VolBank*.

3.11 Những actor nào liên kết với những use case nào. (Lưu ý rằng mỗi actor nên kết hợp với ít nhất một use case, và mỗi use case nên kết hợp với ít nhất một actor).

3.12 Hãy liệt kê độ ưu tiên của các use case.

3.13 Sau đây là một trích từ cuộc phỏng vấn giữa Said Hussain (phân tích viên hệ thống) và Martin Page (giám đốc tuyển dụng của *VolBank*).

Said Hussain: Làm thế nào để lấy được thông tin của các tình nguyện viên và đưa vào hệ thống?

Martin Page: Hầu hết tình nguyện viên sẽ gọi điện đến một tổ chức tình nguyện địa phương, từ đó thông tin của họ sẽ được một trong các nhân viên của tổ chức chúng tôi ghi nhận, một số người khác lại gọi trực tiếp đến *VolBank*, và số khác sẽ truy cập vào web-server.

SH: Còn thời gian tình nguyện thì sao?

MP: Họ có thể cung cấp thông tin ngay khi đăng ký hoặc sau đó đều được.

SH: Nghĩa chúng ta lấy được dữ liệu từ web-server hoặc dữ liệu do nhân viên nhập bằng tay vào máy tính.

Hãy phát triển và lập cấu trúc cho biểu đồ use case của bạn dựa vào thông tin trên.

3.14 Dưới đây là một số thông tin thêm về *VolBank*. Dựa vào thông tin này hãy hoàn chỉnh biểu đồ use case.

Một số tổ chức tình nguyện địa phương sẽ có hệ thống máy tính của riêng họ, thông tin chi tiết về nhu cầu sẽ được chuyển đến hệ thống máy tính của *VolBank*. Các tổ chức tình nguyện địa phương không có hệ thống máy tính riêng sẽ gởi cho *VolBank* bằng cách gọi điện, hoặc bằng cách điền vào biểu mẫu và gởi đến *VolBank* qua bưu điện.

Có hai quá trình cho việc kết hợp các tình nguyện viên và các nhu cầu cần được giúp đỡ. Quá trình thứ nhất xảy ra khi có tình nguyện viên mới, nó kết hợp tình nguyện viên với các nhu cầu chưa được thực hiện. Quá trình thứ hai xảy ra khi có một nhu cầu mới được đăng ký, nó kết hợp nhu cầu này với các tình nguyện viên dựa theo thời gian rỗi của họ. Trong cả hai trường hợp, việc kết hợp được thực hiện chủ yếu là dựa trên vị trí địa lý của cả tình nguyện viên và nhu cầu, và bằng cách đối chiếu các kỹ năng

cũng như sở thích giữa tình nguyện viên và những người có nhu cầu cần giúp đỡ.

Một khi một tình nguyện viên được kết hợp với một nhu cầu, tình nguyện viên này sẽ được thông báo, và nếu anh ta đồng ý thì tổ chức cần giúp đỡ cũng sẽ được thông báo sau đó. Nếu họ chấp nhận tình nguyện viên này, thì một *ghi chép* sẽ được thực hiện trong hệ thống để ghi lại một kết hợp thành công.

Giám đốc tuyển dụng yêu cầu một báo cáo thống kê về số tình nguyện viên, cách tình nguyện viên đăng ký, thời gian họ gởi là bao nhiêu và đã sử dụng hết bao nhiêu.

3.15 Hãy viết mô tả use case cho mỗi use case vừa tìm được cho hệ thống *VolBank*.

Chương 4
BIỂU ĐỒ LỚP, LỚP VÀ MỐI KẾT HỢP

4.1 Mở đầu

Biểu đồ lớp mô tả các *lớp*, là các *viên gạch* để xây dựng bất kỳ hệ thống hướng đối tượng nào. Khả năng cộng tác giữa chúng, bằng cách *truyền thông điệp*, được chỉ ra trong các mối quan hệ giữa chúng.

Các khái niệm có thể được mô tả qua các ký hiệu của biểu đồ lớp là khá rộng. Với những người không chuyên, các khái niệm và ký hiệu của biểu đồ lớp là không dễ hiểu như biểu đồ use case (chương 3) hoặc biểu đồ hoạt động (chương 10). Chương này trình bày các thành phần căn bản, các chương sau chúng ta sẽ giới thiệu sâu hơn về các thành phần khác.

Trong UML, các khái niệm được biểu diễn trong biểu đồ lớp là phần tử của gói *Core* (nòng cốt) là một gói nằm trong gói *Foundation* (cơ sở) – xem chương 2. Ý muốn nói biểu đồ lớp tạo thành một phần chính yếu và nền tảng của bất kỳ mô hình hướng đối tượng nào được tạo ra bằng cách dùng UML.

Biểu đồ lớp cho ta một khung nhìn tĩnh của các lớp trong mô hình hoặc một phần của mô hình. Nó chỉ cho ta thấy các thuộc tính và các thao tác của lớp, cũng như các loại quan hệ giữa các lớp. Biểu đồ lớp giống như một tấm bản đồ, với các lớp là các thành phố còn các mối quan hệ là các đường nối giữa chúng.

Các tương tác và cộng tác, thực sự xảy ra để hỗ trợ cho bất kỳ một yêu cầu chức năng nào, biểu diễn một *tuyến* nào đó trên bản đồ. *Tuyến* này sẽ *len lỏi* (navigate) từ lớp này sang lớp khác thông qua các quan hệ giữa chúng. Lưu ý rằng, chỉ có thể thông thương giữa hai điểm bằng cách dò theo các quan hệ hợp lệ từ điểm bắt đầu, đi qua các điểm trung gian và đến đích.

Biểu đồ lớp tự nó không mô tả một *tuyến* đi cụ thể để thoả mãn một use case. Nó biểu diễn tất cả các *điểm* sẵn có và các *tuyến* có thể có giữa chúng. Một *tuyến* thực sự sẽ được chỉ ra trong các biểu đồ tương tác, như biểu đồ tuần tự (chương 9) hoặc biểu đồ cộng tác (chương 8).

Để làm ví dụ, biểu đồ lớp trong hình 4.1 của hệ thống ngân hàng cho ta thấy cấu trúc tĩnh của mô hình. Trong thể hiện này, biểu đồ chỉ mô tả tên lớp và các quan hệ giữa các lớp. Lớp khách hàng *Customer* biểu diễn cho một *kiểu chung* (general type) hoặc *lớp chung* (general class), chúng có nhiều *đối tượng* (object) hoặc *thể hiện* (instance). Kết hợp giữa lớp khách hàng *Customer* và lớp tài khoản *Account* xác định có một số loại cộng tác giữa hai lớp này. Dựa vào nhãn trên kết hợp chúng ta có thể thấy bản chất của kết hợp này là *một khách hàng sẽ giữ* (hold) *một tài khoản*.

Bằng cách nghiên cứu biểu đồ lớp của hình 4.1, bạn cũng có thể hiểu được các khía cạnh khác của ký hiệu. Tuy nhiên, như đã đề cập, tập ký hiệu hoàn chỉnh cho các biểu đồ lớp khá rộng, các chương từ 4 đến 7 sẽ giới thiệu các ký hiệu còn lại.

Trong chương này, chúng tôi chỉ giới thiệu ký hiệu biểu đồ lớp cơ bản trong biểu diễn các lớp, các đặc điểm của chúng và các kết hợp cơ bản giữa chúng. Các chương sau sẽ giới thiệu các ký hiệu chi tiết hơn và các khía cạnh khác sâu hơn.

4.2 Biểu đồ lớp qua qui trình phát triển

Các lớp được chỉ ra trên biểu đồ tuỳ thuộc vào giai đoạn của qui trình phát triển và mức độ chi tiết đang được xem xét.

Trong giai đoạn phân tích, *các lớp hiển nhiên* thuộc phạm vi hệ thống là các lớp cần được quan tâm nhất. Giai đoạn này các *stakeholder* (những người nắm các quy tắc quản lý của hệ thống) có thể nhận ra rất nhiều lớp. Ở giai đoạn này *các lớp được sử dụng trong cài đặt một giải pháp* là các lớp ít được quan tâm hơn, hoặc là các lớp chưa được xét đến.

Khi qui trình phát triển chuyển sang giai đoạn thiết kế, các lớp và các cấu trúc quan hệ phản ánh mô hình giải pháp sẽ được giới thiệu. Các lớp này không phải là một phần của việc tìm hiểu cộng đồng người dùng cuối, nhưng chúng lại cần thiết để tạo ra một mô hình logic và một mô hình có cấu trúc tốt.

4.3 Mục đích của kỹ thuật

Như đã đề cập bên trên, biểu đồ lớp được sử dụng khắp nơi trong qui trình phát triển. Nó biểu diễn cấu trúc tĩnh của các lớp tạo nên hệ thống hoặc hệ thống con. Cấu trúc tĩnh của các lớp bao gồm các lớp đang xem xét cùng các đặc trưng của chúng, là thuộc tính và thao tác. Chúng sẽ cung cấp các khả năng để thực hiện một phần các yêu cầu chức năng hệ thống.

Biểu đồ lớp không chỉ ra cách các thành phần của mô hình lớp tương tác với nhau, đó là kỹ thuật của biểu đồ tuần tự hoặc của biểu đồ cộng tác.

Biểu đồ lớp minh họa nhiệm vụ quản lý hành vi và dữ liệu của mỗi lớp, và cách chuyển giao các trách nhiệm này trong mô hình lớp. Nó không cho thấy các yêu cầu chức năng của hệ thống, hoặc hệ thống con, từ góc nhìn của người dùng cuối. Đó là mục đích của biểu đồ use case (chương 3).

Nói đúng ra, đặc tả ký hiệu UML (OMG, 1999b) dùng để mô tả các biểu đồ cấu trúc tĩnh. Tuy nhiên khi đặc tả ký hiệu ra đời, thì thuật ngữ *biểu đồ lớp* được dùng rộng rãi bởi nó ngắn gọn và dễ hiểu hơn. Sau đây là các mục đích của việc tạo ra biểu đồ lớp:

- Dùng để sưu liệu các lớp tạo thành hệ thống hoặc hệ thống con.
- Dùng để mô tả các kết hợp, quan hệ tổng quát hoá và quan hệ kết tập (*aggregation*) giữa các lớp.
- Dùng để biểu diễn các thành phần của lớp, chủ yếu là các thuộc tính và các thao tác của mỗi lớp.
- Chúng có thể được dùng trên khắp qui trình phát triển, từ đặc tả của các lớp trong xác định yêu cầu đến mô hình cài đặt cho một hệ thống nào đó, để biểu diễn cấu trúc lớp của hệ thống đó.
- Làm sưu liệu cách tương tác giữa các lớp của hệ thống với các thư viện lớp đang có.
- Dùng để biểu diễn các thể hiện đối tượng riêng lẻ bên trong cấu trúc lớp.
- Dùng để biểu diễn các giao diện được một lớp nào đó hỗ trợ.

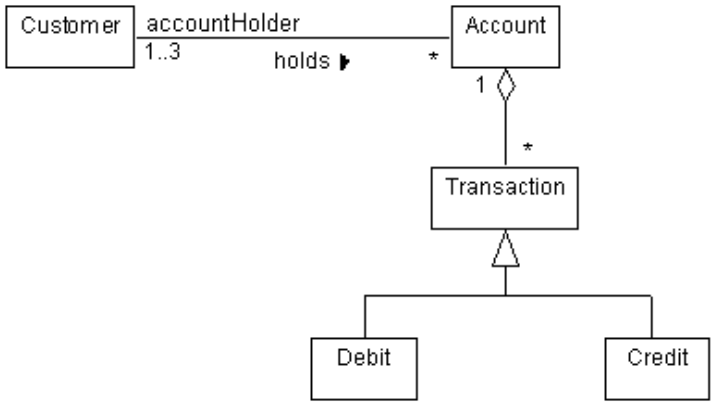
4.4 Biểu đồ lớp – các ký hiệu cơ bản

Mô hình lớp cho một hệ thống có thể rất phức tạp với rất nhiều lớp và rất nhiều quan hệ giữa chúng. Sự phức tạp của mô hình lớn như thế có thể được quản lý bằng cách phân đoạn trên nhiều biểu đồ lớp khác nhau. Một biểu đồ lớp có thể tương ứng một hệ thống (hệ thống con), một gói hoặc một mô hình nào đó. Nói cách khác, một biểu đồ lớp có thể biểu diễn các quan hệ hiện có giữa các lớp, trong các hệ thống (hệ thống con), các gói hoặc các mô hình khác nhau. Bất kỳ quan hệ ngữ cảnh nào giữa một biểu đồ lớp với một hệ thống (hệ thống con), một gói hoặc một mô hình đều phải được xác nhận một cách rõ ràng bởi người phát triển, là người tạo ra biểu đồ lớp trong tài liệu liên quan.

4.4.1 Lớp (class)

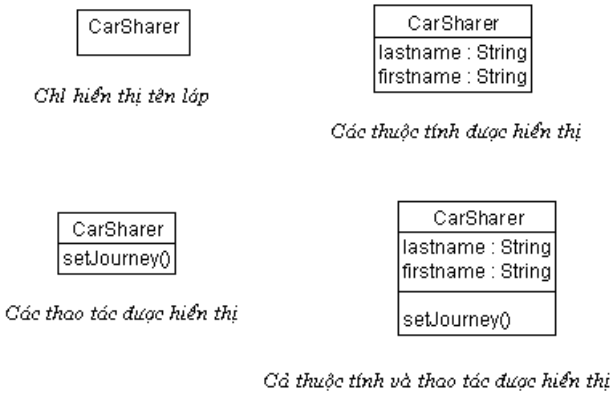
Là khối xây dựng cơ bản cho các biểu đồ lớp. Lớp được biểu diễn bằng một hình chữ nhật, tên của lớp được đặt ở giữa hình chữ nhật. Tên lớp

nên bắt đầu bằng ký tự hoa. Thông thường tên lớp không có khoảng trắng giữa các từ, và nên viết hoa chữ đầu của mỗi từ. Ví dụ, *BankAccount* không là *Bank account*; *CustomerInvoice* không là *Customerinvoice*. Dạng tối thiểu của lớp (chỉ gồm tên lớp) được minh hoạ trong hình 4.1.



Hình 4.1: Ví dụ về biểu đồ lớp của hệ thống ngân hàng

Mỗi biểu tượng lớp có thể có các ngăn tách biệt chứa các đặc trưng của lớp như thuộc tính, thao tác và các đặc trưng khác nếu có. Mỗi ngăn chứa một danh sách các đặc trưng cùng loại. Các ngăn có thể được lược bỏ một cách độc lập. Nói cách khác, một lớp có thể gồm:



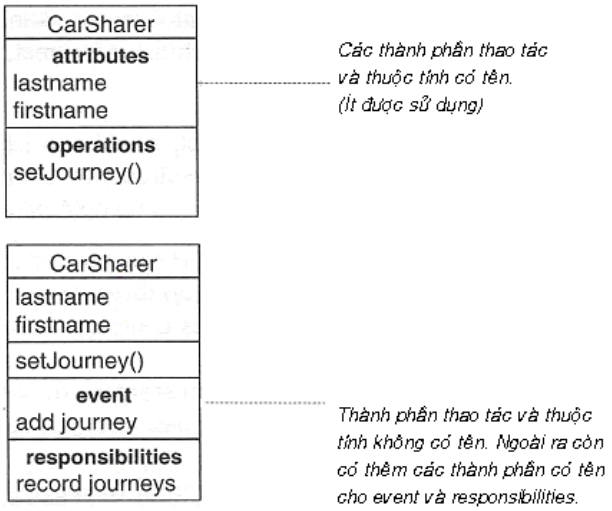
Hình 4.2: Lớp với các ngăn

- Chỉ có tên lớp.
- Tên lớp và ngăn chứa danh sách các thuộc tính.
- Tên lớp và ngăn chứa danh sách các thao tác.

- Tên lớp, ngăn chứa danh sách các thuộc tính và ngăn chứa danh sách các thao tác.

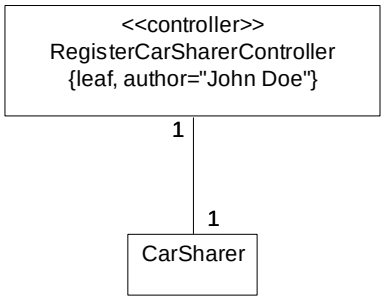
Bốn trường hợp này được minh họa trong hình 4.2.

Mỗi ngăn có thể có tên, riêng tên của các ngăn *thuộc tính* (attributes) và *thao tác* (operations) thường được bỏ qua. Ngoài ra còn có ngăn *biến cố* (events), ngăn *nhiệm vụ* (responsibilities). Xem hình 4.3.



Hình 4.3: Các ngăn được đặt tên

Trong hình 4.4, ngăn chứa tên lớp còn được dùng để chỉ rõ stereotype của lớp và các tính chất, như danh sách các *giá trị được gán nhãn* bên trong cặp dấu ngoặc móc ({..}, xem chương 14). Trong ngữ cảnh biểu đồ lớp, các stereotype thường được dùng để làm rõ hành vi của lớp; hình 4.4 cho biết *RegisterCarSharerController* phù hợp với mẫu hành vi được hiểu bởi stereotype <<*controller*>>.



Hình 4.4: Thông tin được chỉ ra trong ngăn chứa tên lớp

> **Nói riêng**: *Một lớp <<controller>> có thể được dùng để kết hợp sự tương tác giữa các lớp, trong mô hình các lớp chủ yếu của một ứng dụng, với giao tiếp của ứng dụng đó. Việc lập cấu trúc cài đặt theo cách này sẽ che giấu việc cài đặt giao diện ra khỏi sự thay đổi của mô hình lớp. Tương tự, các giao diện khác nhau có thể được làm mà không đòi hỏi phải cấu trúc lại mô hình lớp.*

Các tính chất của lớp cung cấp thông tin mô tả lớp cho người có liên quan trong qui trình lập mô hình. Các tính chất của lớp được liệt kê trong cặp dấu ngoặc móc, có dạng như sau: {<*danh sách tính chất*>}. Các tính chất của lớp có thể là:

- Thông tin quản lý việc lập mô hình như: *tác giả, ngày tạo, ngày sửa chữa cuối cùng* và *trạng thái*.
- Thông tin lớp với kiểu dữ liệu logic như: *là trừu tượng* (isAbstract)*, là lá* (isLeaf)*, là gốc* (isRoot).

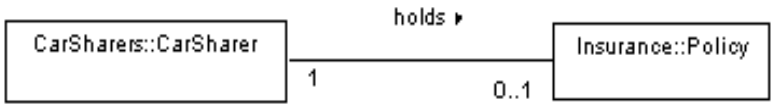
Các lớp trừu tượng không bao giờ có thể hiện, nghĩa là không bao giờ có một đối tượng thể hiện của một lớp trừu tượng, khi đặt *isAbstract* = *false* tính chất trừu tượng được loại bỏ.

Tính chất *isLeaf* được dùng để xác định một lớp có được tạo lớp con thông qua cấu trúc tổng quát hoá hay không. Chương 5 sẽ thảo luận về tổng quát hoá. Đặt *isLeaf* = *true* thì lớp không được chuyên biệt hoá và ngược lại. Nếu tính chất *isLeaf* bị bỏ qua thì mặc định *isLeaf* = *false*.

Tính chất *isRoot* được sử dụng để xác định một lớp có phải là một chuyên biệt hoá của một lớp khác hay không. Gán *isRoot* = *true* thì lớp không phải là một chuyên biệt hoá của một lớp khác và ngược lại. Nếu tính chất *isRoot* bị bỏ qua thì mặc định *isRoot* = *false*..

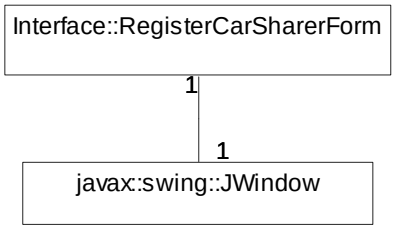
Một ký hiệu nữa là khả năng chứa tên đường dẫn hoặc cấu trúc gói của lớp. Ký hiệu này rất có ích trong trường hợp cần làm rõ nguồn gốc của lớp. Tên lớp đầy đủ có dạng <*tên đường dẫn hoặc tên gói*>::<*tên lớp*>. Ta dùng cặp dấu hai chấm (::) để phân cấp tên đường dẫn, ví dụ *java::sql::connection*. Tên đường dẫn được gọi là *namespace* của một lớp. Trong một *namespace*, tên lớp không được trùng nhau.

Ký hiệu này rất hữu ích nếu các nguyên lý quản lý mô hình được sử dụng để tổ chức một mô hình lớn vào trong các gói (hoặc gói con). Một số biểu đồ lớp cần trình bày các kết hợp giữa các lớp từ những gói khác nhau. Hình 4.5 biểu diễn tên đường dẫn lớp được dùng để biểu diễn hai lớp từ các gói khác nhau.



Hình 4.5: Cách sử dụng tên đường dẫn của lớp

Hình 4.6 được dùng để biểu diễn tên đường dẫn lớp được dùng để truy xuất tới một lớp từ một thư viện lớp bên ngoài (trong hình là thư viện giao diện *Java Swing*).

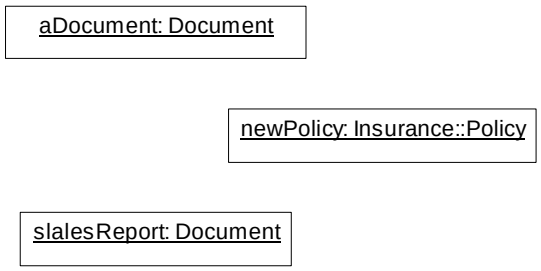


Hình 4.6: Dùng đường dẫn để truy xuất tới thư viện lớp ngoài

4.4.2 Thể hiện đối tượng

Biểu đồ lớp UML cho phép biểu diễn các đối tượng, là các thể hiện của lớp. Trong chương này, chúng tôi chỉ giới thiệu một vài ký hiệu đối tượng cơ bản nhất, chi tiết về ký hiệu của các thể hiện lớp được trình bày trong chương 7.

Ký hiệu đối tượng cơ bản rất giống với ký hiệu lớp cơ bản. Thành phần tên trình bày tên đối tượng và kiểu lớp của nó. Tên của đối tượng có dạng như sau *instanceName:ClassName*, tất cả đều được gạch chân. Ví dụ: *aDocument:Document*, *saleReport:Document*, *newPolicy:Insurance::Policy* (xem hình 4.7).



Hình 4.7: Các thể hiện đối tượng

4.4.3 Thuộc tính và thao tác

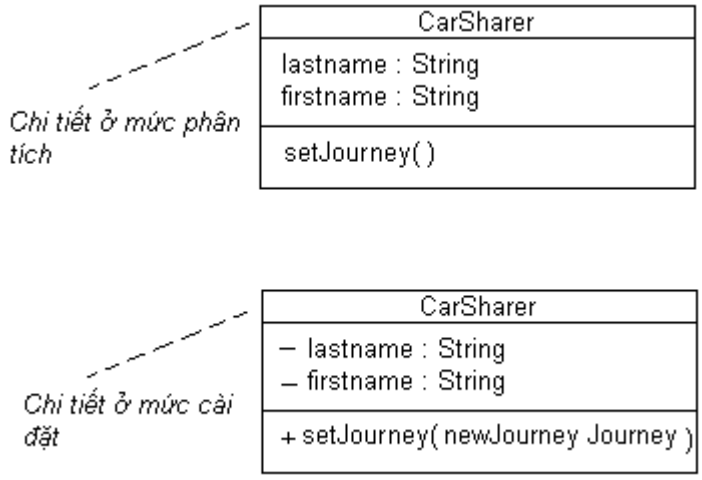
Hai thành phần thông dụng nhất của lớp là danh sách thuộc tính và danh sách thao tác. Chúng được liệt kê từng dòng một, và có thể nối thêm vào sau một danh sách các tính chất, dưới dạng danh sách các *giá trị có nhãn* và được đặt trong cặp dấu ngoặc móc.

Tên thuộc tính và thao tác nên bắt đầu bằng một ký tự thường. Ngoại trừ thao tác khởi tạo, tên phải trùng với tên lớp từng ký tự một - kể cả chữ hoa và chữ thường. Thao tác khởi tạo không phải lúc nào cũng được hiển thị trong biểu đồ lớp, ta xem như tồn tại mặc định. Theo qui ước, tên không có khoảng trắng giữa các từ. Thay vào đó, các từ được phân biệt bằng một ký tự hoa đầu mỗi từ. Ví dụ, là *setSharingAgreement* không là *SetSharingAgreement*, là *addRequirement* không là *add requirement*.

Mức chi tiết của thuộc tính và thao tác tuỳ vào biểu đồ lớp được sử dụng ở giai đoạn phát triển nào. Ở giai đoạn phân tích, chỉ cần hiển thị hoặc cho biết thông tin tổng quát. Ví dụ, các tham số của thao tác chưa được thiết lập một cách rõ ràng. Ở giai đoạn thiết kế, mức độ chi tiết là cần thiết để việc cài đặt có thể được hoàn tất. Ví dụ, không những cần đến các tham số của thao tác mà còn cần đến kiểu dữ liệu của mỗi tham số.

4.4.3.1 Kiểu thuộc tính và kiểu thao tác

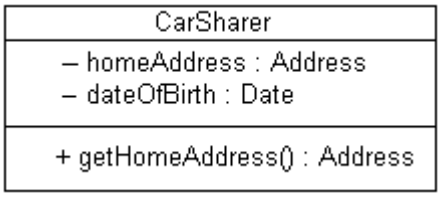
Ví dụ về các biểu diễn ở mức phân tích và cài đặt đối với thuộc tính và thao tác được minh họa trong hình 4.8. Cả thuộc tính và thao tác đều có dạng đơn giản <*featureName*>:<*type*>, trong đó *type* là kiểu dữ liệu của thuộc tính hoặc kiểu dữ liệu trả về của thao tác.



Hình 4.8: Danh sách thao tác và thuộc tính

Việc bỏ qua kiểu trả về của một thao tác (hình 4.8) sẽ có các ý nghĩa khác nhau, tuỳ vào định hướng mà lớp được vẽ. Điều này sẽ được thảo luận trong mục 4.5.

Kiểu của thuộc tính và thao tác có thể là *kiểu lớp* lấy từ thư viện lớp của môi trường cài đặt. Ví dụ *dateOfBirth* có kiểu là *Date* (một lớp thông dụng trong các ngôn ngữ và các môi trường cài đặt hướng đối tượng). Kiểu của thuộc tính và thao tác cũng có thể lấy lớp của mô hình lớp đang được đặc tả. Ví dụ thao tác *getHomeAddress*() trên *CarSharer* có thể có kiểu là *Address* (xem hình 4.9).



Hình 4.9: Các đặc trưng có kiểu lớp

Liên quan đến việc sử dụng kiểu theo hình thức này là thuộc tính hoặc trị trả về sẽ cung cấp các thao tác có sẵn trên lớp. Vì vậy, một thuộc tính kiểu *Date* có thể cung cấp các thao tác định dạng và quốc tế hoá ngày tháng của lớp *Date*.

Trong mô hình lớp, hiếm khi dùng chung các *kiểu gốc* với các *kiểu lớp*. Các kiểu gốc, như int, có thể được dùng để biểu diễn một giá trị nguyên mà chỉ có các phép toán số học đơn giản được thực hiện trên giá trị này. Các ngôn ngữ, như *Java*, thường cung cấp một lớp biểu diễn cho các kiểu gốc này. Ví dụ, lớp *Integer* của *Java* hoạt động như một vỏ bọc xung quanh một giá trị int, để cung cấp các phép toán bổ sung như so sánh (ví dụ, *equals*) và chuyển kiểu (ví dụ *floatValue*). Như vậy các tên kiểu như String, int và Date phản ánh cách sử dụng cả hai loại kiểu.

4.4.3.2 Tính khả kiến của thuộc tính và thao tác

Thuộc tính và thao tác có thể được gán một mức độ *khả kiến* (visibility). Tính khả kiến của một đặc trưng có thể được định nghĩa bằng một từ khoá hoặc ký hiệu. Có ba mức độ khả kiến là: *private* (ký hiệu -), *public* (+) và *protected* (#).

Tính khả kiến của thuộc tính và thao tác liên quan đến tính hiệu lực của nó với các lớp khác. *Private* có nghĩa là một đặc trưng chỉ có thể tồn tại bên trong lớp sở hữu. *Public* có nghĩa là đặc trưng là hiệu lực với bất cứ lớp nào kết hợp với lớp sở hữu. *Protected* có nghĩa là đặc trưng đó chỉ có

trong lớp sở hữu và trong các lớp con cấp 1. *Protected* liên quan đến các khái niệm *generalization* (tổng quát hoá) được trình bày trong chương 5.

Ví dụ 4.1

Tính khả kiến của mỗi thuộc tính và thao tác trong hình 4.9 là gì?

Trả lời: *homeAddress* và *dateOfBirth* là *private*, *getHomeAddress* là *public*.

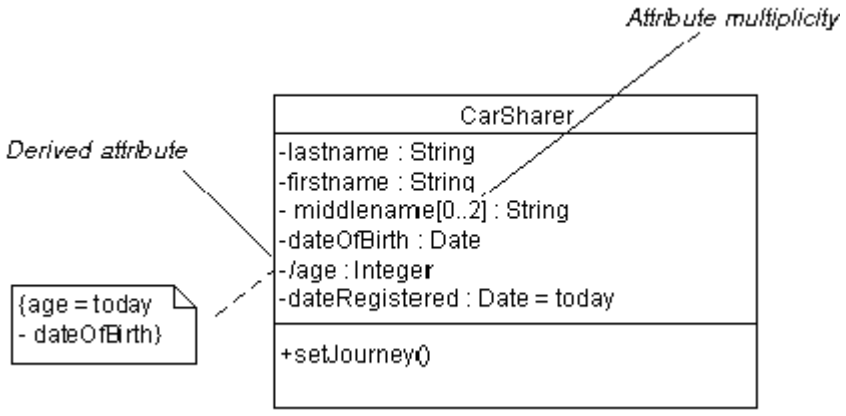
Thông thường, các thuộc tính là *private* và các thao tác hầu hết là *public*. Tuy nhiên các thao tác *private* cũng có thể được sử dụng cho các tác vụ bên trong lớp. Không có giá trị mặc định cho tính khả kiến. Nếu bỏ qua thì có nghĩa là *visibility* không được hiển thị hoặc chưa được định nghĩa.

Cơ chế mở rộng của UML cho phép ngôn ngữ cài đặt xác định tính *visibility*. Điều này cho phép đặc tả *visibility* theo ngôn ngữ định dùng. Ví dụ, nếu dự ta dự định cài đặt lớp bằng C++ thì có lẽ ta mong muốn dùng *friend visibility* với một biểu tượng thích hợp.

Tập các thao tác *public* của một lớp thường được xem như giao diện (interface) của lớp đó. Các ký hiệu UML cho giao diện lớp sẽ được trình bày trong chương 7.

4.4.3.3 Chi tiết về thuộc tính

Có ba tính chất khác của thuộc tính sẽ được thảo luận ở đây, đó là *giá trị khởi tạo* (initial value), *thuộc tính dẫn xuất* (derived attribute) và *bản số* (multiplicity). Ký hiệu cho cả ba tính chất này được trình bày trong hình 4.10.



Hình 4.10: Các tính chất initial value, derived attribute *và* multiplicity

Initial value

Giá trị khởi tạo của một thuộc tính có thể được đặc tả trong biểu đồ lớp bằng cách thêm vào mệnh đề =<*initialValue*> sau kiểu của thuộc tính. Trong hình 4.10, giá trị khởi tạo cho thuộc tính *dateRegistered* là ngày hiện hành của hệ thống. Các phương thức cài đặt các thao tác tạo đối tượng nên để ý đến đặc tả này và đảm bảo rằng *dateRegistered* được đặt là ngày hiện hành của hệ thống. Quá trình xử lý sau đó, trong một hàm khởi tạo, có thể ghi đè lên giá trị khởi tạo này, nhưng giá trị khởi tạo phải được gán ngay lúc ban đầu.

Derived attribute

Giá trị của một thuộc tính dẫn xuất được xác định từ các giá trị của các thuộc tính khác (hoặc từ các đặc trưng của lớp khác). Ta không cần cài đặt một thuộc tính cho thuộc tính dẫn xuất, vì giá trị của nó đã được xác định từ các giá trị của các đặc trưng khác. Sự có mặt của nó trong mô hình quan niệm hoặc trong mô hình đặc tả sẽ truyền đạt một yêu cầu và để cải thiện sự rõ ràng của mô hình. Trong thực tế, một thuộc tính dẫn xuất có thể được cài đặt như một thao tác tường minh hoặc một thuộc tính để giữ giá trị dẫn xuất. Trong UML, một thuộc tính dẫn xuất được chú thích bằng cách thêm một dấu gạch xuôi (/) ngay trước tên thuộc tính. Đặc tả cho thuộc tính dẫn xuất có thể được biểu diễn như một ghi chú đính kèm.

Trong hình 4.10, thuộc tính *age* được chỉ định là dẫn xuất. Sự dẫn xuất được biểu diễn như một ràng buộc dựa trên ngày hiện hành của hệ thống và giá trị của thuộc tính ngày sinh *dateOfBirth* (lấy ngày hiện hành của hệ thống trừ cho ngày sinh).

Fowler & Scott (1997) chỉ ra rằng tương tác của thuộc tính dẫn xuất tuỳ thuộc vào ngữ cảnh mà dẫn xuất được xem xét. Dẫn xuất đơn giản là biểu diễn một ràng buộc giữa thuộc tính dẫn xuất và các giá trị khác. Ví dụ, để tăng hiệu suất chúng ta muốn lưu trữ tuổi của *CarSharer* như là một thuộc tính trong cài đặt hệ thống. Sau đó lớp có trách nhiệm quản lý giá trị của *age* dựa vào ràng buộc đã mô tả.

Multiplicity

Một mệnh đề multipliciy có thể được dùng ngay sau tên thuộc tính để chỉ định số lượng các giá trị mà một thuộc tính có thể giữ. Multiplicity của một thuộc tính có dạng [m..n]. Trong đó m là số ít nhất và n là số nhiều nhất các giá trị mà thuộc tính có thể giữ. Cận dưới m là một số nguyên không âm, cận trên n là một số nguyên lớn hơn hoặc bằng m hoặc là ký

tự *, có nghĩa là '*many*' (một số không xác định, không giới hạn nhưng lớn hơn cận dưới). Có một số qui ước cho multiplicity:

- 1..1 được rút gọn là 1
- 0..* được rút gọn là *
- Nếu không xác định multiplicity thì mặc định là 1 (dạng rút gọn của 1..1)

Giá trị đơn được dùng để chỉ định số lượng cố định các giá trị của thuộc tính, ví dụ giá trị 7 trong mảng *dayName*[7]. Vài giá trị hợp lệ của multiplicity có thể là:

1. 'telephoneNumber[1..3]' - định nghĩa này nói rằng "*có ít nhất một giá trị được giữ cho số điện thoại, và có thể tăng lên đến ba*"
2. 'telephoneNumber[0..1]' - định nghĩa này nói rằng "*số điện thoại có thể là null hoặc chỉ có một số được giữ*". Giá trị null có nghĩa là không có giá trị nào cả. Null khác với 0 (số nguyên) hoặc khoảng trắng (đối với chuỗi).
3. "telephoneNumber[1..*]" – định nghĩa này nói rằng "*có ít nhất một giá trị cho số điện thoại, và có thể tăng lên đến vô hạn*".

Khi biết rõ thuộc tính sẽ có một dãy các giá trị, muliplicity cụ thể là chưa biết nhưng không thể là [1]. Trong trường hợp như thế này dùng [*] cho tổng quát.

Đặc tả UML (OMG 1999b, xem mục 4.43) cho phép một mệnh đề multiplicity được tạo từ một danh sách các mệnh đề con phân cách nhau bởi dấu phẩy. Ví dụ *attributeName*[1, 6..12] cho biết hoặc duy nhất một, hoặc từ 6 đến 12 giá trị có thể được lưu giữ. Minh hoạ cho các mệnh đề nhiều multiplicity như vậy có thể là hệ thống theo dõi các đội đăng ký thi đấu bóng chuyền. Trong đó, lớp *Team* có thuộc tính *playerName*[1, 6..12]: String. Multiplicity này cho biết hoặc là *một tên* sẽ được giữ (chẳng hạn như tên liên lạc của một đội khi đội đăng ký lần đầu), hoặc là giữ tên của các vận động viên của đội, từ 6 đến 12 vận động viên (giả sử luật cho phép một đội đăng ký tối thiểu 6 vận động viên và tối đa 12 vận động viên).

Lưu ý rằng thứ tự của các mệnh đề multiplicity khác nhau không ám chỉ đến trình tự thời gian. Trong ví dụ trên, không có điều nào cho thấy tên liên lạc phải được giữ trước khi giữ tên của cả đội.

4.4.3.4 Chi tiết về thao tác

Đặc tả cơ bản của các thao tác đã được phác thảo. Mỗi thao tác có thể bao gồm một danh sách các tham số được phân cách bởi dấu phẩy. Các

đặc tả mở rộng cho thao tác liên quan đến mức độ chi tiết của các tham số này.

Parameter

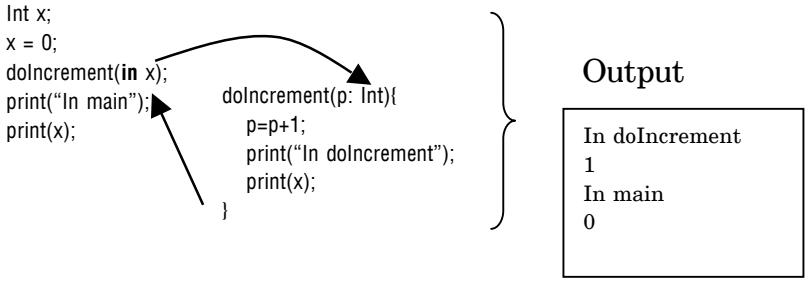
Tham số có dạng <*parameterName*> : <*type*>, ví dụ *setNextOfKin*(*name* : *String*), *setFirstMatched*(*firstDate* : *Date*) hoặc *equals*(*testAddress* : *Address*).

Ta có thể bỏ đi <*parameterName*> chỉ để lại <*type*> làm tham số, ví dụ như *setNextOfKin*(*String*), *setFirstMatched*(*Date*) hoặc *equals*(*Address*).

Cũng có thể gán giá trị mặc định cho tham số bằng cách thêm giá trị này vào sau tham số với mệnh đề =<*defaultValue*>. Ví dụ *approveApplication*, có một tham số kiểu *Date* với giá trị mặc định là ngày hiện hành của hệ thống, được đặc tả là *approveApplication*(*dateApproveed* : *Date* = *today*).

Cũng có thể đặc tả *loại tham số* (parameter kind) ở trước tham số. Parameter kind có thể là *vào* (in), *ra* (out) hoặc *vào ra* (inout), mặc định là *in*. Parameter kind là đặc tả được sử dụng trong các môi trường cài đặt, ở đó các tham số có thể được truyền theo kiểu tham chiếu thay vì tham trị.

Nếu tham số được truyền dạng tham trị, thì giá trị của tham số được giữ như một giá trị riêng (nghĩa là nó có riêng một địa chỉ trong bộ nhớ). Có nghĩa là bất kỳ thay đổi nào đối với giá trị của tham số đều không ảnh hưởng đến *biến gốc* trừ phi những thay đổi này được gởi trở lại (dùng giá trị trả về) và ghi tường minh vào biến gốc. Xem hình 4.11.

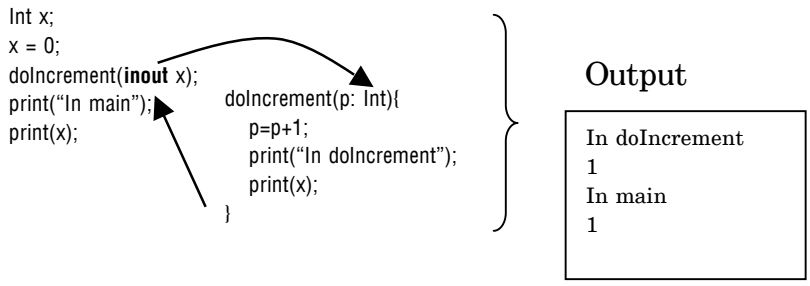


Hình 4.11: Tham số được truyền dạng tham trị

Trong ví dụ này, biến *nguyên* x được tạo ra được gán giá trị 0 rồi được truyền theo dạng tham trị (in) qua tham số p cho thao tác *doIncrement*. Thao tác này tăng *p* lên 1 rồi in ra thông báo "*In doIncrement*" cùng với giá trị của biến (là 1). Kết thúc, *doIncrement* không trả về bất kỳ giá trị nào (kiểu void). Cuối cùng hàm *main* in ra thông báo "*In main*" cùng với

giá trị của biến x trong hàm *main* (là 0). Giá trị này không bị ảnh hưởng bởi tiến trình bên trong thao tác *doIncrement*. Biến x và tham số p có địa chỉ khác nhau nên việc thay đổi trên p không ảnh hưởng đến x.

Ngược lại, khi tham số được truyền theo kiểu tham chiếu, thì địa chỉ bộ nhớ của biến được truyền cho thao tác. Vì vậy biến gốc và tham số cùng chia sẻ một địa chỉ bộ nhớ duy nhất. Bất kỳ thay đổi nào lên tham số cũng sẽ làm thay đổi giá trị của biến bên ngoài. Kết quả là biến gốc đã thay đổi bởi quá trình xử lý của thao tác. Hình 4.12 là ví dụ minh hoạ.



Hình 4.12: Tham số được truyền dạng tham chiếu

Trong ví dụ thứ hai này, biến *nguyên* x một lần nữa được tạo là 0 nhưng được truyền bằng tham chiếu (*inout*) qua tham số p cho thao tác *doIncrement*. Thao tác này sẽ cũng tăng giá trị của p lên 1 và in thông báo "*In doIncrement*" cùng giá trị của p (là 1). Sau đó thao tác kết thúc và trả quyền điều khiển về cho hàm *main*. Hàm *main* sẽ in thông báo "*In main*" và giá trị của biến x. Do biến x và tham số p có cùng một địa chỉ bộ nhớ nên việc thay đổi giá trị trên p đã làm ảnh hưởng trực tiếp đến giá trị của x.

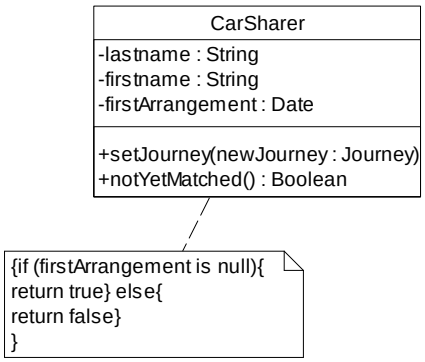
Như vậy, tham số *in* là tham số được truyền theo tham trị, còn tham số *out* và *inout* được truyền theo tham chiếu. Một tham số *out* không cung cấp thông tin cho thao tác nhưng để nhận giá trị trả về khi thao tác kết thúc.

Ghi chú cho phương thức

Đặc tả *thân phương thức* (method body) cho một thao tác có thể bao gồm một ghi chú được đính kèm theo thao tác trên biểu đồ lớp (xem hình 4.13). Cách biểu diễn này thường được dùng khi đặc tả có kích thước nhỏ, và đặc tả này sẽ góp phần tiện ích cho việc làm sưu liệu trong biểu đồ lớp.

Một lưu ý quan trọng là ghi chú không chỉ ra cách cài đặt phương thức. Nó cung cấp những gì mà phương thức sẽ làm. Đặc tả này sẽ thực hiện

một cách hình thức dưới dạng ràng buộc, sử dụng ngôn ngữ ràng buộc đối tượng (xem chương 12).

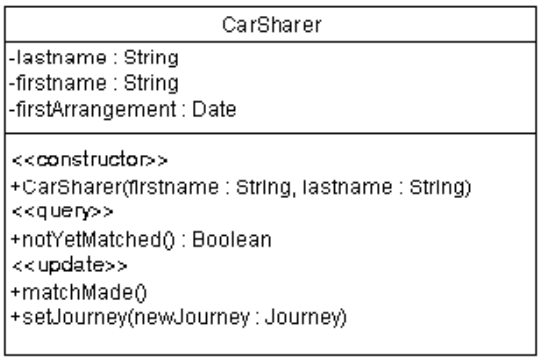


Hình 4.13: Ghi chú đặc tả phương thức

Nhóm theo stereotype

Khía cạnh cuối cùng của ký hiệu UML cho các thao tác, là khả năng nhóm các thao tác theo loại. Việc nhóm các thao tác chỉ được thực hiện khi sử dụng các công cụ CASE (chương 15) có hỗ trợ ký hiệu UML này.

Các stereotype được dùng cho việc gom nhóm này có tên như *<<constructor>>*, *<<query>>*, *<<update>>*. Các thao tác *khởi tạo* (constructor) sẽ tạo ra các thể hiện của lớp. Các thao tác *truy vấn* (query) không làm thay đổi các giá trị thuộc tính của thể hiện. Các thao tác *cập nhật* (update) có thể làm thay đổi các giá trị thuộc tính của thể hiện. Xem ví dụ của *nhóm thao tác* trong hình 4.14.



Hình 4.14: Các thao tác được gom nhóm bởi khuôn dạng

4.4.4 Kết hợp

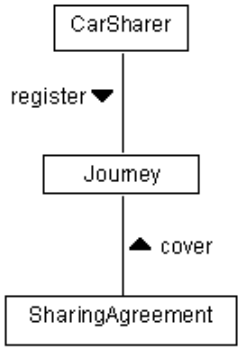
Một hệ thống hướng đối tượng được xây dựng từ các *kiểu lớp*, chúng cộng tác với nhau bằng cách gởi thông điệp và nhận phản hồi. Khi đang chạy, một hệ thống hướng đối tượng chứa các thể hiện tương ứng với các kiểu lớp của nó. Khi các thể hiện của một lớp truyền thông điệp cho các thể hiện của một lớp khác, một kết hợp được tạo ra giữa hai lớp này.

4.4.4.1 Tên của mối kết hợp

UML minh hoạ mối kết hợp giữa hai lớp bằng một đường thẳng liền nét. Kết hợp có thể được gán nhãn để cho biết bản chất của nó. Nếu một kết hợp được gán nhãn thì nên dùng kèm một mũi tên để giải thích thêm cho nhãn.

Hình 4.15 minh hoạ 2 kết hợp đơn giản. Một giữa *Journey* và *SharingAgreement*, một giữa *CarSharer* và *Journey*. Hai kết hợp này cho biết một *CarSharer đăng ký* (register) một *Journey* và một *SharingAgreement bao gồm* (cover) một *Journey*.

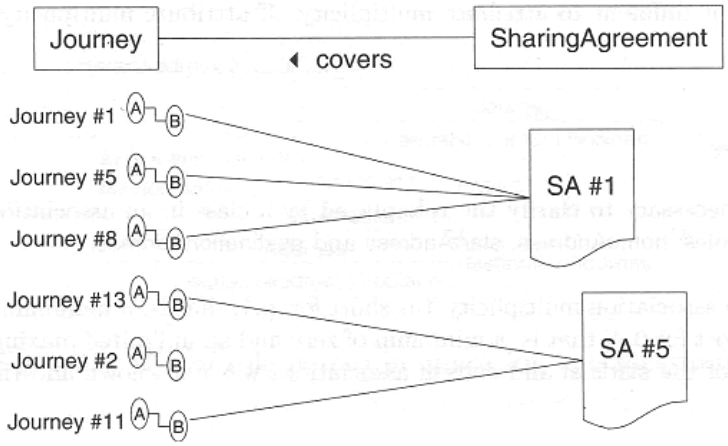
Một kết hợp giữa hai lớp còn được gọi là *kết hợp hai phía*. Tuy nhiên, đây là dạng thường gặp và, ta dùng *kết hợp* cho ngắn gọn.



Hình 4.15: Các kết hợp có tên

4.4.4.2 Bản số

Một thành phần khác có thể được thêm vào kết hợp là thông tin về *số lượng* (multiplicity) của kết hợp. Trong một kết hợp, multiplicity cho biết số lượng các thể hiện của lớp ở đầu bên kia của kết hợp so với một thể hiện của lớp ở đầu bên này của kết hợp. Khái niệm này được minh hoạ trong hình 4.16. (Lưu ý rằng ký hiệu được dùng để minh họa các thể hiện của các lớp trong hình 4.16 không phải là thành phần của ký hiệu UML).



Hình 4.16: Các xuất hiện của kết hợp

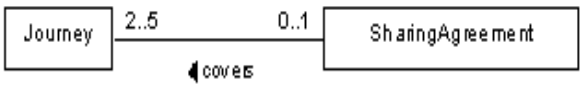
Phải có ít nhất hai *Journey* trong một *SharringAgrement*, nếu không chúng sẽ không được chia sẻ. *CarMatch* có tối đa 5 *Journey* trong một *SharringAgreement*, vì hầu hết các xe hơi tối đa chở được 5 người. Cho nên multiplicity của *Journey* cho mỗi *SharrringAgrement* là từ 2 đến 5.

Ở chiều ngược lại, một *Journey* có thể không có *SharingAgreement*. Trường hợp này có thể là do *Journey* này chưa được kết hợp trong một bản thoả thuận nào hoặc người tham gia trong bản thoả thuận đã kết thúc. Để tránh xung đột về quyền lợi, *CarMatch* giới hạn một *Journey* không thể cùng một lúc đăng ký nhiều hơn một *SharingAgreement*. Như vậy multiplicity là 0 hay 1 *SharingAgreement* cho 1 *Journey*.

Tóm lại, kết hợp giữa *Journey* và *SharingAgreement* có các multipicity sau: một *SharingAgreement* bao gồm từ 2 đến 5 *Journey* và một *Journey* được bao gồm từ 0 đến 1 *SharingAgreement*.

Thông tin này có thể được chỉ ra trên mối kết hợp của biểu đồ lớp. Như trên đã nói, multiplicity nằm ở cuối của kết hợp theo hướng phát biểu. Và như vậy, multiplicity 0..1 được đặt ở phía *SharingAgreement*.

Trong biểu đồ lớp, ký hiệu multiplicity trên mối kết hợp giống với ký hiệu multiplicity của thuộc tính (mục 4.4.3.3). Ký hiệu đầy đủ của multiplicity cho kết hợp *cover* (bao gồm) được minh họa trong hình 4.17.



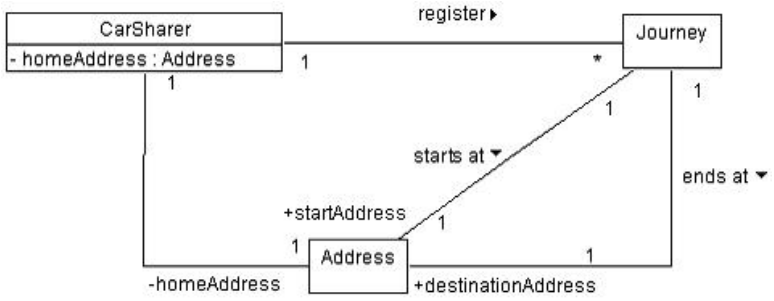
Hình 4.17: Multiplicity của kết hợp

Nếu multiplicity ở một đầu kết hợp không được chỉ ra thì có nghĩa là chưa biết hoặc không xác định, điều này khác với multiplicity của thuộc tính, nếu bỏ trống thì được xem là 1.

4.4.4.3 Tên vai trò

Trên các kết hợp, cần phải làm rõ vai trò mà lớp phải đóng. Hình 4.18 minh họa ba vai trò của lớp *Address* là *homeAddress*, *startAddress* và *destinationAddress*.

Nhớ rằng multiplicity 1 của kết hợp là dạng rút gọn của 1..1, và * là dạng rút gọn của 0..*. Multiplicity ở phía *Journey* của kết hợp *starts at* và kết hợp *ends at* là không biết nên được bỏ trống.



Hình 4.18: Tên vai trò trên kết hợp

Trong kết hợp vô danh giữa *CarSharer* và *Address*, thì *Address* đóng vai trò *homeAddress* của *CarSharer* và ta có thể thấy một *CarSharer* có một và chỉ một *homeAddress*. Cách cài đặt kết hợp này như một *thuộc tính private* kiểu *Address* trong *CarSharer* cũng được chỉ ra.

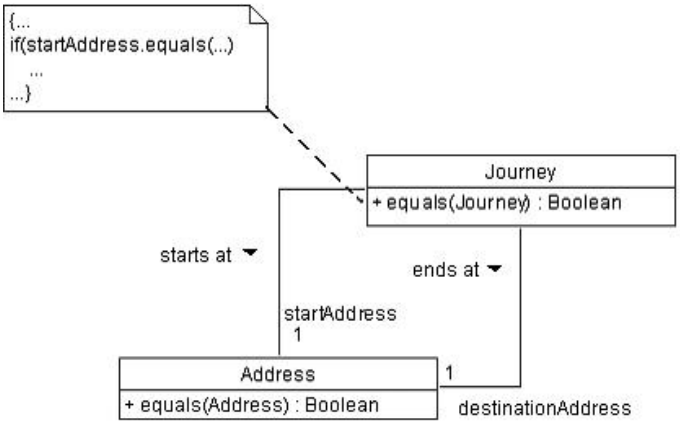
Kết hợp *one-to-one* giữa *CarSharer* và *Address* ngụ ý rằng ngay cả khi có hai người dùng chung xe có cùng một địa chỉ nhà, thì hai địa chỉ này vẫn sẽ được giữ như hai thể hiện phân biệt và độc lập của *Address*. Chúng có cùng giá trị, nhưng vẫn là các thể hiện khác nhau của Address.

Ví dụ 4.2

Thảo luận vai trò của *Address* trong hai kết hợp *starts at* và *ends at* giữa *Journey* và *Address* trong hình 4.18.

Lời giải: Từ multiplicity của hai kết hợp, chúng ta có thể thấy một *Journey* sẽ *bắt đầu tại* (starts at) một *Address*, và sẽ *kết thúc tại* (ends at) một *Address*. Các thể hiện của *Address* có thể đóng vai trò *startAddress* cũng như *destinationAddress* của một *Journey*. Điều này có thể được biểu diễn như là các thuộc tính cài đặt private – *startAddress*:*Address* và –*destinationAddress*:*Address* tương ứng.

Tên vai trò có thể có ích trong đặc tả phương thức, vì trong *thân* của phương thức có thể dùng nó tham chiếu đến thể hiện của lớp kết hợp (hình 4.19).



Hình 4.19: Cách sử dụng tên vai trò trong đặc tả thao tác

4.5 Hướng dẫn lập mô hình

Fowler và Scott (1997) đã thảo luận quan điểm về các *định hướng* trong qui trình phát triển mô hình lớp. Các định hướng bao gồm định hướng quan niệm, định hướng đặc tả và định hướng cài đặt.

Mô hình và biểu đồ theo hướng quan niệm, phản ánh các yêu cầu của hệ thống, sẽ chứa cả hai loại lớp gồm các lớp được nhìn thấy hiển nhiên và các lớp phải nhìn thấu vào bên trong hệ thống mới thấy được. Một mô hình và biểu đồ như vậy có thể được xây dựng dựa trên các kết quả của hoạt động thu thập thông tin lúc bắt đầu dự án. Các kết quả này có thể bao gồm các đoạn trích phỏng vấn, các tài liệu mẫu, các qui định và sách hướng dẫn. Mô hình và biểu đồ được cải tiến thông qua quá trình lặp đánh giá lại và thu thập thêm thông tin.

Mô hình và biểu đồ theo hướng đặc tả hoặc cài đặt sẽ phản ánh ý tưởng thiết kế hay cài đặt hệ thống phần mềm. Các mô hình này có thể được xây dựng tốt trên các lớp trong mô hình quan niệm. Tuy nhiên, cũng phù hợp khi tiếp cận theo hướng dựa trên nguyên lý chuyển giao chức năng (xem mục 4.5.2).

Mục 4.4.3 giới thiệu các ký hiệu cho kiểu trả về của một thao tác. Ở đây, cần lưu ý rằng định hướng mà biểu đồ lớp được xây dựng sẽ ảnh hưởng đến cách hiểu việc không khai báo kiểu dữ liệu trả về. Trong biểu đồ quan niệm, việc bỏ qua giá trị trả về có thể được hiểu là kiểu trả về

không quan trọng hoặc không xác định hay chưa được hoàn thành. Trong biểu đồ đặc tả và cài đặt thì không khai báo kiểu trả về nghĩa là thao tác không có trị trả về (tương đương với kiểu *void*).

4.5.1 Lập mô hình quan niệm

Việc tạo ra biểu đồ lớp theo hướng quan niệm liên quan đến các hoạt động sau:

- Tìm các lớp và các mối kết hợp
- Xác định các thuộc tính các thao tác và quyết định lớp chứa chúng.
- Xác định cấu trúc tổng quát hoá (xem chương 5)

Chúng ta không cần phải thực hiện lần lượt các bước trên. Thay vào đó, các vòng lặp sớm xoay quanh việc xác định các lớp, thuộc tính và mối kết hợp. Trong các vòng lặp sau, các thao tác và cấu trúc tổng quát hoá sẽ bắt đầu hình thành.

4.5.1.1 Tìm các lớp và các mối kết hợp

Các lớp và các mối kết hợp có thể được xác định từ các mô tả use case (chương 3), trực tiếp từ kết quả của quá trình thu thập thông tin, hoặc bằng cách đầu tư công sức vào biểu đồ lớp (luôn nghĩ về nó và mang các kinh nghiệm ra áp dụng).

Các danh từ và cụm danh từ trong các use case và các đoạn trích thường chỉ ra các lớp. Bennet và các cộng sự (1999) gợi ý một vài loại lớp có thể được xác định:

Sự xuất hiện cụ thể của một kiểu chung chung, chẳng hạn như người ('*John Doe*'), tổ chức ('*Blad Eagle Insurance*') và đơn vị của tổ chức ('*The slales team*').

Các cấu trúc vốn có trong hệ thống, như '*car sharer*', '*volunteer team*'.

Sự trừu tượng như:

Con người và vai trò như '*Car sharer*', '*Volunteer*', '*Account holder*', '*Insurance sales advisor*'.

Vật như '*car*', '*cover note*', '*insurrance policy*'.

Khái niệm như '*sale*', '*skill*', '*requirement*'

Các quan hệ lâu dài giữa các lớp, như '*agreement*', '*registration*'.

Các cụm động từ có thể xác định các mối kết hợp giữa các lớp. Nói cách khác, các kết hợp logic vốn có giữa các lớp có thể trở nên hiển nhiên khi

tinh chế biểu đồ. Ví dụ, *customer holds account* (khách hàng giữ tài khoản), *car sharer registers journey* (thành viên đăng ký lộ trình) hoặc *volunteer holds particular skills* (tình nguyện viên giữ các kỹ năng riêng).

Ví dụ 4.3

Dưới đây là một đoạn trích khác từ cuộc phỏng vấn giữa *Mick Perez* và *Janet Hoffner* (*Janet Hoffner* là một trong các vị giám đốc của *CarMatch*).

Yêu cầu: Hãy xác định các lớp và các kết hợp trong đoạn trích này.

> **Mick Perez:** Chúng ta có thể thảo luận về cách thức dùng chung xe được tổ chức như thế nào ngay bây giờ được không? Tôi muốn tìm hiểu thêm các ý tưởng của ông trong vấn đề này.
>
> **Janet Hoffner:** Được. Tôi cho rằng vấn đề ở đây là các thành viên, là người đã đăng ký với chúng tôi để họ có thể chia sẻ lộ trình với những người khác.
>
> **MP:** Xin ông vui lòng cho biết cách lưu giữ các chi tiết của lộ trình mà một người nào đó muốn chia sẻ.
>
> **JH:** Thực ra một thành viên có thể đăng ký một vài lộ trình với chúng tôi. Họ không bị giới hạn ở mức một lộ trình.
>
> **MP:** Tôi nghĩ rằng họ muốn chia sẻ ít nhất một lộ trình....?
>
> **JH:** Đúng vậy. Chúng ta phải nhớ rằng họ có thể đăng ký bao nhiêu lộ trình đều được. Khi chúng ta tìm được những người khác muốn chia sẻ lộ trình phù hợp với người này, thì chúng ta kết hợp họ và các thứ khác trong một bản thoả thuận.

Lời giải:

Mick Perez đã xem qua đoạn phỏng vấn, dùng kinh nghiệm của anh ấy để lấy ra các lớp và các kết hợp có thể có cho biểu đồ quan niệm. Mick đã gạch dưới các danh từ và cụm danh từ có thể làm lớp, và vẽ một hộp đóng các động từ và các cụm động từ có thể làm mối kết hợp.

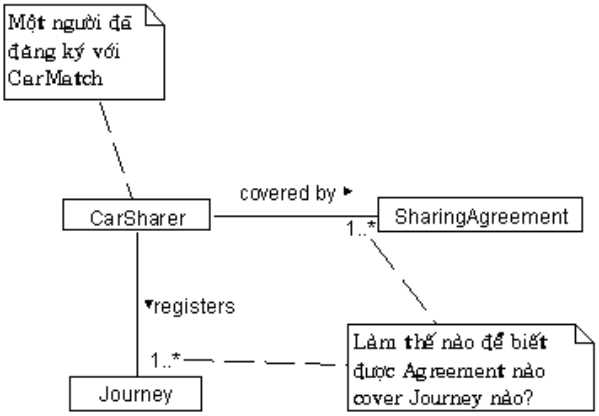
> **Mick Perez:** Chúng ta có thể thảo luận về cách thức dùng chung xe được tổ chức như thế nào ngay bây giờ được không? Tôi muốn tìm hiểu thêm các ý tưởng của ông trong vấn đề này.
>
> **Janet Hoffner:** Được. Tôi cho rằng vấn đề ở đây là các thành viên, là người đã đăng ký với chúng tôi để họ có thể chia sẻ lộ trình với những người khác.

MP: Xin ông vui lòng cho biết cách lưu giữ các chi tiết của lộ trình mà một người nào đó muốn chia sẻ.

JH: Thực ra một thành viên có thể đăng ký một vài lộ trình với chúng tôi. Họ không phải bị giới hạn ở mức một lộ trình.

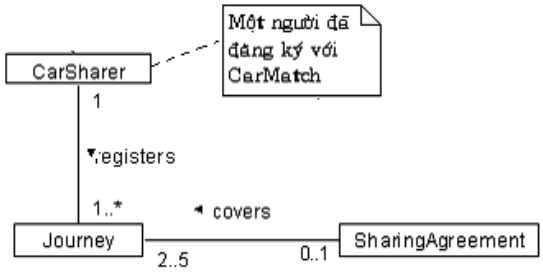
MP: Tôi nghĩ rằng họ muốn chia sẻ ít nhất một lộ trình...?

JH: Đúng vậy. Chúng ta phải nhớ rằng họ có thể đăng ký bao nhiêu lộ trình đều được. Khi chúng ta tìm được những người khác muốn chia sẻ lộ trình phù hợp với người này, thì chúng ta kết hợp họ và các thứ khác trong một bản thoả thuận.



Hình 4.20: Biểu đồ quan niệm được phác thảo lần đầu

Phác thảo đầu tiên của Mick là biểu đồ lớp như trong hình 4.20. Các lớp được xác định trong đoạn trích là *CarSharer*, *Journey*, *SharingAgreement* đều có trong biểu đồ. Hơn nữa các kết hợp ban đầu khích lệ Mick rằng các lớp này là hợp lý.



Hình 4.21: Biểu đồ quan niệm phác thảo lần hai

Nhưng Mick không chắc lắm về các kết hợp như ghi chú trong hình 4.20. Các câu hỏi cần phải được giải đáp, hoặc bằng cách khảo sát kỹ hơn các

khái niệm hoặc bằng cách mang biểu đồ và các ghi chú trở lại buổi phỏng vấn và làm rõ ràng các quan hệ. Trong trường hợp này, khái niệm *SharingAgreement* thực tế liên quan đến các *Journey* mà không liên quan gì đến *CarSharer* (xem hình 4.21).

4.5.1.2 Xác định thuộc tính và thao tác

Các mục dữ liệu trong mỗi thể hiện là các thuộc tính của lớp. Trong mô hình quan niệm, các thuộc tính nên được nhận biết ngay từ các nguồn thông tin có sẵn.

Các thao tác mô tả khả năng xử lý của một lớp. Trong mô hình quan niệm, không dễ xác định các thao tác đầy đủ cho mỗi lớp cho đến khi chi tiết trên các biểu đồ tương tác được hoàn tất (xem chương 8 và 9). Tuy nhiên một số thao tác có thể được thấy rõ từ các nguồn thông tin có sẵn và nên ghi chép lại.

Ví dụ 4.4

Khảo sát một đoạn phỏng vấn giữa Mick Perez và Janet Hoffner. Dựa vào đoạn trích này, hãy xác định các thuộc tính và thao tác bổ sung vào mô hình lớp.

> **Mick Perez:** Những thông tin nào về lộ trình ông sẽ giữ khi thành viên đăng ký?
>
> **Janet Hoffner:** Tôi nghĩ chắc anh cũng nhận ra, đó là "*từ đâu*" và "*đến đâu*". Chúng tôi cần biết nơi bắt đầu mỗi lộ trình và nơi lộ trình sẽ kết thúc. Chúng tôi cũng muốn biết thời gian cho mỗi hướng của lộ trình. Ý của tôi là các thời điểm đi và đến của mỗi lộ trình cho cả hai hướng.
>
> **MP:** Ông cần biết những loại thông tin gì về điểm xuất phát và đích đến của mỗi lộ trình? Thông tin đó sẽ được sử dụng để kết hợp các lộ trình có thể chia sẻ phải không?
>
> **JH:** Vâng**,** chắc chắn là chúng tôi sẽ giữ địa chỉ xuất phát và địa chỉ đích, chúng tôi sẽ dùng thông tin này để thực hiện việc kết hợp tự động. Địa chỉ nhà của họ cũng có thể được dùng trong một số trường hợp.

Lời giải:

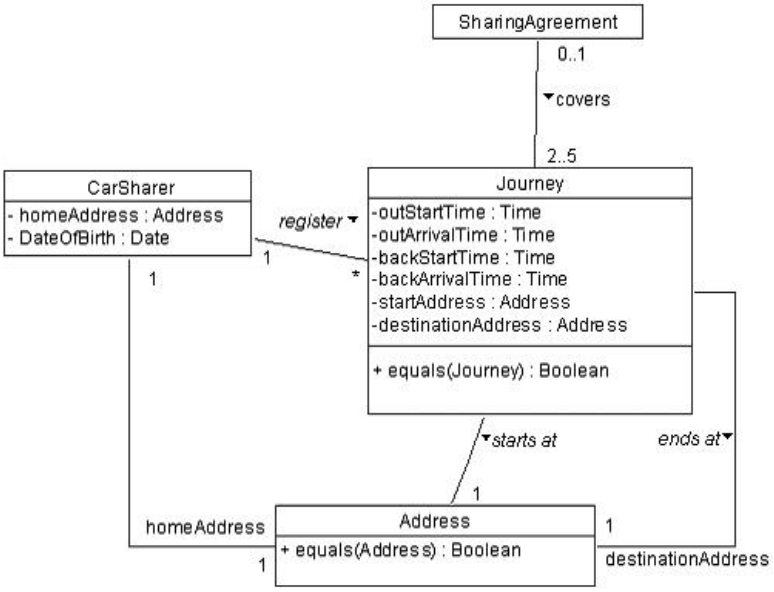
Dựa vào đoạn trích trên, có thể thấy một số thuộc tính về thời gian được yêu cầu cho các thời điểm đi và đến, cũng có 3 thông tin về địa chỉ được đề cập. Các địa chỉ này có thể được lưu giữ như các chuỗi văn bản nhưng sẽ không làm gì hơn là *đối sánh thô* cho mục đích chia sẻ.

Có một cách mô hình các địa chỉ trong vòng lặp sớm này là tạo ra lớp *Address*. Lớp này có nhiệm vụ quản lý thông tin liên quan đến địa chỉ và kết hợp hai địa chỉ thích hợp lại với nhau.

Để hỗ trợ, ta định nghĩa thao tác *equals*. Thao tác này trả về một giá trị boolean đúng hoặc sai phụ thuộc vào kết quả so sánh một địa chỉ là đủ gần với một địa chỉ khác. Thế nào là *đủ gần* cần được khảo sát thêm trong các vòng lặp sau.

Nếu lớp *Address* chứng tỏ sự chia trách nhiệm là không cần thiết, vì việc xử lý và đối sánh địa chỉ sẽ được mô hình trong lớp *CarSharer*, thì lớp *Adrress* có thể bị loại bỏ trong các vòng lặp sau. Nếu sự có mặt của lớp *Adrress* trở nên chắc chắn hơn thì đặc tả của nó phải được tinh chế và cải tiến trong các vòng lặp sau. Những gì làm cho sự tồn tại của một lớp trở nên chắc chắn chính là việc xác định và định vị các thuộc tính và các thao tác trong lớp này hoặc sự tham gia của nó trong các kết hợp ở tương lai.

Thao tác *equals* được gán cho *Journey* để xét tính đủ gần của hai lộ trình như yêu cầu. Một lần nữa tiêu chuẩn và đặc tả *đủ gần* cần được thiết lập.



Hình 4.22: Xác định các thuộc tính và thao tác

Các thuộc tính được xếp vào một lớp hầu như tương ứng với mục dữ liệu được biểu diễn bởi thuộc tính. Nói chung, các thao tác sẽ ở trong cùng một lớp với thuộc tính; cùng với các thuộc tính, chúng hoạt động để cung

cấp nhiệm vụ chức năng của chúng. Các thuộc tính và các thao tác có thể được chuyển sang lớp khác khi tinh chế biểu đồ lớp.

Các thuộc tính và thao tác được xác định từ đoạn trích và được thảo luận ở trên được minh hoạ trong hình 4.22.

Trong vòng lặp kế tiếp, lớp *Address* được gán các thuộc tính thích hợp để biểu diễn thông tin địa chỉ (ví dụ đường, phố, bang hoặc quốc gia, mã vùng hoặc mã bưu điện). Trong hình 4.22 cài đặt của các kết hợp *starts at* và *end at* được mô tả như là các thuộc tính kiểu *Address* trong lớp *Journey*. Mối kết hợp vô danh giữa *CarSharer* và *Address* cũng được cài đặt là thuộc tính *homeAddress:Address* trong lớp *CarSharer*. Các kết hợp có thể không được cài đặt ở giai đoạn đầu này.

Việc mô hình theo cách này được tiếp tục cho đến khi phân tích viên thoả mãn. Mô hình nên kết hợp chặt chẽ các lớp chính tồn tại trong lĩnh vực vấn đề cần giải quyết, làm các thuộc tính và một số thao tác của chúng sẽ trở nên rõ ràng trong cách giải quyết vấn đề.

4.5.2 Mô hình hoá bằng chuyển giao chức năng

Khi phát sinh mô hình và biểu đồ theo hướng đặc tả hoặc cài đặt, ta có thể gặt hái nhiều thành công hơn nếu dùng các nguyên lý chuyển giao chức năng. Các biểu đồ đặc tả và cài đặt sẽ phản ánh bản thiết kế hoặc cài đặt được dự tính của một hệ thống phần mềm tương ứng.

Nguyên lý chuyển giao chức năng đó là *trách nhiệm về quyền sở hữu dữ liệu và xử lý dữ liệu nên được ủy thác cho lớp thích hợp nhất*. Chúng ta đã thấy một minh họa đơn giản trong mục trên. Lớp *Journey* có trách nhiệm xác minh sự giống nhau giữa hai lộ trình (thao tác *equals(Journey):boolean*). Thao tác *equals* có thể sẽ đòi hỏi xác định các địa chỉ đầu và cuối giữa hai lộ trình là giống nhau. Nhiệm vụ kiểm tra hai địa chỉ là giống nhau về mặt địa lí không là việc của *Journey*. Nhiệm vụ này được chuyển giao cho thao tác *equals(Address):boolean* trong lớp *Address*.

Mô hình hoá bằng cách chuyển giao chức năng dựa trên việc khảo sát các tương tác giữa các lớp để thực hiện một số chức năng được yêu cầu trong hệ thống. Điều này muốn nói đến ba thành phần chính của lớp có thể được khảo sát là

- Lớp
- Thao tác
- Kết hợp

Rõ ràng các tương tác không thể xảy ra nếu không có lớp. Bất kỳ một tương tác nào đều được bắt đầu bằng một số thăm dò hay thông điệp từ

một nguồn phát ở bên ngoài. Nó có thể là một biến cố trong giao tiếp, một thông điệp từ một lớp trong hệ thống khác, một hành động ngắt (ví dụ, từ một sensor),.... Bằng cách xác định cách đáp ứng của một lớp, thao tác bắt đầu hiện ra. Thao tác xử lý thông điệp không nhất thiết được xử lý bởi phương thức của thao tác. Thay vì thế, các phương thức khác trong lớp có thể được gọi để giúp đỡ xử lý thông điệp này.

Khi một thao tác trong lớp cần thu nhận thông tin hoặc phát sinh quá trình mà không thuộc trách nhiệm của lớp. Thao tác sẽ gởi thông điệp đến lớp có trách nhiệm với thông tin này hoặc đến thao tác được yêu cầu chuyển giao trách nhiệm này. Việc truyền thông điệp từ lớp này đến lớp khác chỉ ra sự có mặt của một mối kết hợp giữa hai lớp. Như vậy xuất hiện một bức tranh các lớp cộng tác với nhau qua các thao tác để đạt đến mục tiêu chức năng lớn hơn.

Người ta cho rằng kỹ thuật thông dụng nhất để mô phỏng sự cộng tác giữa các lớp là kỹ thuật dùng thẻ *Lớp – Nhiệm vụ – Cộng tác CRC* (Class – Responsibility – Collaborate) được đưa ra bởi Beck & Cunningham – 1989; Wirfs - Brock, Wilkerson & Wiener – 1990. Thẻ CRC là một thẻ có kích thước bé (15cm x 8cm). Tên lớp được viết trên đỉnh của thẻ. Dọc theo bên trái thẻ là danh sách các nhiệm vụ. Dọc theo bên phải thẻ là danh sách các lớp cộng tác. (Hình 4.23)

Journey	
Responsibilities	**Collaborations**
Kiểm tra xem một Journey khác có giống với Journey này không. Bảo trì các chi tiết của một lộ trình.	Address hỗ trợ kiểm tra tính ngang bằng giữa hai địa chỉ.

Hình 4.23: Thẻ CRD của lớp Journey

Các phân tích viên tham gia phát triển hệ thống lấy một hoặc nhiều thẻ CRC. Sự cộng tác, cần thiết để thực hiện một yêu cầu, được diễn bởi nhóm phân tích với mỗi phân tích viên đóng vai trò của lớp họ đang giữ. Khi vở diễn tiếp tục, nhu cầu về một nhiệm vụ hay cộng tác được xác định. Một thu xếp được tổ chức để thoả thuận các lớp hiện tại hay lớp mới là phù hợp nhất với nhiệm vụ được xác định.

Mô hình với tiếp cận chuyển giao chức năng bắt đầu che mờ điểm bắt đầu theo UML. Tương tác giữa các lớp có được nhờ kỹ thuật diễn kịch

như CRC nên được mô hình hoá theo ký hiệu của các biểu đồ tương tác (cộng tác và tuần tự – xem chương 8 và 9)

Một biểu đồ lớp được đưa ra đầu tiên, dùng tiếp cận mô hình quan niệm, giữ vai trò cơ sở cho các kỹ thuật dựa trên tương tác. Như một sự lựa chọn, dùng tiếp cận mô hình dựa trên chuyển giao chức năng, biểu đồ lớp được phác thảo như là một trừu tượng của cấu trúc tĩnh của các lớp, được xác định qua mô hình của các tương tác cần thiết hỗ trợ một chức năng nào đó. Trong một số trường hợp (Bellin & Simone, 1997), việc phát thảo một biểu đồ quan niệm khởi tạo một vở diễn là một chiến lược phát sinh.

4.5.3 Tóm tắt các tiếp cận lập mô hình

Tiếp cận được chọn phụ thuộc vào phân tích viên có ý định đưa ra mô hình quan niệm hoặc đặc tả hoặc cài đặt. Với mô hình đặc tả và cài đặt, tiếp cận dựa trên tương tác động giữa các lớp và chuyển giao chức năng là phù hợp. Với mô hình quan niệm rõ ràng tiếp cận là mô hình khái niệm. Trong trường hợp đầu, một biểu đồ lớp có thể được đưa ra như là một trừu tượng của các lớp được xác định trong suốt quá trình lập mô hình hành vi động được yêu cầu trong hệ thống.

Trong thực hành, hai tiếp cận này có thể được dùng luân phiên cái này cạnh cái kia. Mô hình khái niệm xác định các lớp chính, đưa ra một tập các thẻ lớp (CRC) để bắt đầu lập mô hình chuyển giao chức năng.

Trong thực hành, các phân tích viên có kinh nghiệm sẽ dùng bất cứ tiếp cận nào thích hợp nhất. Tuy nhiên, quan trọng là khi kết thúc giai đoạn lập mô hình biểu đồ lớp là một biểu diễn rõ ràng và phù hợp với các khái niệm được mô hình. Nên có sự phân biệt rõ ràng giữa một biểu đồ lớp biểu diễn mô hình định hướng quan niệm với một biểu đồ lớp biểu diễn mô hình định hướng đặc tả hoặc cài đặt.

4.5.4 Các khái niệm hướng đối tượng

Đây là điểm tiện lợi để xác định và thảo luận các khái niệm định hướng đối tượng cơ sở được minh họa bởi các ký hiệu đã dùng trong các mục trước

4.5.4.1 Thao tác

Tên, danh sách tham số và kiểu trả về của một thao tác là *chữ ký* (signature) của thao tác. Hoàn toàn có thể có vài thao tác cùng tên và kiểu trả về trong một lớp với danh sách tham số khác nhau. Các danh sách tham số khác nhau phản ánh ngữ cảnh khác nhau khi gọi thao tác.

Khả năng xác định thao tác không đặc tả phương thức là đặc tính có ích trong mô hình hướng đối tượng. Biết tên của thao tác để gởi thông điệp,

kiểu để chờ giá trị trả về và các đối số phải truyền, cung cấp một đặc tả rõ ràng cho sự cộng tác giữa các thao tác.

4.5.4.2 Đóng gói

Ký hiệu của thao tác và tính khả kiến dẫn ta đến một khái niệm hướng đối tượng chính yếu khác, đó là *sự đóng gói* (encapsulation). Bằng cách đặc tả các thuộc tính là private và các thao tác là public; bằng cách yêu cầu dùng chính xác các *chữ ký của thao tác*, một lớp sẽ che giấu phương thức của thao tác khỏi thao tác gọi.

Điều này có ý nghĩa trong việc giới hạn phạm vi ảnh hưởng đến sự thay đổi trong cài đặt thao tác. Chừng nào thao tác vẫn còn cung cấp khả năng chức năng được đặc tả và chữ ký vẫn còn thích hợp thì cài đặt của một thao tác có thể bị thay đổi mà không phải thay đổi bất kỳ thao tác nào sử dụng nó. Ví dụ, giải thuật thực hiện một tính toán hoặc kiểu dữ liệu được sử dụng cho các mục dữ liệu phục vụ việc tính toán này có thể bị thay đổi mà không ảnh hưởng đến chữ ký của thao tác.

4.5.4.3 Trạng thái (state) của đối tượng

Đối tượng là thể hiện của lớp. Khi lớp có các thuộc tính và các thành phần kết hợp, mỗi đối tượng sẽ giữ các giá trị cụ thể cho mỗi thuộc tính và là thành viên trong kết hợp cụ thể với các đối tượng khác.

Nói chung tập các giá trị thuộc tính và các thể hiện của các kết hợp với các đối tương khác phản ánh trạng thái của một đối tượng. Nếu giá trị của một thuộc tính bị thay đổi hoặc một kết hợp được thực hiện hay bị cắt đứt thì trạng thái của lớp bị thay đổi. Trong qui trình phân tích, các thuộc tính mà các giá trị của nó có ý nghĩa sẽ được xác định. Mô hình kết quả thay đổi trạng thái có thể được thực hiện trong UML bằng cách dùng kí hiệu của biểu đồ chuyển trạng thái (xem chương 11)

4.6 Quan hệ với các biểu đồ khác

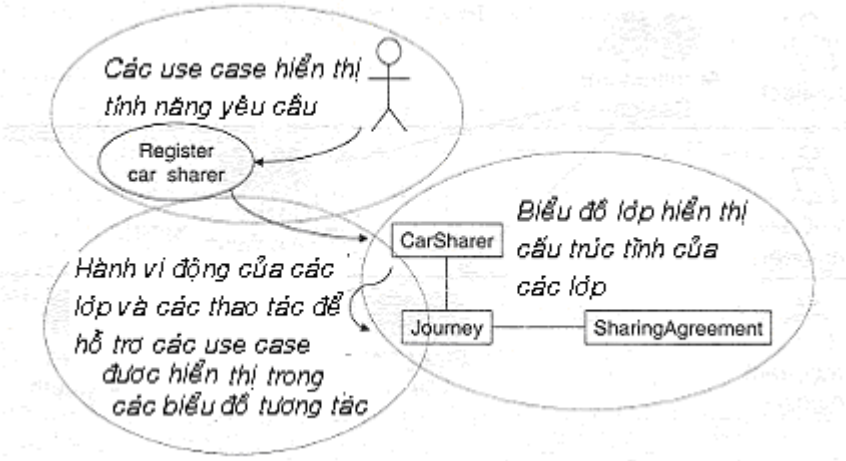
Biểu đồ lớp cho ta thấy cấu trúc tĩnh của các lớp trong hệ thống. Biểu đồ lớp được dùng để mô tả mô hình lớp ở ba định hướng chính: quan niệm, đặc tả và cài đặt.

Trong hệ thống hướng đối tượng, các thao tác được định nghĩa trong biểu đồ lớp sẽ tương tác với nhau để thực hiện các yêu cầu chức năng của hệ thống. Vì thế biểu đồ lớp sẽ gắn kết với các yêu cầu về chức năng nhờ vào động thái của chúng trong các biểu đồ tương tác (hình 4.24).

Động thái của biểu đồ lớp, gồm lớp và thao tác, được sưu liệu trong biểu đồ cộng tác hoặc biểu đồ tuần tự (là các biểu đồ tương tác, xem chương 8

và 9), còn các yêu cầu chức năng của hệ thống được mô tả bởi biểu đồ use case (chương 3).

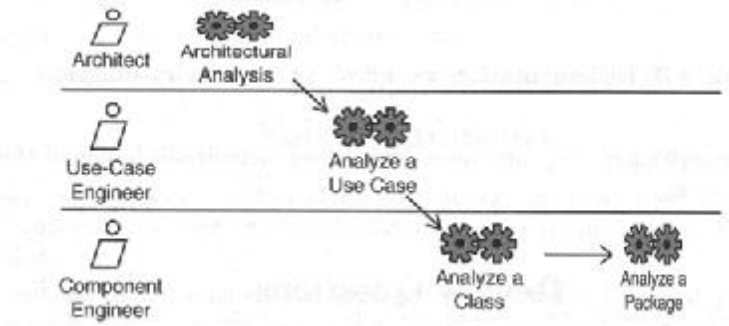
Tương tự các mô hình UML khác, mô hình lớp có thể được tổ chức thành các *gói* như là bộ phận của khung nhìn *Model Management* (chương 14).



Hình 4.24: Biểu đồ lớp quan hệ với các biểu đồ khác

4.7 Biểu đồ lớp trong UP

Chương 3 đã thảo luận về UP là qui trình *hướng đến* use case. Với tiếp cận cơ sở này, việc sử dụng biểu đồ lớp trong UP chịu ảnh hưởng mạnh bởi các khái niệm chuyển giao chức năng.

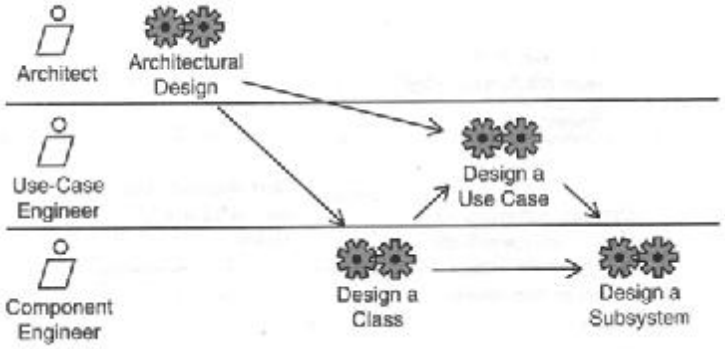


Hình 4.25: Analysis Workflow

Trong mô hình phân tích, một biểu đồ lớp theo hướng quan niệm được tạo ra từ phân tích use case (hình 4.25). Mô hình lớp được tạo ra trong

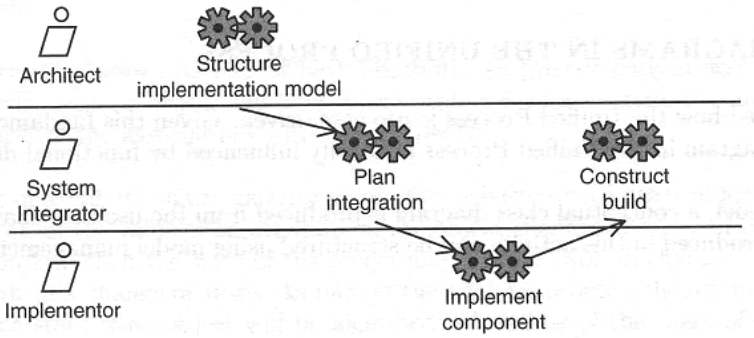
hoạt động này, có thể được cấu trúc hoá bằng cách dùng các khái niệm quản lý mô hình, chẳng hạn như các *gói*.

Sự cộng tác giữa các lớp, nhằm cung cấp khả năng cần thiết để hỗ trợ các yêu cầu (dưới dạng use case), sẽ được khảo sát chi tiết trong qui trình thiết kế. Việc xem xét các cộng tác này sẽ làm sáng sủa các đặc tả của lớp (hình 4.26). Khi đặc tả trở nên vững chắc, các lớp có thể được tổ chức thành các hệ thống con theo chức năng.



Hình 4.26: Design Workflow

Khi chuyển đổi từ đặc tả sang cài đặt, các hệ thống con sẽ hoạt động như một bộ khung để xác định các thành phần chức năng riêng lẻ. Các thành phần này có thể được xây dựng cùng nhau để cung cấp một mô hình (sau mỗi bước lặp) có thể xác định và kiểm tra khả năng thuộc chức năng của hệ thống cài đặt (xem hình 4.27).



Hình 4.27: Implementation Workflow

Khi các thành phần đã được cài đặt, chúng được kết hợp chặt chẽ trong các mô hình sau mỗi bước lặp để kiểm tra hệ thống.

Câu hỏi ôn tập

4.1 Trình bày các ký hiệu cơ bản của biểu đồ lớp?

4.2 Mục đích của biểu đồ lớp là gì?

4.3 Lớp là gì?

4.4 Ký hiệu cơ bản nhất của lớp?

4.5 Tên đường dẫn của lớp truyền đạt điều gì?

4.6 Trình bày ký hiệu biểu diễn tên đường dẫn của lớp?

4.7 Thuộc tính là gì?

4.8 Trình bày ký hiệu cơ bản cho một thuộc tính?

4.9 Các tính chất khác có thể có trong định nghĩa thuộc tính là gì?

4.10 Trình bày ký hiệu cho các tính chất nói trên?

4.11 Thao tác là gì?

4.12 Tham số là gì?

4.13 Thế nào là "*truyền tham số bằng tham chiếu*" và "*truyền tham số bằng tham trị*"? Cho biết sự khác biệt giữa chúng?

4.14 Ký hiệu cơ bản của thao tác là gì?

4.15 Ký hiệu tham số của thao tác là gì?

4.16 Trình bày và minh hoạ sự ảnh hưởng của các giai đoạn của dự án lên mức độ chi tiết trên các lớp.

4.17 Giá trị khởi tạo của tham số được đặc tả như thế nào?

4.18 Giá trị mặc định của thuộc tính được đặc tả như thế nào?

4.19 Hãy định nghĩa ý nghĩa của một kết hợp giữa hai lớp.

4.20 Ký hiệu cơ bản cho một kết hợp có nhãn là gì?

4.21 Ký hiệu để xác định vai trò của một lớp trong một kết hợp là gì?

4.22 Multiplicity là gì?

4.23 Trình bày ký hiệu của multiplicity?

4.24 Hai phần tử nào của một biểu đồ lớp có thể có multiplicity?

4.25 Hãy định nghĩa các định hướng của biểu đồ lớp.

4.26 Hai cách chính để vẽ biểu đồ lớp là gì?

Bài tập có lời giải

4.1 Trong hệ thống con *Insurrance* của *CarMatch*, thành viên có thể nhận được một hợp đồng bảo hiểm. Mỗi hợp đồng bảo hiểm là một

phần của một *lược đồ* (scheme) bảo hiểm của một công ty bảo hiểm nào đó. Vậy các lớp nào có thể được liệt kê ở đây?

Lời giải:

Theo trên, thông tin là ở mức quan niệm và cách tiếp cận quan niệm dường như là hợp lý nhất. Không cần lớp cho *CarMatch*, nó là ngữ cảnh hơn là một lớp. Các khái niệm được đề cập trong ví dụ này có thể được mô hình hoá thành các lớp như: *insurranc policy*, *insurrance company*, *insurrance scheme* và *car sharer*.

Lược đồ bảo hiểm là một khái niệm phức tạp. Cụm từ "*...hợp đồng bảo hiểm là một phần của một lược đồ bảo hiểm của một công ty bảo hiểm nào đó*" ngụ ý rằng một hợp đồng là một phần của một lược đồ có hiệu lực trong một công ty, cụm từ "*của công ty*" nghĩa là công ty sở hữu lược đồ.

4.2 Ví dụ trên đã xác định được các lớp của hệ thống con *Insurrance*. Ta nên đặt tên (theo UML) cho các lớp là gì? Dùng các tên đường dẫn để xác định vị trí của mỗi lớp.

Lời giải:

Có thể đặt tên cho các lớp là *InsurrancePolicy*, *InsurranceCompany*, *InsurranceScheme* và *CarSharer*. Tuy nhiên, *CarSharer* đã được mô hình hoá như một phần của gói *CarSharers*, nên tên lớp đầy đủ là *CarSharers::CarSharer*.

Giả sử các lớp còn lại là bộ phận của hệ thống con *Insurrance*, thì tên của lớp có kèm tên đường dẫn sẽ là *Insurrance::InsurrancePolicy*, ... Trong ngữ cảnh ở đây, cách viết như vậy là hơi thừa, có thể bỏ đi phần ngữ cảnh và tên của các lớp còn lại lúc này là *Policy*, *Scheme* và *Company*.

4.3 Phí bảo hiểm của hợp đồng phụ thuộc vào địa chỉ nhà của thành viên. Lịch thanh toán cho một hợp đồng sẽ phát sinh một số các giao dịch. Các giao dịch này thuộc hệ thống con *Accounts*. Hãy vẽ một biểu đồ lớp kết hợp với các lớp từ hệ thống con *Insurrance* cùng với các lớp vừa được mô tả. Thêm các kết hợp với nhãn thích hợp vào biểu đồ.

Lời giải: xem hình 4.28

4.4 Các câu hỏi sau kiểm tra hiểu biết của bạn về multiplicity. Bạn hãy dựa vào multiplicity trong hình 4.28 để trả lời cho các câu hỏi này.

a Một *hợp đồng* (Policy) có thể được *phát hành dưới* (issued under) ba *lược đồ* (Scheme) hay không?

b Chúng ta có thể giữ các chi tiết của *lược đồ* mà không cần giữ các chi tiết của *hợp đồng* được *phát hành dưới* lược đồ này không?

c Một *người dùng chung xe* (CarSharer) phải *nhận được* (take out) một *hợp đồng* phải không?

d Một *người dùng chung xe* có thể *nhận được* nhiều hơn một *hợp đồng* hay không?

e Chúng ta có thể giữ chi tiết của một *hợp đồng*, mà không có một *người dùng chung xe* nào *nhận được,* hay không?

Lời giải:

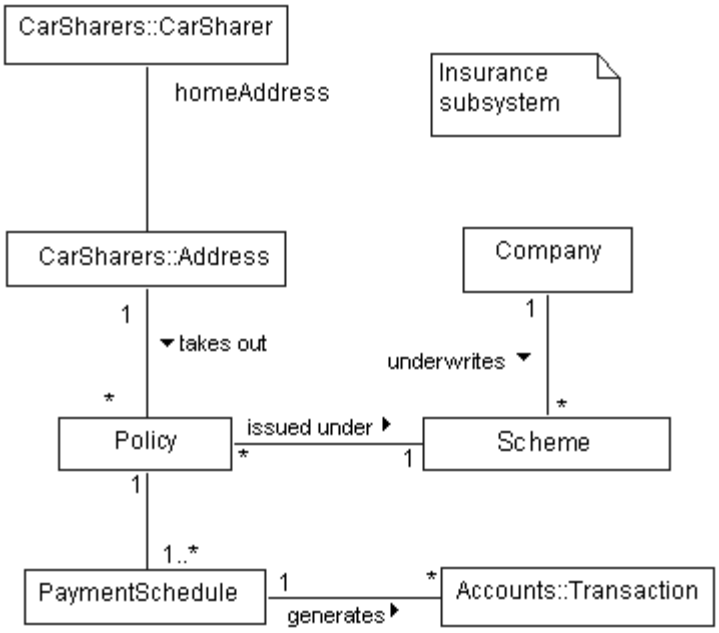
a Không. Một hợp đồng chỉ có thể được phát hành dưới một và chỉ một lược đồ mà thôi.

b Được. Một lược đồ có thể không có bất kỳ hợp đồng nào được phát hành dưới nó (* = 0..*).

c Không.

d Có.

e Không. Một hợp đồng phải được trao cho một (1 = 1..1) thành viên.

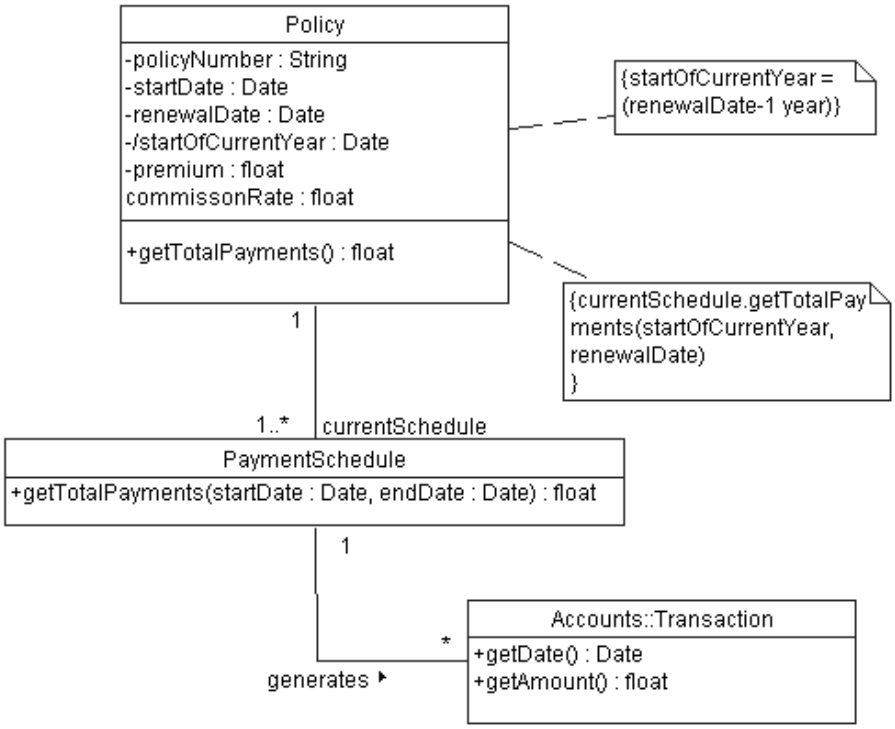


Hình 4.28: Biểu đồ lớp biểu diễn các lớp và các kết hợp

4.5 Đối với mỗi hợp đồng bảo hiểm, ta cần biết đến số hợp đồng, ngày bắt đầu hợp đồng, khi nào gia hạn và tỉ lệ hoa hồng trên phí bảo hiểm. Ngày bắt đầu được tính từ khi hợp đồng được nhận lần đầu tiên. Ngày gia hạn là ngày kết thúc hợp đồng. Chúng ta cũng cần biết tổng số tiền phải trả cho một hợp đồng (trong năm hiện tại của hợp đồng). Năm hiện tại được xác định là ngày gia hạn trừ đi một năm. Hãy xác định các thuộc tính và các thao tác. Đặt chúng vào các lớp. Phác thảo kiểu thích hợp và xác định *visibility* cho chúng.

Lời giải:

Rõ ràng ở đây có một vài thuộc tính cho Policy, dễ nhận ra là các thuộc tính *PolicyNumber*, *startDate*, *renewalDate*, *premium* và *commisionRate*. Thuộc tính dẫn xuất, *startOfCurrentYear* cũng được kèm vào. Đặc tả cho thuộc tính dẫn xuất này có thể đặt trong một ghi chú. Tất cả các thuộc tính này đều là *private*. (xem hình 4.29)



Hình 4.29: Biểu đồ lớp biểu diễn attribute và operation

Với thao tác, hợp đồng giờ đây có một thao tác *getTotalPayments*. Ghi chú trên biểu đồ chỉ ra rằng phương thức cho thao tác này sẽ gọi thao tác *getTotalPayments* trên thể hiện của *PaymentSchedule*, thực hiện vai trò của *currentSchedule*. Các giá trị, kiểu Date, của *startOfCurrentYear* và *renewalDate* cũng được truyền.

Biểu đồ lớp được xác định với giả thiết rằng mỗi giao dịch được thực hiện dựa vào vai trò *currentSchedule*. *PaymentSchedule* sẽ được lựa chọn nếu vào năm xác định (dùng thao tác *getDate* và so sánh giá trị nhận được với *startOfCurrentYear* và *renewalDate*). Nếu giao dịch không rơi vào năm hiện tại thì giá trị của giao dịch sẽ được phục hồi thao tác *getAmount* và được thêm vào tổng số. Hành vi này được xác định trong biểu đồ tương tác (xem chương 8 và 9).

Chú ý có sự mập mờ về định hướng của biểu đồ ở đây. Xét thao tác *getTotalPayments* trong *Policy*, biểu đồ lớp đã chỉ ra một đặc tả để hỗ trợ cho thao tác này. Các thuộc tính và các thao tác khác hỗ trợ cho chức năng này lại không được chỉ ra. Ví dụ, giao dịch sẽ cần có các thuộc tính để thể hiện ngày và giá trị của giao dịch. Các thông tin này chưa được chỉ ra. Bằng cách thêm vào các thuộc tính và các thao tác còn thiếu này, chúng ta có thể giải quyết được sự khác biệt giữa định hướng khái niệm và định hướng đặc tả cũng như cung cấp các định nghĩa hoàn chỉnh cho các lớp trong hệ thống này.

Bài tập bổ sung

4.6 Đọc lại tài liệu của case study *VolBank* trong chương 1, hãy liệt kê các lớp tìm được dựa vào phần mô tả, sau đó hãy phác thảo biểu đồ lớp đầu tiên.

4.7 Dưới đây là một đoạn trích từ cuộc phỏng vấn giữa Said Huffsain (phân tích viên hệ thống) và Martin Page (giám đốc tuyển dụng của VolBank):

Said Huffain: Xin ông vui lòng cho tôi biết thêm một ít về thông tin mà ông cần lưu giữ đối với các tình nguyện viên.

Martin: À, chúng tôi cần biết các thông tin chẳng hạn như tên, các chi tiết liên lạc.

SH: Chi tiết liên lạc à?

MP: Vâng, số điện thoại và địa chỉ e-mail của họ. Ngoài ra tuổi của họ là một thông tin có ích.

SH: Vậy ông có giữ ngày sinh của tình nguyện viên hay không?

MP: Có, bởi điều này sẽ phân biệt được hai tình nguyện viên trùng tên.

SH:...

Hãy xác định các thuộc tính (cả dẫn xuất) trong đoạn trích này và đặt chúng vào lớp thích hợp. Với các thuộc tính dẫn xuất, hãy đính kèm một đặc tả dạng ghi chú. Phác thảo kiểu thích hợp cho mỗi thuộc tính.

4.8 Dựa vào đoạn phỏng vấn dưới đây, hãy xác định các kết hợp giữa các lớp mà bạn đã xác định, đồng thời chỉ ra các multiplicity giữa các kết hợp này.

Said Huffsain: Các tình nguyện viên liên quan như thế nào với các nhu cầu cần giúp đỡ?

Martin Page: Các tình nguyện viên sẽ *gởi* thời gian biểu cho chúng tôi. Đó là thời gian biểu mà họ sẵn sàng làm việc trong các dự án tình nguyện. Dựa trên đó, chúng tôi có thể kết hợp họ với các nhu cầu thích hợp.

SH: Được rồi. Tôi đã thấy cách mà ông có thời gian biểu của tình nguyện viên. Thế còn dự án thì sao?

MP: Ồ, có một sự bắt đầu tương tự ở đây với việc các tổ chức gởi thời gian biểu trên các dự án khác nhau mà họ cần giúp đỡ.

SH: Thế à. Ông có bao giờ có các dự án được thiết lập bởi nhiều hơn một tổ chức không?

MP: Không, nhưng một tổ chức có thể có vài dự án trên sổ sách của chúng tôi tại bất kỳ thời điểm nào. Tình nguyện viên có thể làm việc trên vài dự án khác nhau, nhưng tại một thời điểm chỉ có thể tham gia trên một dự án mà thôi.

Hướng dẫn:

Dẫn giải cuối cùng của Martin Page chứa dựng một cặp kết hợp trong một giai đoạn. Ta có thể lấy được multiplicity của các kết hợp từ dẫn giải này.

4.9 Dựa vào đoạn trích sau, hãy xác định các thao tác để thực hiện các yêu cầu chức năng trong đoạn trích này.

Said Hussain: Ông có ý định kết hợp các tình nguyện viên với các công việc như thế nào?

Martin Page: Chúng tôi sẽ bắt đầu từ các tình nguyện viên và tìm kiếm các công việc phù hợp cho họ. Chúng tôi cũng có thể bắt đầu từ các công việc và tìm kiếm các tình nguyện viên phù hợp.

SH: Việc kết hợp dựa vào cơ sở thời gian?

MP: Tại điểm này là như vậy. Chúng tôi sẽ xem xét các khả năng khác sau, nhưng hãy bám vào thời gian đã.

SH: Như vậy chúng ta phải kiểm tra thời gian biểu gởi của tình nguyện viên có trùng với thời gian biểu của nhu cầu không?

MP: Vâng, chúng ta phải thực hiện điều đó

SH :...

4.10 Hãy mở rộng các thao tác đã xác định trong bài tập 4.9, thêm một ghi chú đặc tả hành vi của thao tác tìm tình nguyện viên phù hợp với một nhu cầu. Bạn cần dùng tên vai trò để làm rõ vai trò của lớp trong các kết hợp.

4.11 Để kết hợp thời gian biểu *gởi* của tình nguyện viên và thời gian yêu cầu của một dự án, một thao tác *so sánh bằng* là cần thiết. Nếu bạn chưa có thao tác như vậy, hãy thêm nó vào lớp thích hợp, xác định các tham số và kiểu trả về.

4.12 Nếu chưa xác định *visibility* cho tất các các thuộc tính và các thao tác, hãy làm điều này.

Chương 5
BIỂU ĐỒ LỚP KẾT TẬP, HỢP THÀNH VÀ TỔNG QUÁT HOÁ

5.1 Giới thiệu

Chương 4 đã giới thiệu biểu đồ lớp cơ bản với hai phần tử là *lớp* và *kết hợp*. Khởi tạo một kết hợp giữa hai lớp ngụ ý cho biết có sự cộng tác giữa chúng thông qua việc gởi thông điệp. Chương này sẽ giới thiệu một số ký hiệu mở rộng đối với các khái niệm lập mô hình cơ bản này.

Là mở rộng của chương 4, cho nên chương này sẽ không lặp lại mục "*Biểu đồ lớp trong UP*" và "*Quan hệ với các biểu đồ khác*".

5.2 Mục đích của kỹ thuật

Khi vẽ các kết hợp giữa các lớp trên biểu đồ, các kết hợp có thể được tạo ra với tên gọi *gồm có* (consists of) hoặc *được cấu thành* (is made up of). Trong lúc ta muốn có một kết hợp diễn tả tinh tế hơn ngữ nghĩa *"các đối tượng của lớp này bao gồm các đối tượng của lớp kia"*. Đây là mục đích của *aggregation* (kết tập) và *composition* (hợp thành). Một ký hiệu khác mô tả tính hợp thành cũng được thảo luận trong chương này, đó là *đối tượng ghép* (composite object).

Khái niệm *tổng quát hóa* (generalizaton) cho phép thừa kế các *đặc trưng* giữa các lớp. Các đặc trưng được thừa kế có thể là thuộc tính, thao tác và các thành phần kết hợp. Tổng quát hóa trong các hệ thống hướng đối tượng là một cấu trúc rất mạnh, có thể làm tăng khả năng sử dụng lại các đoạn mã và tránh được việc lập lại các đoạn mã.

5.3 Ký hiệu của *aggregation* và *composition*

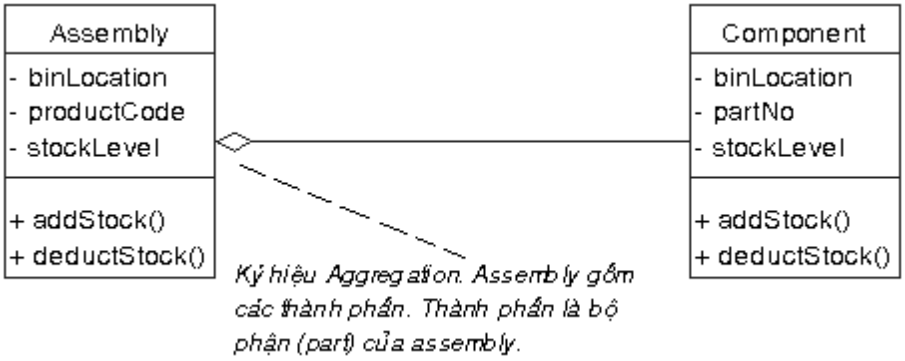
Trong một số trường hợp, kết hợp cho biết đối tượng của một lớp được tạo ra từ các đối tượng của một lớp khác, hoặc đối tượng của một lớp bao gồm các đối tượng của lớp khác. UML có hai loại kết hợp đặc biệt dùng để để biểu diễn điều này, đó là kết hợp *aggreagation* (kết tập) và

composition (hợp thành). Cùng với chúng, một ký hiệu mới được bổ sung vào tập ký hiệu đã có từ chương 4. Khái niệm chung của *aggregation* thường được đề cập như là quan hệ *whole-part* (toàn thể - bộ phận) hoặc *part-of* (bộ phận của) – theo đó một lớp là một bộ phận của lớp khác.

5.3.1 Aggregation

Một kết hợp *aggregation* thường được dùng để cho biết một thể hiện của một lớp bao gồm hoặc chứa các thể hiện của một lớp khác. Số lượng các thể hiện được bao gồm là bao nhiêu tuỳ thuộc vào multiplicity ở đầu cuối của kết hợp. Việc sử dụng aggregation không thay thế đặc tả multiplicity.

Một ví dụ điển hình của aggregation là hệ thống gia công, ở đó một *sản phẩm* (assembly) được lắp ráp từ các *thành phần* (component). Kết hợp này có thể được biểu diễn trong biểu đồ lớp bằng một kết hợp aggregation thay vì một kết hợp bình thường (xem hình 5.1).



Hình 5.1: Ký hiệu aggregation

UML đặt một hình thoi ở đầu mút để biểu diễn kết hợp aggregation. Hình thoi luôn được nối với *lớp kết tập*. Cụm từ *whole-part* đôi khi còn được dùng để ám chỉ đến các kết hợp aggregation. Các thể hiện ở một đầu của kết hợp, là *whole*, được tạo ra từ các thể hiện ở đầu bên kia của kết hợp, là *part*. Hình thoi aggregation luôn ở đầu *whole* của kết hợp. Trong hình 5.1, *Assembly* là *whole* và *Component* là *part*.

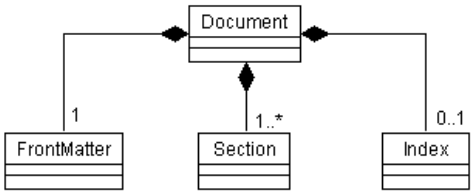
5.3.2 Composition

Aggregation là một kết hợp biểu diễn cấu trúc *whole-part* giữa hai lớp. Với kết hợp composition cũng vậy, tuy nhiên một kết hợp composition còn hàm ý *cùng gắn kết* (coincident lifetime, theo OMG). Một *coincident lifetime* có nghĩa là khi đầu *whole* của kết hợp được tạo thì các thành phần của đầu *part* cũng được tạo. Khi đầu *whole* bị xoá, các thành phần

part cũng bị xoá theo. Nói cách khác, một *part* không thể tồn tại nếu nó không phải là một phần của một *whole*.

Với aggregation thì không phải vậy, một *part* có thể tồn tại bên ngoài kết hợp *whole-part*. Trong hình 5.1, một thành phần có thể tồn tại mà không nhất thiết phải là một *part* của một sản phẩm. Tương tự một sản phẩm có thể được tạo ra từ các thành phần có trước. Composition là một hình thức chặt chẽ hơn aggregation.

Trong UML có hai ký hiệu cho composition. Ký hiệu thứ nhất, được sử dụng khá phổ biến, giống với aggregation ngoại trừ hình thoi được tô đậm. Hình 5.2 là một ví dụ minh họa cho ký hiệu này. Trong hình, một *tài liệu* (Document) gồm một *chủ đề* (FrontMatter), một hoặc nhiều *phần* (Section) và một *chỉ mục* (Index). Biểu diễn ngụ ý, các bộ phận *FrontMatter*, *Section* và *Index* không tồn tại nếu chúng không phải là bộ phận của một *Document*. Hơn nữa, nếu một *Document* bị xoá thì tất cả các bộ phận cấu tạo nên tài liệu đó cũng bị xoá theo.



Hình 5.2: Ký hiệu Composition

Nếu đối tượng bộ phận (ví dụ như *FrontMatter*, *Section* hoặc *Index*) có multiplicity với cận dưới là 1 (1..) thì bộ phận này nên được tạo khi phía *whole* được tạo. Trường hợp cận dưới là 0 (0..) thì bộ phận này có thể được tạo sau khi *whole* được tạo (nhưng phải trước khi *whole* bị huỷ).

Ví dụ 5.1

Trong biểu đồ lớp hình 5.2, có nhất thiết phải tạo một *Section* khi một *Document* được tạo hay không? Còn đối với *Index* thì sao?

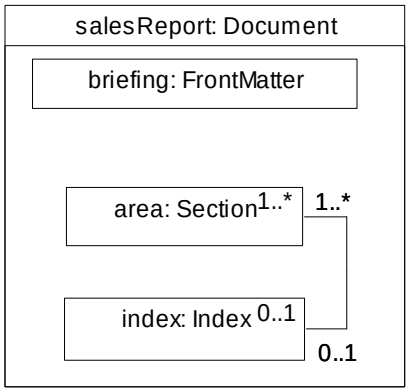
Lời giải:

- Với *Section*, do multiplicity là 1..* phải có ít nhất một *Section* được tạo ra trước khi một *Document* được tạo.
- Với *Index*, do multiplicity là 0..1 nên không cần thiết phải tạo ra một *Index* một cách tự động khi một *Document* được tạo ra.

Ký hiệu thứ hai để biểu diễn composition là dùng *graphical containment* (vẽ một nút bên trong một nút khác). Graphical containment có thể được sử dụng để biểu diễn các lớp và các thể hiện của lớp (tức là các đối

tượng). Thường thì graphical containment được sử dụng để minh hoạ cho composition với các đối tượng hơn là với các lớp.

Hình 5.3 minh họa cho mô tả composition bằng cách dùng graphical containment. Bản chất nền tảng của composition vẫn giống như trong hình 5.2. Multiplicity của các đối tượng bộ phận được biểu diễn ở góc trên bên phải của phần tên của chúng.

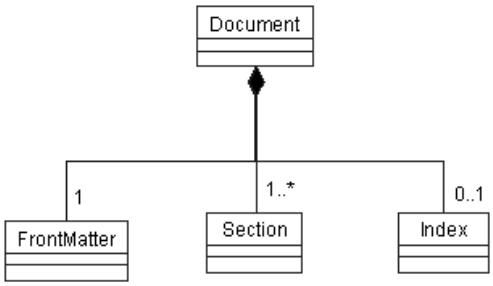


Hình 5.3: Composition dùng graphical containment

Các lớp hoặc các đối tượng được sử dụng trong dạng composition này có thể kết hợp bên trong composition hoặc có thể có kết hợp với các nút bên ngoài composition. Nếu kết hợp đó liên kết đến hai đối tượng bên trong composition thì điều đó ngụ ý rằng kết hợp cũng là một bộ phận của composition. Kết hợp giữa *area : Section* và *index : Index* minh hoạ cho điều này (kết hợp này không được mô tả trong hình 5.2).

5.3.3 Các đường nối dùng chung

Nếu có nhiều kết hợp từ một lớp cùng là aggregation hoặc composition thì chúng có thể được vẽ đồng quy về một biểu tượng aggregation hoặc composition. Hình 5.4 là một ví dụ, thay vì vẽ như trong hình 5.2.

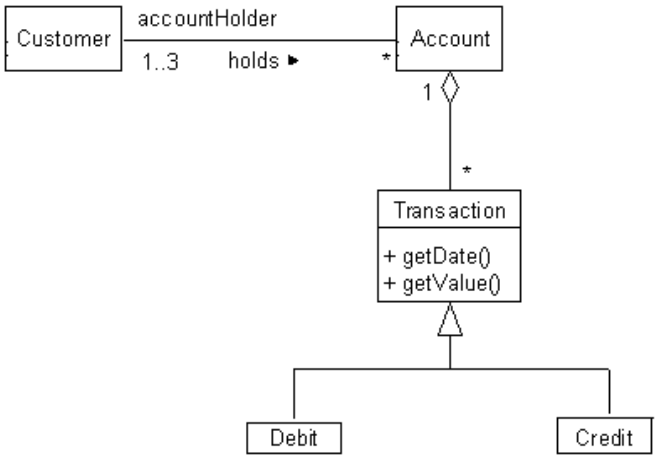


Hình 5.4: Aggregation *với các đường nối đồng quy*

5.4 Ký hiệu của *generalization*

Genaralization (tổng quát hoá) là một hình thức quan hệ khác giữa hai lớp. Vì vậy nó không dùng ký hiệu kết hợp đã khảo sát trong chương 4 (cũng không dùng các ký hiệu aggregation và composition vừa được trình bày).

Quan hệ generalization đôi khi được mô tả như là một quan hệ "*kind of*". Để dễ hiểu, chúng ta hãy khảo sát một ví dụ, hình 5.5 trình bày một phiên bản mở rộng của biểu đồ lớp ATM trong chương 4.

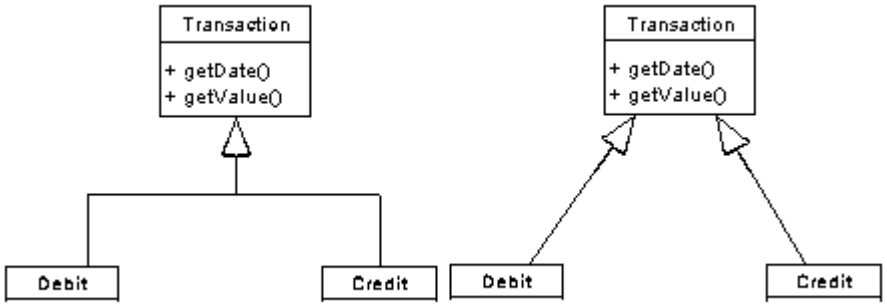


Hình 5.5: ATM được mở rộng với thừa kế

Generalization được biểu diễn trong UML bằng một đường thẳng kết thúc bằng một hình tam giác hướng đến lớp tổng quát hơn. Ví dụ, trong hình 5.5 có một quan hệ generalization giữa *Debit* với *Transaction* (giao dịch) và giữa *Credit* với *Transaction*. Hai quan hệ generalization này có các đường nối đồng quy. Chúng cho biết rằng cả *Debit* và *Credit* đều là *loại* (kind of) *Transaction*. *Transaction* là một tổng quát hoá của *Debit* và *Credit*. Ngược lại, *Debit* và *Credit* là các *chuyên biệt hoá* (specialization) của *Transaction*.

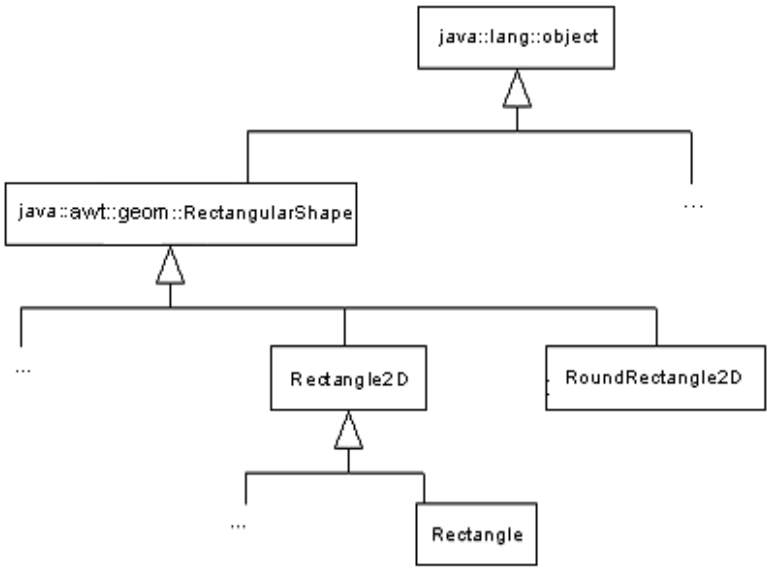
Trong một quan hệ generalization, các *chuyên biệt hoá* (specialization) được biết như là các lớp con (*subclass* hay *subtype*). Trong hình 5.5, *Debit* và *Credit* là hai lớp con. Lớp tổng quát hoá được gọi là lớp cha (*superclass* hay *supertype*). Trong hình 5.5, *Transaction* là một lớp cha. Generalization cho phép lớp con thừa kế các thuộc tính và các thao tác của lớp cha. Trong ví dụ ở hình 5.5, thông điệp *getDate* có thể được truyền cho một thể hiện của lớp *Debit* vì thao tác này được thừa kế từ lớp *Transaction*.

UML cho phép quan hệ generalization có đường nối đồng quy chia sẻ một đầu tam giác hoặc có các đường nối độc lập (xem hình 5.6).



Hình 5.6: Hai cách biểu diễn đường dẫn cho quan hệ generalization

Cấu trúc generalization đôi khi được xem như cấu trúc *phân cấp* (hierarchy), phản ánh cấu trúc cây của chúng. Không có lý do gì để một *cây phân cấp tổng quát hoá* (generalization hierarchy) không có thêm một vài cấp. Khi đó, một lớp con cũng là một lớp cha. Hình 5.7 minh hoạ một *cây phân cấp*, với ví dụ được lấy từ thư viện lớp đồ họa Java (của Sun Microsystem). Trong UML, dấu ba chấm (...) cho biết có các lớp con nhưng không được chỉ ra trên biểu đồ. Ký hiệu này được dùng trong hình 5.7, hình này không thể hiện toàn bộ *cây* ở mọi cấp.



Hình 5.7: Cấu trúc phân cấp tổng quát hóa *trong Java*

Như được minh họa trong hình 5.7, quan hệ giữa lớp con và lớp cha có thể được mở rộng. Một lớp là tổng quát hóa của lớp khác, có thể qua nhiều *tầng*, được gọi là lớp *tiền bối* (ancester) của các lớp sau. Ở đây, *java::lang::Object*, *Rectangle2D* và *java::awt::geom::RectangularShape* đều là tiền bối của *Rectangle*. Lớp tiền bối gần nhất của một lớp được gọi là *lớp cha* (parent class) của nó.

Ngược lại, lớp chuyên biệt hoá của một lớp, bất chấp số tầng, được gọi là lớp *hậu duệ* (descendant). Trong hình 5.7, *Rectangle*, *Rectangle2D* và *RectangularShape* đều là các hậu duệ của *java::lang::Object*.

Ta vẫn thường dùng thuật ngữ *lớp con* và *lớp cha*, thay cho *hậu duệ* và *tiền bối*, nếu quan hệ generalization giữa chúng là trực tiếp trong cây phân cấp.

Hình 5.7 cũng cho thấy một lớp trong một gói có thể là một chuyên biệt hoá của một lớp trong một gói khác. Ví dụ, lớp *Object* ở trong gói *java::lang*, trong khi đó lớp *RectangularShape* và các hậu duệ của ở trong gói *java::awt::geom*.

Hiểu về tổng quát hóa phụ thuộc vào định hướng của biểu đồ lớp. Trong mô hình hướng quan niệm, tổng quát hóa chỉ ra rằng một thể hiện của một lớp cũng là một thể hiện của một lớp khác. Với mô hình hướng đặc tả hoặc cài đặt, tổng quát hóa ngụ ý rằng một đặc tả của thao tác phải được lớp con hỗ trợ. Ví dụ trong hình 5.5, *Debit* và *Credit* phải hỗ trợ các thao tác *getValue* và *getDate*. Các lớp này cài đặt các thao tác như thế nào là không được xác định. Việc thừa kế chỉ có thể cho biết chúng phải cung cấp các thao tác đó mà thôi. Nghĩa là cài đặt của một thao tác, và các thuộc tính thực tế được dùng trong cài đặt này, có thể khác nhau giữa lớp con với lớp con hoặc giữa lớp con với lớp cha.

Cùng với việc thừa kế các đặc trưng và các thành phần kết hợp, lớp con có thể mở rộng lớp cha với các trách nhiệm cụ thể. Ví dụ trong hình 5.7, lớp *RoundRectangle2D*, từ môi trường Java, cung cấp một hình chữ nhật với các góc được bo tròn. Các cung tròn tại các góc của hình chữ nhật được hỗ trợ bởi các thao tác riêng như *getArcHeight*(), *getArcWidth*() trả về kích thước của cung tròn, các thao tác khác như *getBounds*(), thừa kế từ lớp *RectangularShape* cũng được hỗ trợ.

Trong mục 4.4.3, đã giới thiệu ý tưởng về tính khả kiến *protected* đối với thuộc tính và thao tác. Các thuộc tính và thao tác *protected* của một lớp có thể được truy xuất bởi các hậu duệ và không được truy cập bởi các lớp không phải là hậu duệ. Với các thành phần *private*, các lớp hậu duệ hoặc bất kỳ lớp nào khác đều không thể truy xuất được chúng.

Cần chú ý đến việc coi ký hiệu *kind of* như là một kiểu nghĩ về tổng quát hoá đang tiềm ẩn sự sai sót. Có thể có một chút ít mơ hồ khi xem xét các đặc tả chính xác hơn về hành vi của thao tác trên một lớp. Một ký hiệu đúng và hình thức hơn chính là một sự thay thế. Vì cuốn sách này thuần tuý nói về UML hơn là bản chất của mô hình hoá định hướng đối tượng, nên chúng ta không bàn xa hơn về sự thay thế. Chi tiết về sự thay thế có thể tìm thấy ở Priestley (2000, pp. 127-128) và Bennett cùng các cộng sự (1999, p 289).

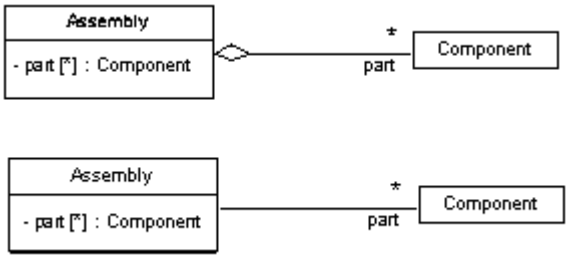
5.5 Hướng dẫn lập mô hình

5.5.1 Aggregation và composition

Aggregation và composition không được phép dùng một cách tuỳ tiện. Ví dụ Fower & scott (1997) thảo luận về kết hợp giữa một tổ chức và các nhân viên của tổ chức đó. Có phải quan hệ là aggregation không? Biểu đồ lớp có nên gợi ý rằng một tổ chức là một aggregation của các nhân viên hay không?

Trả lời cho câu hỏi có dùng aggregation hay không tuỳ vào ngữ cảnh ở đầu *whole* và đầu *part* của kết hợp. Các kết hợp khác mà chúng tham gia vào? Toàn bộ mục đích của việc dùng các lớp trong hệ thống mong muốn là gì? Chúng có được xử lý cùng nhau (như một aggregation) hoặc độc lập nhau hay không?

Ví dụ với *Assembly* và *Component* ở trên, cả *Assembly* và *Component* được xử lý cùng nhau, nên kết hợp aggregation được sử dụng. Khi mô hình các khái niệm như nhân viên trong phòng ban hoặc một tổ chức, thì nhân viên hoặc phòng ban có thể được xử lý độc lập với nhau, nên kết hợp bình thường sẽ thích hợp hơn.



Hình 5.8: Cài đặt của kết hợp có và không có aggregation

Nhớ rằng aggregation là một ký hiệu theo hướng quan niệm. Cài đặt một kết hợp có thể được chỉ dẫn bởi thông tin biểu diễn trên biểu đồ lớp, nhưng cài đặt không nhất thiết phải tuân thủ theo thông tin đó. Hình 5.8 minh họa hai cách mô hình hoá một kết hợp, một với aggregation và

một thì không. Ta cũng thấy biểu đồ cài đặt kết hợp như là một thuộc tính của *Assembly*, ở đầu *Component* của kết hợp với multiplicity là *, là giống nhau cho cả hai trường hợp.

Nhớ rằng kết hợp *composition* ngụ ý thể hiện *part* sẽ bị xóa một khi đối tượng *whole* bị xóa. Căn cứ điều này và nhận xét về aggregation ở trên, ta thấy composition nên được sử dụng cẩn thận hơn so với aggregation. Nếu composition được sử dụng trên một kết hợp, nó sẽ mang ý nghĩa rất cụ thể và được ràng buộc một cách chặt chẽ hơn khi cài đặt.

5.5.2 Generalization

Khi phát triển biểu đồ lớp, bất kỳ một cây *tổng quát hóa* nào trong mô hình lớp đều cần được xác định và mô hình. Đây là điều mong muốn bởi nó khuyến khích việc tái sử dụng cũng như cải thiện tính nhất quán về đặc tả và cài đặt.

Từ nội dung thảo luận trong mục 5.4, chúng ta phải biết rằng định hướng của biểu đồ lớp sẽ ảnh hưởng đến cách hiểu về tổng quát hóa. Khi làm việc trên mô hình quan niệm, một thể hiện của một lớp con cũng sẽ là một thể hiện của lớp cha. Trên biểu đồ đặc tả hoặc cài đặt, ta dùng tổng quát hóa để biểu diễn đặc tả hoặc mô tả các thao tác sẽ được cài đặt trong lớp con.

Mô hình các cây tổng quát hoá hàm ý đến các đặc trưng xác định trong phần mã cài đặt. Trước khi xem xét các tiếp cận, ta cần hệ thống lại những ngụ ý này bởi chúng có liên quan đến phần thảo luận lập mô hình tới đây.

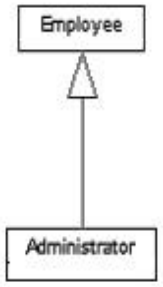
Mở rộng các đặc trưng của lớp: Các lớp con thừa kế tất cả các thuộc tính và thao tác của (các) lớp cha. Việc thừa kế này lặp lên đến đỉnh của cây tổng quát hoá. Ngoài ra lớp con cũng có riêng các thuộc tính và thao tác của nó. Theo cách này, lớp con có thể mở rộng đặc tả của lớp cha. Trong mục 5.4, lớp *RoundRectangle2D* là một ví dụ về sự mở rộng này.

Định nghĩa lại các thao tác: Các thao tác có thể được thừa kế dạng *như là* (as is). Nói cách khác, cả khai báo và phương thức của thao tác đều được thừa kế từ lớp cha. Một lớp con có thể định nghĩa lại phương thức của một thao tác được thừa kế theo ngữ nghĩa của nó. Khai báo của thao tác được thừa kế vẫn giữ nguyên, nhưng phương thức thì được lớp con định nghĩa lại.

Các thao tác *giữ chỗ*: Trong một số trường hợp, thao tác có trong lớp cha chỉ nhằm mục đích bảo đảm rằng các lớp con sẽ hỗ trợ. Trong lớp cha không định nghĩa một phương thức nào cả,

chúng ta muốn lớp con cung cấp phương thức thích hợp. Trên lớp cha, thao tác như vậy là có tính chất trừu tượng (mặc định tính chất này không được hiển thị tường minh trong biểu đồ lớp). Trong ngữ cảnh như vậy, ta đặc tả hình thức toàn bộ lớp cha là trừu tượng. Điều này có nghĩa là nó sẽ không có thể hiện.

Trong một số tình huống xác định, tổng quát hóa được phép dùng dù chỉ có một lớp con xuất hiện tại thời điểm mô hình được phát triển (hình 5.9). Điều này làm tăng tính linh động trong tương lai của mô hình lớp.



Hình 5.9: Tổng quát hóa với một lớp con duy nhất

Ví dụ trong hình 5.9, người phân tích biết rằng trong tương lai sẽ có các lớp con khác của *Employee*. Mỗi lớp con sẽ đặc tả hành vi tổng quát của *Employee* theo cách thích hợp. Tuy nhiên, người phân tích vẫn không tạo ra các lớp con khác vì chưa cần thiết.

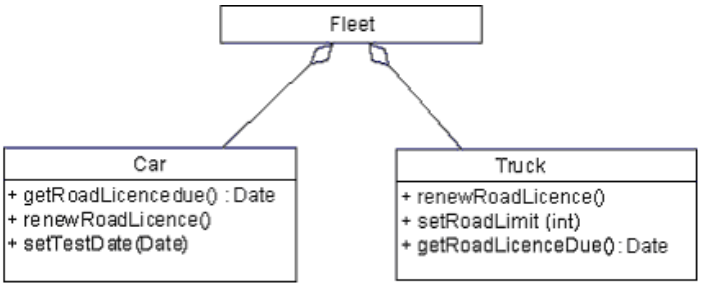
Khi đưa ra mô hình lớp, các cây tổng quát hóa có thể được xác định theo hướng phân tích *bottom-up* (từ dưới lên) hoặc *top-down* (từ trên xuống).

5.5.2.1 Tổng quát hóa bottom-up

Khi các mô hình lớp được đưa ra, các lớp chia sẻ các đặc trưng, trách nhiệm và cộng tác trở nên rõ ràng. Các đặc trưng chung của những lớp này có thể được tổng quát hoá trong một lớp cha thích hợp, các cây tổng quát hóa sẽ được tạo ra giữa lớp cha mới và các lớp gốc (bây giờ chúng trở thành các lớp con).

Ví dụ 5.2

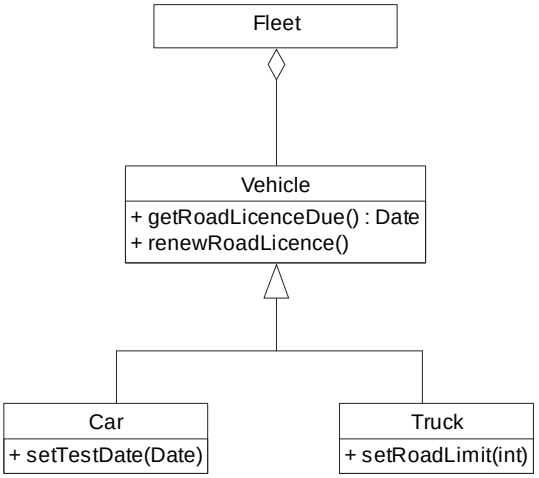
Trong hình 5.10, cả *Car* (xe hơi) và *Truck* (xe tải) đều có chung các nhiệm vụ (các thao tác *getRoadLicenceDue*():*Date* và *renewRoadLicence*()) cũng như có chung các cộng tác (là *fleet consists of cars* và *fleets consists of trucks*). Làm thế nào để các đặc điểm chung được tổng quát hoá một cách hợp lý?



Hình 5.10: Các lớp chia sẻ chung trách nhiệm và cộng tác

Lời giải:

Người phân tích tìm thấy có một sự tương đồng cao giữa *Car* và *Truck* và đã chọn *Vehicle* làm tổng quát hoá cho hai lớp này, xem hình 5.11.



Hình 5.11: Các cộng tác và trách nhiệm chung được tổng quát hoá

Trong ví dụ này, việc tổng quát hoá có vẻ hợp lý. Các trách nhiệm và cộng tác chung được tổng quát hoá thành lớp *Vehicle* mà không làm ảnh hưởng đến ngữ nghĩa của mô hình.

Sự tổng quát hoá có thể được điều khiển bởi việc xác định và trừu tượng hoá thích hợp các đặc trưng được chia sẻ như thuộc tính, thao tác và kết hợp.

5.5.2.2 Tổng quát hoá Top-down

Khi mô hình lớp phát triển, người phân tích cần mô hình các lớp có các đặc trưng chung và chia sẻ các kết hợp chung. Thay vì mô hình những

lớp này cùng với các kết hợp như khảo sát trong chương 4, người phân tích có thể dùng tổng quát hoá.

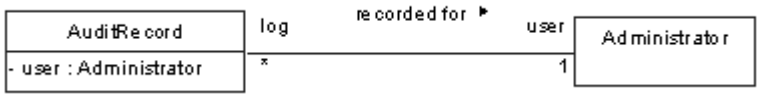
Ví dụ 5.3

Dựa vào đoạn phỏng vấn sau, hãy xác định các lớp cần thiết và xây dựng một cây tổng quát hoá thích hợp.

> **Janet Hoffner:** Chúng tôi muốn hệ thống có thể ghi nhận mỗi khi người quản trị thay đổi các bản ghi trong hệ thống. Điều này có thực hiện được hay không?
>
> **Mick Perez:** Được. Chúng ta đang nói về việc các quản trị viên thay đổi các bản ghi hiện tại phải không?
>
> **JH:** Đúng vậy.
>
> **MP:** À, nhưng sẽ có những người dùng khác có thể thay đổi các bản ghi của hệ thống trong tương lai có phải không?
>
> **JH:** Vâng, tôi nghĩ vậy. Kế hoạch ban đầu của chúng ta là để cho quản trị viên làm tất cả các công việc trên bàn phím, nhưng tôi nghĩ rằng không phải trong mọi trường hợp.

Lời giải:

Đầu tiên, Mick nghĩ rằng sẽ mô hình hoá các yêu cầu trên như trong hình 5.12. Trong phác thảo này, lớp *AuditRecord* kết hợp trực tiếp với lớp *Administrator*. Mick cũng chỉ ra một cài đặt của kết hợp này như một thuộc tính của *AuditRecord*.



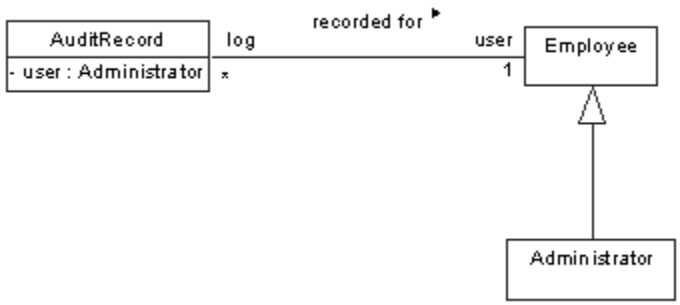
Hình 5.12: Cấu trúc lớp không có tổng quát hoá

Sau đó Mick nhận ra rằng cấu trúc này sẽ không ổn với sự mở rộng trong tương lai. Nếu trong tương lai các bản ghi *AuditRecord* cần giữ lại cho các kiểu người dùng khác, thì cả hai lớp (*AuditRecord* và *Adminsitrator*) cần phải thay đổi.

Mick đi đến kết luận rằng tốt nhất là tổng quát hoá các thuộc tính, các trách nhiệm và các cộng tác của *Aministrator* thành lớp *Employee*. Hình 5.13 trình bày kết quả của sự cải tiến này.

Các thay đổi trong tương lai, chẳng hạn như thêm các lớp con *Employee* khác (ví dụ như *Manager*, *Agent*) sẽ không cần các bổ sung cho lớp *AuditRecord*.

Chỗ nào mà tổng quát hoá được đưa ra theo hướng top-down, phân tích viên phải bảo đảm các thuộc tính, trách nhiệm và cộng tác của lớp cha đều áp dụng hoàn toàn cho các lớp con. Tính tương thích này có thể được kiểm tra bằng một cách hình thức hơn, chẳng hạn như nguyên lý thay thế *Liskov* (Priestley, 2000; Bennett và các cộng sự,1999).



Hình 5.13: Cấu trúc lớp với tổng quát hoá

5.5.3 Các khái niệm hướng đối tượng

5.5.3.1 Trạng thái đối tượng

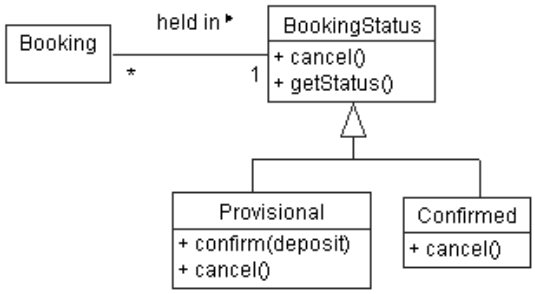
Chương 4 đã giới thiệu khái niệm trạng thái của đối tượng. Xin nhắc lại, trạng thái của đối tượng phụ thuộc vào giá trị của các *thuộc tính chính* nào đó và các thể hiện của các kết hợp. Những thay đổi đến các thuộc tính hoặc các kết hợp này làm thay đổi trạng thái của đối tượng.

Đôi khi cây tổng quát hoá được dùng để quản lý hành vi của đối tượng theo trạng thái. Trong biểu đồ lớp ở hình 5.14, lớp *Booking* (đặt chỗ) chuyển trách nhiệm xử lý, mà cách cư xử của nó lại phụ thuộc trạng thái cụ thể, cho lớp *BookingStatus*. *BookingStatus* có hai lớp con là *Provisional* (tạm thời) và *Confirmed* (đã xác nhận)

Tất cả các lớp con của *BookingSatus* đều hỗ trợ thao tác *getStatus* và *cancel*. Lớp con *Provisional* cung cấp thao tác *confirm* cho phép một *provisional booking* được *confirm* (ví dụ, đặt cọc trước). Phương thức của thao tác *confirm* sẽ thay thể hiện của *BookingStatus* (*BookingStatus.Provisional*) bằng một thể hiện khác (*BookingStatus.Confirmed*). Lớp *Confirmed* không cung cấp thao tác này, vì vậy một *Booking* ở trạng thái *Confirmed* sẽ không được *confirm* nữa. Thể hiện cũ sẽ sao chép tất cả các giá trị thuộc tính của nó cho thể hiện mới (trừ ra các giá trị bị ảnh hưởng bởi việc đổi trạng thái) và sau đó liên kết đối tượng *Booking* với đối tượng trạng thái mới (đối tượng *BookingStatus.Confirmed* mới).

5.5.3.2 Polymorphism (đa hình)

Biểu đồ lớp trong hình 5.14 cũng minh hoạ cho khái niệm *đa hình* trong hướng đối tượng. Đa hình là khả năng của hai thao tác có khai báo giống hệt nhau thi hành cùng một chức năng trừu tượng giống nhau theo các cách khác nhau. Ví dụ, hàm *cancel* trong *Provisional* chỉ đơn giản là trả lại tài nguyên đã đặt (như bàn ghế trong một nhà hàng hoặc các chỗ ngồi trong một rạp phim). Tuy nhiên, thao tác *cancel* trong *Confirmed* phải quản lý tiền đặt cọc. Ví dụ, thao tác *cancel* trong *Confirmed* có thể giải phóng tài nguyên nhưng được đặc tả yêu cầu khách hàng khác giữ chỗ (và trả tiền đặc cọc) trước khi trả lại tiền cọc cho khách hàng trước.



Hình 5.14: Minh họa việc theo dõi trạng thái dùng tổng quát hóa

5.6 Ký hiệu tổng quát hóa nâng cao

UML cũng cung cấp các ký hiệu tổng quát hoá khác có thể được dùng để mở rộng ký hiệu tổng quát hoá thông dụng đã được đề cập trong mục 5.4.

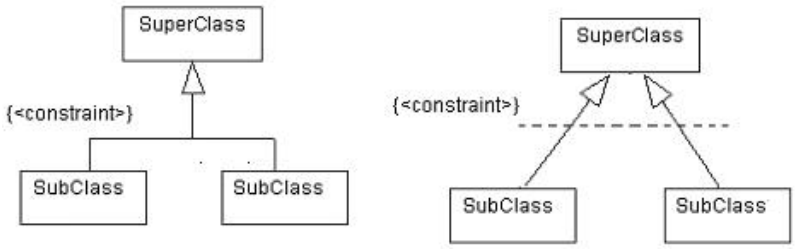
5.6.1 Chú thích của tổng quát hóa

Cấu trúc tổng quát hoá có thể được chú thích bởi các *ràng buộc* (constraint) và các *discriminator* (phân biệt). Ràng buộc được dùng trong cây tổng quát hoá nếu như các lớp con là chủ đề cho các điều kiện ngữ nghĩa nào đó. *Discriminator* có thể được xem như tên vai trò của các tổng quát hoá, dùng để làm sáng tỏ ngữ nghĩa của tổng quát hoá hoặc cho phép nhiều tổng quát hoá từ một lớp cha.

5.6.1.1 Ràng buộc

Các ràng buộc tổng quát hoá có thể được định nghĩa trước hoặc được người dùng định nghĩa. Ký hiệu UML cho các ràng buộc được minh họa trong hình 5.15. Bình thường, ràng buộc được đặt trong cặp dấu ngoặc móc ({..}). Nơi nào dùng đường nối chung, các ràng buộc có thể được đặt ở phía lớp cha, gần mũi tên hình tam giác (hình 5.15, bên trái). Nơi nào dùng đường nối riêng biệt thì một đường thẳng đứt nét được vẽ ngang

qua các đường nối mà ràng buộc áp đặt. Tên của ràng buộc được viết gần đầu mút của đường đứt nét này (hình 5.15, bên phải).

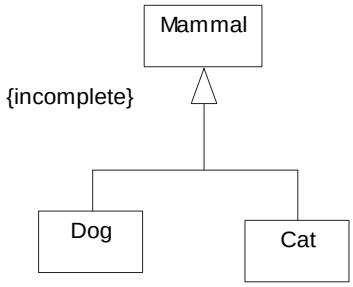


Hình 5.15: Ký hiệu ràng buộc tổng quát hóa

UML định nghĩa trước 4 ràng buộc thông dụng sau đây:

- **Incomplete (không đầy đủ)**: cho biết chưa xác định hết các lớp con.

  Ví dụ hình 5.16 minh họa cách chuyên biệt hoá *Dog* và *Cat* không là toàn bộ các chuyên biệt hoá của *Mammal*.

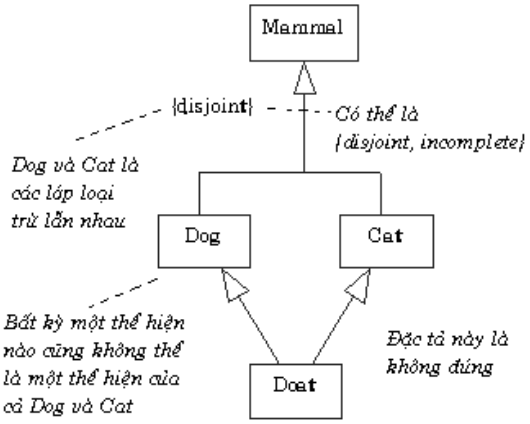


Hình 5.16: Minh hoạ cho ràng buộc không đầy đủ

- **Complete (đầy đủ)**: cho biết tất cả các lớp con đều đã được xác định. Nên nhớ không phải tất cả chúng đều được hiển thị trên một biểu đồ.
- **Disjoint (rời)**: cho biết một thể hiện chỉ có thể là thể hiện của một trong số các lớp con. Nếu một trong chúng có hậu duệ thì hậu duệ này không thể thừa kế các lớp con *disjoint* khác. Các lớp con *disjoint* là không tương đồng. Một thể hiện của một trong chúng không thể là một thể hiện của lớp con khác trong chúng.

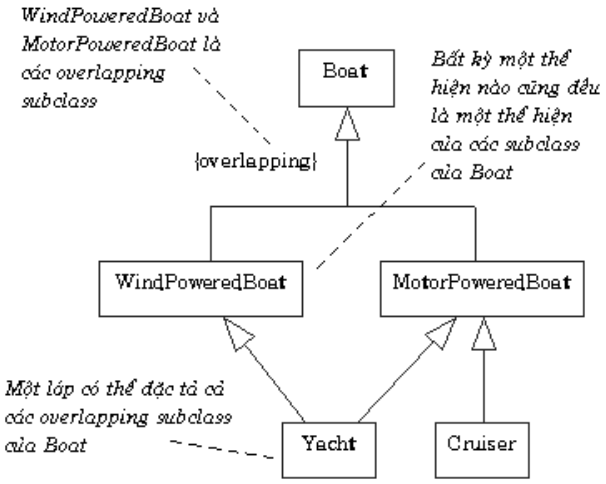
  Hình 5.17 minh họa điều này bằng cách dùng lại *Mammal*, *Dog* và *Cat*. Ràng buộc *disjoint* xác định một thể hiện không thể là thể hiện của cả *Dog* và *Cat*, cũng vậy sẽ không có các lớp con thừa kế cả hai *Dog* và *Cat*. Lớp *Doat*, chuyên biệt hoá của *Dog* và

Cat như trong hình 5.17 là không đúng vì giữa *Dog* và *Cat* có ràng buộc *disjoint*.



Hình 5.17: Ví dụ minh hoạ cho ràng buộc disjoint

- **Overlapping (chập)**: ngược với *disjoint*, ràng buộc này cho biết một thể hiện có thể là thể hiện của nhiều hơn một lớp con. Các hậu duệ của một trong chúng có thể là hậu duệ của lớp khác trong chúng, xem hình 5.18.



Hình 5.18: Ví dụ minh hoạ cho ràng buộc overlapping

Trong ví dụ này, có thể có thể hiện của cả *WindPoweredBoat* (thuyền buồm) và *MotorPoweredBoat* (thuyền máy). Các lớp con có ràng buộc *overlapping* là không loại trừ nhau. Theo bản chất của *overlapping*, ràng buộc này xác định các lớp con sâu hơn có

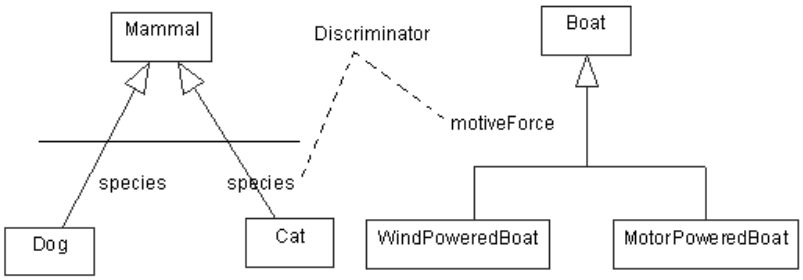
thể thừa kế cả *WindPoweredBoat* và *MotorPoweredBoat*. Lớp *Yacht* (du thuyền), chuyên biệt hoá của cả hai lớp con *WindPoweredBoat* và *MotorPoweredBoat* (hình 5.18), là hoàn toàn chấp nhận được.

5.6.1.2 Discriminator

Discriminator trên quan hệ tổng quát hoá tương tự như *tên vai trò* của kết hợp. Nó cung cấp tên cho tổng quát hoá. Tên của *discriminator* phải khác với tên các đặc trưng sau đây của lớp cha:

1. Tên của một kết hợp (bắt đầu hoặc kết thúc ở lớp cha).
2. Tên của một thuộc tính.

Discriminator được đặt gần mũi tên hình tam giác đối với các đường nối chung, hoặc gần mỗi quan hệ tổng quát đối với các đường nối riêng. Hình 5.19 minh hoạ cho cả hai trường hợp.



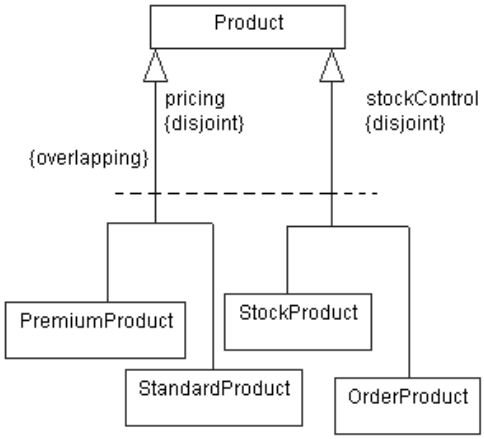
Hình 5.19: Ký hiệu cho các discriminator *tổng quát hóa*

Tên của *discriminator* rất hữu ích trong trường hợp một lớp là lớp cha của hai hoặc nhiều cấu trúc tổng quát hoá. Trong trường hợp này, *discriminator* cho phép phân biệt các cấu trúc tổng quát hoá khác nhau.

Hình 5.20 minh hoạ cách sử dụng cho cả discriminator và ràng buộc. Lớp *Product* là lớp cha của 4 lớp con: *PremiumProduct*, *StandardProduct*, *OrderProduct* và *StockProduct*.

Theo hình, các lớp con *pricing* đưa ra các đặc trưng liên quan đến xử lý giá dựa trên chức năng (ví dụ như *discount*, *markup*, *calculateProfit*). Tương tự, các lớp con *stockControl* đưa ra các đặc trưng liên quan đến xử lý tồn kho (ví dụ như *stockLevel*, *reorderLevel*, *leadTime*, *reorderStock*).

Cả hai cây tổng quát hoá *pricing* và *stockControl* đều có ràng buộc *disjoint*, nghĩa là một *Product* hoặc là một *PremiumProduct* hoặc là một *StandardProduct*. Tương tự, một *Product* hoặc là một *StockProduct* hoặc là một *OrderProduct*.



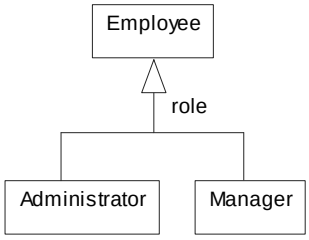
Hình 5.20: Sử dụng discriminator *với nhiều quan hệ generalization*

Tuy nhiên hai cây tổng quát hoá này lại có ràng buộc *overlapping*. Ràng buộc này cho biết một *Product* có thể là một trong những *pricing* và cũng có thể một trong những *stockControl* trong cùng một thời điểm. Tóm lại, một thể hiện có thể là:

1. Một thể hiện của *Product*.
2. Một *PeremiumProduct*, nhưng không là *stockControl*.
3. Một *StandardProduct*, nhưng không là *stockControl*.
4. Một *StockProduct*, nhưng không là *pricing*.
5. Một *OrderProduct*, nhưng không là *pricing*.
6. Một *PeremiumProduct* và *StockProduct*.
7. Một *PeremiumProduct* và *OrderProduct*.
8. Một *StandardProduct* và *StockProduct*.
9. Một *StandardProduct* và *OrderProduct*.

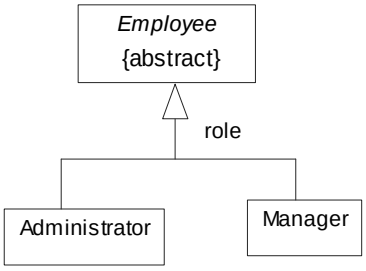
5.6.2 Các lớp cha trừu tượng

Trong một cây tổng quát hoá, một đối tượng có thể là thể hiện của một hoặc nhiều lớp con, hoặc là thể hiện của lớp gốc. Ví dụ, một tổ chức có thể tuyển dụng một người mà không giao cho anh ta một công việc hoặc một vai trò cụ thể nào. Như trong hình 5.21, hai lớp con *Administrator* và *Manager* và lớp cha *Employee* (cũng là lớp gốc của cấu trúc này) đều có thể có thể hiện. Các thể hiện *Employee* biểu diễn các nhân viên nhưng không xác định vai trò là *Administrator* hay *Manager*.



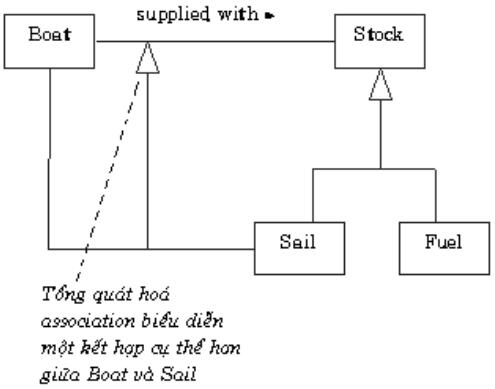
Hình 5.21: Tổng quát hoá với các thể hiện của superclass

Ngược lại, nếu như tổ chức này yêu cầu mỗi người phải có một vai trò cụ thể nào đó, thì *Employee* sẽ không có thể hiện trực tiếp. Để phản ánh yêu cầu này trong biểu đồ lớp, tính chất {*abstract*} trong phần tên lớp được dùng để cho biết lớp *Employee* sẽ không bao giờ có bất kỳ thể hiện trực tiếp nào (xem hình 5.22). Tên của lớp trừu tượng được in nghiêng.



Hình 5.22: Tổng quát hoá với lớp cha trừu tượng

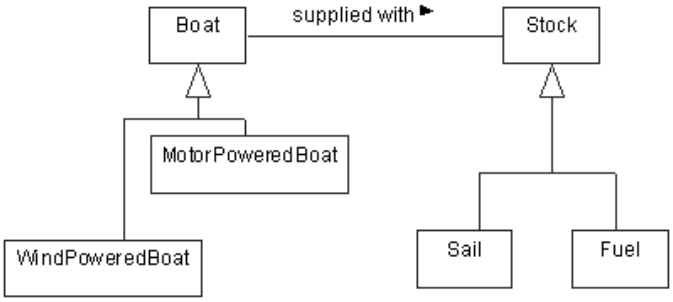
5.6.3 Tổng quát hoá của kết hợp



Hình 5.23: Ví dụ về ký hiệu tổng quát hoá trên kết hợp

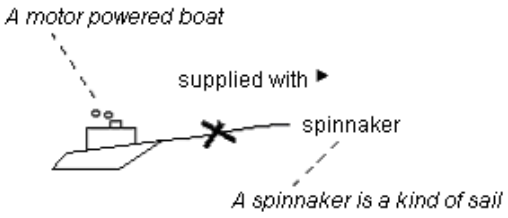
Chúng ta đã thảo luận quan hệ tổng quát hoá trên lớp. Trong mục này chúng ta sẽ tìm hiểu thêm một khía cạnh nữa của ký hiệu tổng quát hoá, đó là ký hiệu về tổng quát hoá trên các kết hợp. Ký hiệu tổng quát hoá này tương tự như ký hiệu của tổng quát hoá của lớp, điểm bắt đầu và điểm kết thúc của tổng quát hoá bây giờ là một kết hợp thay vì một lớp như trước đây. Hình 5.23 minh hoạ cho điều này.

Hình 5.23 minh họa một chuyên biệt hoá của kết hợp, nó cho biết trong khi kết hợp *supplied with* (được trang bị) mô tả liên kết giữa *Boat* (thuyền) và *Stock* (trang bị) nói chung, thì bất kỳ một kết hợp nào giữa *Boat* và *sail stock* (cánh buồm) nên được xem xét trong phương diện của kết hợp chuyên biệt hơn (là kết hợp không tên trong hình 5.23). Kết hợp chuyên biệt này có thể có thêm multiplicity hoặc vai trò hoặc ràng buộc.



Hình 5.24: Sự mơ hồ trong kết hợp giữa các lớp cha

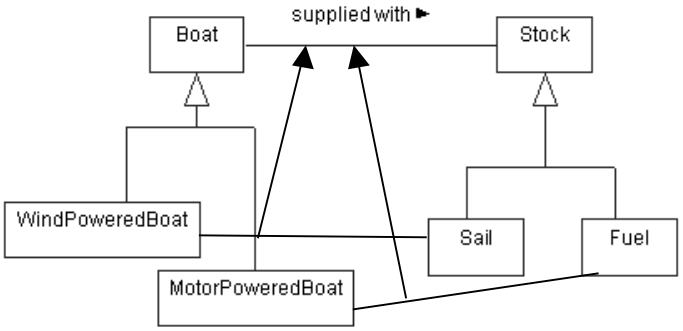
Tổng quát hóa trên kết hợp được dùng nếu như một kết hợp giữa các lớp cha dẫn đến một sự mơ hồ hoặc không thống nhất nào đó. Hình 5.24 minh họa cho một vấn đề có thể xảy ra. Trong hình, kết hợp giữa *thuyền* và các nhu cầu *trang bị* được biểu diễn bằng một kết hợp giữa hai lớp cha *Boat* và *Stock*. Kết hợp chung chung này có thể quá lỏng lẻo khi nói đến các thể hiện có thể có của nó. Ví dụ trong hình 5.25 minh họa một thể hiện, trong đó một thuyền máy lại được trang bị một cánh buồm. Rõ ràng đây là điều không mong muốn dưới góc độ toàn vẹn.



Hình 5.25: Thể hiện của kết hợp tổng quát hoá

Trong trường hợp này, kết hợp *supplied with* cần được chuyên biệt hoá xa hơn. Hình 5.26 minh họa rõ hơn mối kết hợp này nhờ thêm vào hai kết hợp chuyên biệt. Bây giờ thuyền buồm sẽ được trang bị buồm và thuyền máy sẽ được trang bị nhiên liệu.

Tổng quát hoá của kết hợp sẽ được yêu cầu trong trường hợp như ví dụ trên, nơi mà kết hợp giữa hai lớp cha (gốc) của hai cây tổng quát hoá cần được định nghĩa cụ thể hơn. Tổng quát hoá của kết hợp có thể gây rối cho mắt vì có thêm nhiều đường. Chúng giao nhau làm giảm tính rõ ràng đối với người đọc. Một cơ chế, súc tích hơn và rõ ràng hơn, mô tả ràng buộc kết hợp các lớp con là dùng các ràng buộc OCL trong các ghi chú đính kèm mối kết hợp (xem chương 12)



Hình 5.26: Một kết hợp chuyên biệt hoá liên kết các lớp con

Câu hỏi ôn tập

5.1 Aggregation là gì?

5.2 Các kết hợp aggregation khác với các kết hợp bình thường như thế nào?

5.3 Composition là gì?

5.4 Composition khác với aggregation như thế nào?

5.5 Ký hiệu cho aggregation là gì?

5.6 Hai ký hiệu để biểu diễn composition là gì?

5.7 Tại sao aggregation (hoặc composition) được xem như là quan hệ *whole-part*?

5.8 Tại sao tổng quát hoá được xem như là quan hệ *kind of*?

5.9 Ký hiệu cho tổng quát hoá là gì?

5.10 Hai ký hiệu đường nối nào có thể được dùng cho tổng quát hoá?

5.11 Ràng buộc tổng quát hoá là gì? Bốn ràng buộc tổng quát hoá nào được UML định nghĩa trước? Cho ví dụ về ràng buộc tổng quát hoá.

5.12 Cho ví dụ sử dụng discriminator.

5.13 Trình bày cách biểu diễn súc tích một ràng buộc trên kết hợp tổng quát hoá.

5.14 Tại sao quan hệ aggregation và quan hệ composition có thể làm cho lạc lối trên một biểu đồ lớp?

5.15 Hãy mô tả qui trình top-down xác định các cấu trúc tổng quát hoá.

5.16 Hãy mô tả qui trình bottom-up xác định các cấu trúc tổng quát hoá.

Bài tập có lời giải

Aggregation và composition

5.1 Sau đây là một đoạn trích phỏng vấn giữa Mick Perez và Janet Hoffer. Dựa vào đoạn phỏng vấn, hãy xác định các kết hợp aggregation.

Mick Perez: Thông tin nào ông cần biết về điểm bắt đầu và kết thúc của mỗi lộ trình?

Janet Hoffner: Với mỗi lộ trình, chúng tôi cần biết về *tên và số* của toà nhà, *số phòng*, *đường*, *vùng*, *thành phố hoặc thị trấn*, *quốc gia và mã bưu điện hoặc mã vùng*. Chúng tôi cũng muốn lưu giữ thông tin tương tự đối với *địa chỉ nhà* của *người chia sẻ xe*.

MP: OK. Ý ông muốn nói rằng địa chỉ bắt đầu và kết thúc sẽ được dùng để kết hợp những lộ trình có thể chia sẻ được với nhau phải không?

JH: Vâng – điểm này rất hay. Nhưng tôi hoàn toàn không chắc lắm về cách thực hiện của ông. Chúng tôi muốn xác định có hay không hai địa chỉ là đủ gần nhau để có thể kết hợp chúng chung một lộ trình. Ví dụ, hai người muốn đi từ hai góc kề nhau của hai toà nhà đến hai tầng khác nhau của cùng một toà nhà khác. Ta muốn tìm một địa chỉ là đủ gần với một địa chỉ, nhưng, với kiểu văn bản của địa chỉ, điều này thật khó.

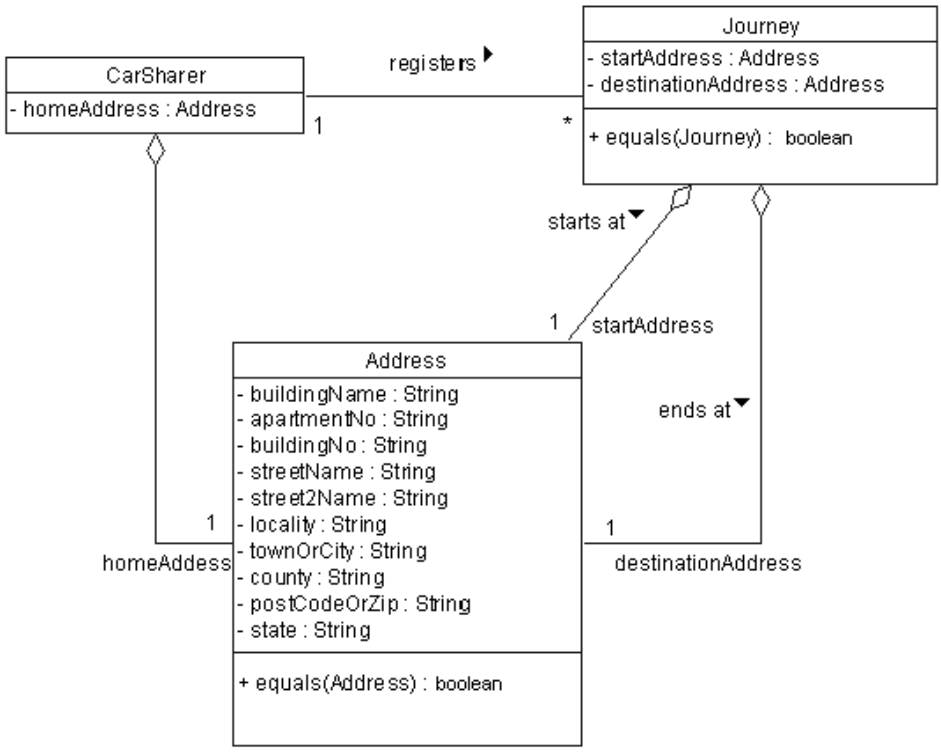
MP:...

Lời giải:

Mick nhận thấy rằng thông tin về địa chỉ là cần cho người chia sẻ xe. Một địa chỉ có thể được lưu giữ như *địa chỉ nhà* của người chia sẻ xe, *địa chỉ bắt đầu* và *địa chỉ kết thúc* của mỗi lộ trình. Giả định rằng thông tin địa chỉ có thể được lưu giữ cho ba yêu cầu trên, Mick đã mô hình hoá một lớp *Address* với các kết hợp đến *CarSharer* và *Journey*. Mick gán tất cả các thông tin địa chỉ mà *CarMacth* cần và một hàm *equals*

(để xác định xem một *Address* có thể được xem là *giống* với một *Address* khác hay không) cho lớp *Address*. Hình 5.27 trình bày biểu đồ lớp của Mick.

Để xác định cài đặt của các kết hợp này, Mick đặt hai thuộc tính *startAddress* và *destinationAddress* vào *Journey* và thuộc tính *homeAddress* vào *CarSharer*. Tất cả những thuộc tính địa chỉ này đều có kiểu *Address* (xem hình 5.27). Hình 5.27 có cả kết hợp aggregation và các thuộc tính đã được cài đặt (*homeAddress, startAddress* và *destinationAddress*).



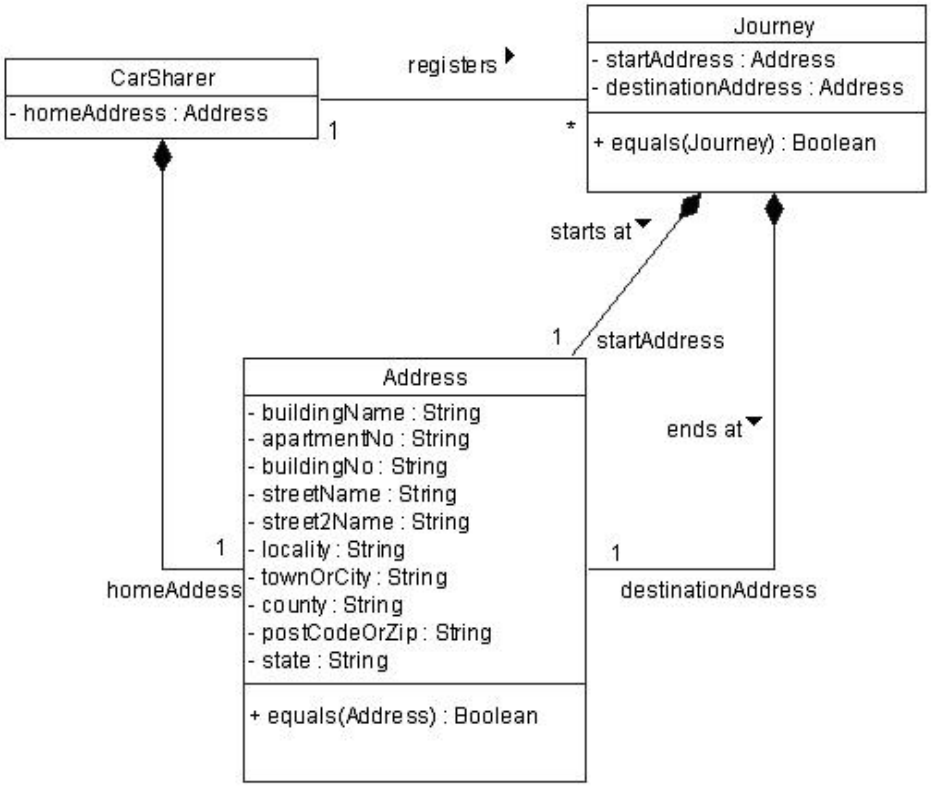
Hình 5.27: Biểu đồ lớp với aggregation Address

5.2 Mick đã xem xét bản chất của lớp *Address* rất cẩn thận sau đó. Anh ấy có dự định tạo ra một địa chỉ mới cho mỗi người chia sẻ xe hoặc cho mỗi lộ trình không? Nếu một người chia sẻ xe đăng ký một hoặc nhiều lộ trình từ địa chỉ nhà của anh ta thì chúng ta tính là một địa chỉ hay nhiều địa chỉ có cùng trạng thái (các giá trị của thuộc tính)?

(Mick đã không xem xét điều này một cách chi tiết, anh ta đã bỏ trống multiplicity của các kết hợp *Journey-Address* ở đầu *Journey* trong biểu đồ lớp ở hình 5.27).

Lời giải:

Mick quyết định rằng mỗi địa chỉ là khác nhau. Việc kết hợp các địa chỉ vẫn như thế bất chấp các địa chỉ có giống nhau hay không. Nếu các địa chỉ được chia sẻ, nghĩa là nhiều người hoặc nhiều lộ trình có thể có cùng một địa chỉ, thì các phương tiện tìm kiếm là cần thiết trong lớp *Address*. Điều này là cần thiết để cho phép các thể hiện mới của *CarSharer* và *Journey* tìm thấy địa chỉ của nó nếu địa chỉ này đã có. Như thế, Mick suy luận rằng hướng tiếp cận trực tiếp nhất là coi các địa chỉ nhà của những người chia sẻ xe cũng như địa chỉ bắt đầu và kết thúc của các lộ trình là khác nhau.



Hình 5.28: Biểu đồ lớp với composition

Theo cách này, Mick nhận thấy rằng *vòng đời* của một thể hiện *Address* trùng với *CarSharer* hoặc *Journey* khi nó được tạo ra. Một thể hiện của *Address* chỉ được công nhận khi được tạo bởi *CarSharer* hoặc *Journey* và sẽ chấm dứt khi *CarSharer* hoặc *Journey* dùng bị xoá đi.

Như vậy, các aggregation có thể thực sự được biểu diễn như là các composition. Hơn nữa, multiplicity của các kết hợp giữa *Journey* và *Address* bây giờ cũng rõ ràng, xem hình 5.28.

Generalization

Tiếp theo Mick tập trung mô hình một số đặc tả về cách đối sánh hai *Journey* với nhau. Xem lại mô hình lớp của mình, Mick ghi chú rằng lớp *Journey* đã chuyển trách nhiệm kiểm tra một địa chỉ có giống với một địa chỉ khác hay không cho lớp *Address*. Việc ủy thác này phát sinh thao tác *equals* trong lớp *Address*.

Bây giờ Mick phải thiết lập cách địa chỉ thực hiện việc đối sánh nó với một địa chỉ khác là có giống nhau hay không. Một so sánh *String* đơn giản là không đủ chính xác cũng như không đủ khả năng đối sánh hai địa chỉ là tương tự nhau. Mick quyết định dựa vào khả năng của *hệ thông tin địa lý* GIS (Geographical Information System).

5.3 Mick đã trao đổi việc xử lý địa chỉ với Jan Cusack, một thành viên trong nhóm, biết GIS.

Jan Cusack: Tôi đã xem kỹ vấn đề đối sánh địa chỉ mà ông đang đề cập.

Mick Perez: Rất tốt, vậy ông có lời khuyên nào cho tôi không?

JC: À, có ba thư viện lớp chuẩn *standard geo-locating* mà ông cần làm việc. Một là *OS* (United Kingdom Ordnance Survey), là hệ thống dựa trên bản đồ, hai là hệ thống dựa trên vĩ độ và kinh độ, cuối cùng là hệ thống tham chiếu *Tiger*.

MP: Chúng làm việc giống nhau chứ ?

JC: Nói chung là như vậy. Việc ông cần làm là đóng gói việc truy cập, đến mỗi thư viện này, bằng các lớp con của mình.

MP: Tôi nghĩ rằng khi đối sánh các địa chỉ, tôi phải để tâm đến việc chuyển đổi giữa định dạng mà tôi có thể dùng trong *CarMatch* và định dạng dùng trong số các thư viện chuẩn mà ông đề cập.

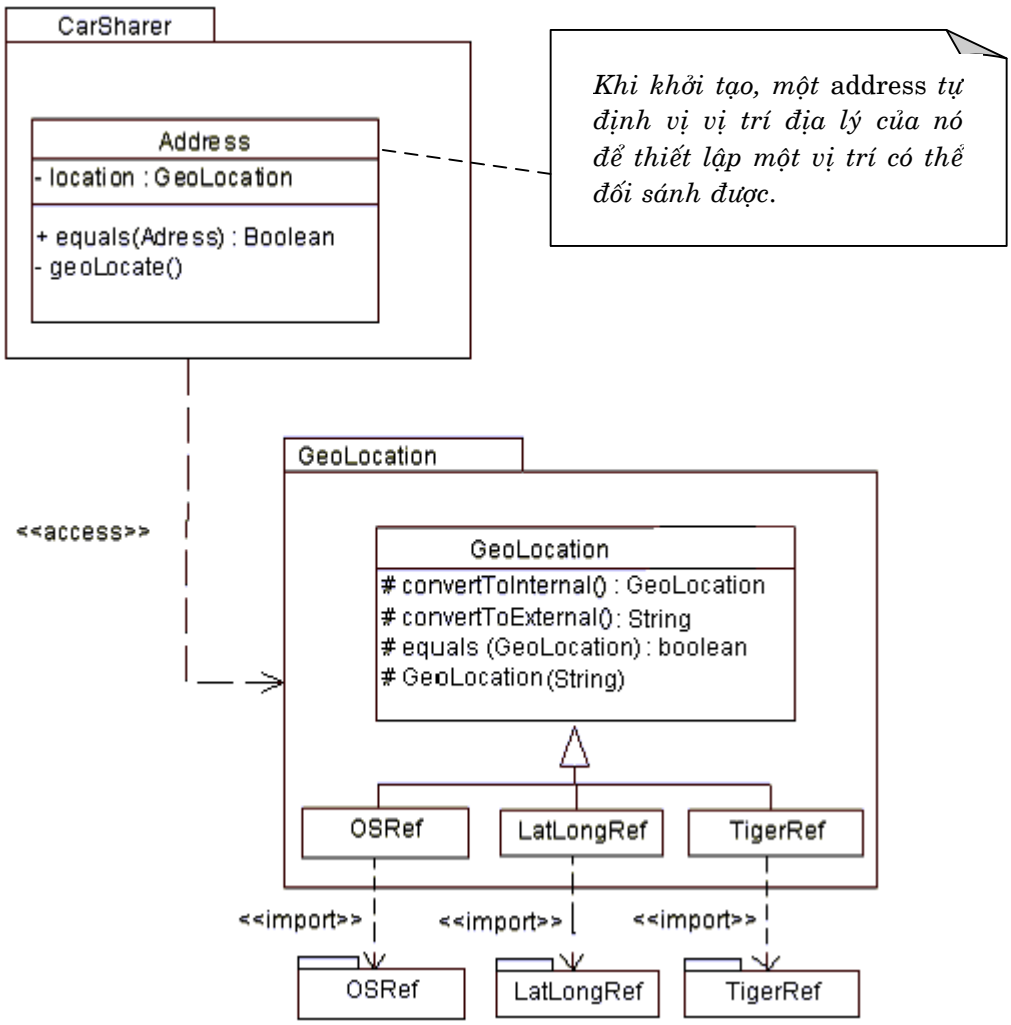
JC: Hẳn là vậy, có lẽ ông muốn cung cấp một thao tác *equals* để thực hiện việc kiểm tra giữa hai vị trí địa lý.

MP: Tốt, tôi sẽ mô hình ngay đây. Tôi có thể mang mô hình đến ông không?

JC: Ồ, không cần đâu. Nhưng nếu ông muốn.

Lời giải:

Mick biết rằng anh ta có thể tạo ra các lớp của riêng mình có thừa kế các chức năng của các lớp có sẵn trong ba thư viện lớp *GIS* ở trên. Các lớp của anh ta sẽ mở rộng các chức năng này để cung cấp các thao tác mà anh ta cần đối sánh một địa chỉ với một địa chỉ khác. Tuy nhiên, Mick cũng nhận ra rằng anh ta cần quản lý sự chuyển đổi giữa định dạng được sử dụng bên trong mỗi hệ thống GIS với định dạng bên ngoài, và ngược lại. Mick đã phác thảo ra một số ý tưởng dựa vào cuộc trao đổi trên như trong hình 5.29.

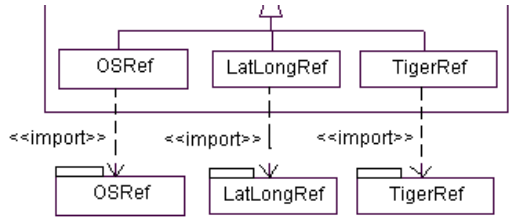


Hình 5.29: Address được mô hình bằng generalization

Biểu đồ này khá phức tạp. Hãy xem xét các khía cạnh khác nhau của nó, có rất nhiều điểm đáng chú ý. Mick đã dùng ký hiệu *quản lý mô hình* biểu diễn lớp, của anh ta, sử dụng các gói khác (hình 5.30). Ba thư viện lớp được *import*. Rốt cuộc ba lớp của anh ta là *GeoLocation::OSRef*, *GeoLocation::LatLongRef* và *GeoLocation::TigerRef* sẽ là các chuyên biệt hoá của các lớp thích hợp, chưa xác định, trong ba gói vừa được *import* này. Mick giả sử rằng các lớp này đã có sẵn dựa trên thông tin Jan đã kể với anh ta.

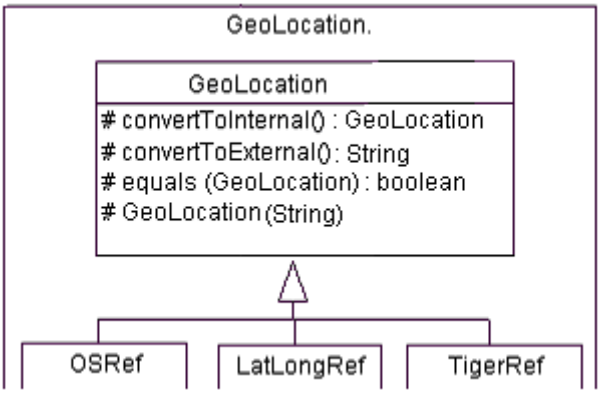
Gói *GeoLocation* của Mick được gói *CarSharer* truy xuất (có quan hệ phụ thuộc <<*access*>>). Điều này cho biết dự định của Mick là tạo ra một kết hợp giữa lớp *Address* và lớp *GeoLocation*. Phụ thuộc *access* cho thấy các lớp trong gói *CarSharer* sẽ có thể truy xuất đến các lớp bên trong gói *GeoLocation*.

Bên trong gói *GeoLocation*, các lớp *OSRef*, *LatLongRef* và *TigerRef* là các chuyên biệt hoá của lớp *GeoLocation*. *GeoLocation* có ba thao tác *protected*. Ba lớp con nói trên sẽ thừa kế các thao tác này.



Hình 5.30: Các package *import*

Ý định của Mick là thao tác *equals* sẽ dùng các thao tác *convertToInternal* và *convertToExternal* của lớp *GeoLocation*. Mỗi lớp con *OSRef*, *LatLongRef* và *TigerRef* sẽ cài đặt các thao tác *convertToInternal* và *convertToExternal* để đóng gói các thuộc tính và các thao tác khác được thừa kế từ một hệ thống GIS cụ thể được chỉ định bởi quan hệ phụ thuộc <<*import*>>.



Hình 5.31: Generalization

Bài tập bổ sung

5.4 Sau đây là một đoạn trích khác từ cuộc phỏng vấn giữa Said Hussain (người phân tích hệ thống) và Martin Page (giám đốc tuyển dụng của *VolBank*).

Said Hussain: Chúng ta có thể quay lại ý tưởng *thời gian được ký gởi* mà anh đã đề cập trước đây không? Hình như anh nói rằng thời gian này được lưu lại cho cả tình nguyện viên và các tổ chức tình nguyện.

Martin Page: Đúng vậy. Chúng tôi muốn kết hợp thời gian ký gởi của tình nguyện viên với thời gian được yêu cầu của cơ hội tình nguyện.

SH: Vậy *thời gian được ký gởi* có tồn tại cho chuyện gì khác hay không?

MP: Không, chỉ cho các tình nguyện viên và các nhu cầu mà thôi. Thực ra, chúng tôi xem xét một công việc tình nguyện theo các yêu cầu thời gian của nó.

SH: Còn tình nguyện viên thì sao?

MP: Tôi nghĩ rằng tình nguyện viên được xem xét theo thời gian tình nguyện mà họ gởi.

SH: ...

Xem lại các kết hợp mà bạn đã phát triển trong bài tập 4.8. Có cần phải bổ sung các ký hiệu và khái niệm aggregation (hoặc composition) không? Bạn có quyết định thêm các kết hợp aggregation (hoặc composition) không? Hãy điều chỉnh biểu đồ cho tốt hơn.

5.5 Đoạn phỏng vấn sau đây liên quan đến tổng quát hoá trong VolBank.

Said Hussain: Liệu chúng ta có thể bàn sang việc ghi nhận chi tiết của tình nguyện viên, thời gian biểu *gởi* và các công việc tình nguyện không? Tôi muốn tìm hiểu việc sắp xếp các tình nguyện viên với các công việc.

Martin Page: Được. Một khi chúng tôi có được một kết hợp, chúng tôi sẽ xếp tình nguyện viên vào dự án, tôi đã đề cập điều này rồi.

SH: Dự án là gì?

MP: Một công việc tình nguyện.

SH: OK, có một nhóm làm việc trên một dự án phải không?

MP: Đúng vậy. Mặc dù chúng tôi cũng có một số người làm việc ở các dự án khác. Trên một số dự án, chúng tôi có một nhóm đủ lớn để cho phép có một hoặc nhiều người hướng dẫn.

SH: Vâng, ông cứ nói tiếp đi.

MP: À, chúng tôi cần biết vai trò của tình nguyện viên trong dự án.

SH: Ông đang nói rằng có ba loại vai trò khác nhau có phải không?

MP: Đúng vậy. Không có lý do gì mà một tình nguyện viên không thể làm việc trên một dự án như một cá nhân riêng lẻ hoặc như một lãnh đạo nhóm, hoặc như một thành viên.

Dựa vào đây, hãy xác định các tổng quát hoá nếu có. Hãy cải tiến biểu đồ lớp của bạn cho phù hợp hơn.

5.6 *Volbank* đang xem xét đưa ra một lược đồ tính điểm. Trong lược đồ này, các tình nguyện viên được xây dựng một *profile* tính điểm. Các tình nguyện viên càng làm việc nhiều thì họ càng lấy được nhiều điểm. Các điểm được bù đắp như một tỉ lệ giảm giá tại các cửa hàng bán lẻ nào đó đang hỗ trợ lược đồ này. Tất cả các sắp xếp cần hỗ trợ cho việc tính toán. Cách tính toán điểm dựa vào các kiểu sắp xếp khác nhau. Thành viên nhóm được tính dựa trên *giờ làm việc* và *mức thành viên*, giống nhau cho tất cả các thành viên nhóm. Với phụ trách nhóm, tính toán dựa trên *giờ làm việc*, *số thành viên* trong nhóm và *mức phụ trách nhóm*; cũng giống nhau cho tất cả các phụ trách nhóm. Cuối cùng, với tình nguyện viên làm việc một mình, điểm được tính trên *giờ làm việc*, *điểm thưởng cá nhân* và *mức cá nhân*.

Sửa đổi cấu trúc tổng quát hoá giữa lớp cha và các lớp con của bạn để thêm các nhiệm vụ được chỉ ra trong đoạn văn trên.

5.7 Hãy xem xét tổng quát hoá mà bạn đã tạo trong bài tập 5.5. Bạn có thể áp dụng bất kỳ các ràng buộc UML đã được định nghĩa trước cho tổng quát hoá này hay không? Hãy giải thích chọn lựa của bạn.

5.8 Cũng với tổng quát hoá bạn đã tạo trong bài tập 5.5, hãy đưa ra một số tên *discriminator* thích hợp cho cấu trúc tổng quát hoá này.

5.9 Với tổng quát hoá bạn đã tạo trong bài tập 5.5, lớp gốc có trừu tượng không? Nếu có, tại sao có? Nếu không, tại sao không? Các tình huống nào ảnh hưởng đến kết luận của bạn?

Chương 6
BIỂU ĐỒ LỚP, TÌM HIỂU THÊM VỀ KẾT HỢP

Các ký hiệu và ngữ nghĩa cơ bản của kết hợp đã được giới thiệu trong chương 4. Có hai loại tên được sử dụng, gồm *tên kết hợp* – chỉ rõ bản chất của kết hợp và *tên vai trò* –mô tả thành phần tham gia vào kết hợp.

Các ví dụ về tên kết hợp như kết hợp *holds* giữa *Customer* và *Account* (hình 4.1 và 5.5), kết hợp *registers* giữa *CarSharer* và *Journey* (hình 4.18). Ví dụ về tên vai trò như vai trò *accountHolder* của *Customer* trong kết hợp *Customer – Account* (hình 4.1 và 5.5).

Multiplicity của kết hợp đã được giới thiệu như là một cách xác định số các đối tượng tham gia trong kết hợp khi được nhìn từ đầu bên kia của kết hợp. Hai multiplicity phổ biến trên các kết hợp là 1 và *, (dạng rút gọn của 1..1 và 0..*).

Chương 5 mở rộng hai ký hiệu kết hợp cơ bản là *aggregation* và *composition*. Chúng được dùng để chỉ rõ bản chất của kết hợp *whole-part* giữa 2 lớp. Kết hợp *composition* là một dạng kết hợp chặt chẽ hơn, liên quan đến chu trình sống. Trong quan hệ composition, khi một lớp *whole* trong kết hợp bị huỷ, thì mọi kết hợp trong lớp *part* cũng bị huỷ theo.

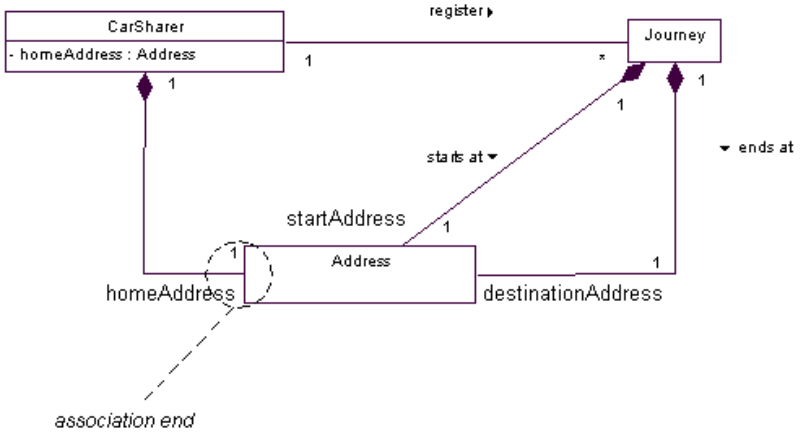
6.1 Giới thiệu

Chương này sẽ thảo luận các ký hiệu còn lại của kết hợp. Các ký hiệu này biểu diễn các ngữ nghĩa chuyên biệt hơn, dùng cho phạm vi lớn hoặc nhỏ hơn trong qui trình lập mô hình. Mọi ký hiệu được trình bày trong chương này đều ít phổ biến hơn các ký hiệu mà chúng ta đã thảo luận trong các chương trước. Tỉ lệ sử dụng thấp hơn là lý do để chúng ta xem xét các ký hiệu này trong một chương riêng.

6.2 Các ký hiệu trên *đầu mút của kết hợp* (association end)

Tất cả các ký hiệu được trình bày trong mục này nhằm thêm ý nghĩa cụ thể cho *đầu mút của kết hợp*, chỗ nối với lớp. Tên vai trò mà chúng ta đã thảo luận trong chương 4 là một ký hiệu như vậy, nó cho biết vai trò mà lớp phải đóng trong kết hợp và được đặt ở đầu mút của kết hợp. Hình 6.1 là một ví dụ.

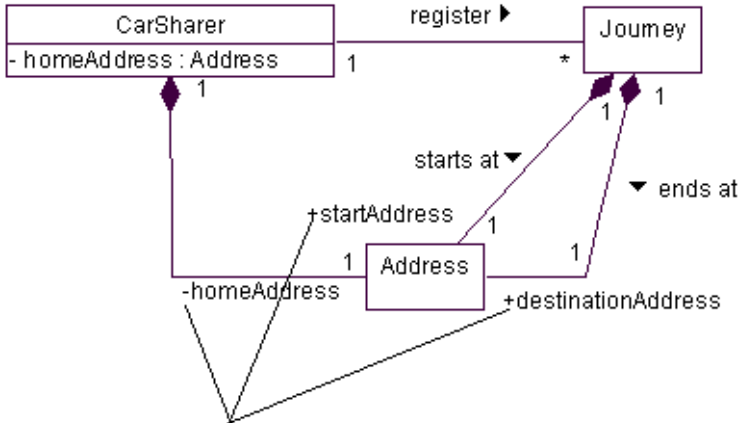
Hình 6-1 mô tả *Address* đóng vai trò *homeAddress* trong kết hợp vô danh với *CarSharer*. Nó cũng đóng vai trò *startAddress* và *destinationAddress* trong các kết hợp *starts at* và *ends at* giữa hai lớp *Journey* và *Address*.



Hình 6.1: Ký hiệu tên vai trò

6.2.1 Visibility (tính khả kiến)

Một *chỉ báo về tính khả kiến* (visibility indicator) có thể được thêm ngay trước *tên vai trò*. *Visibility* đã được giới thiệu trong mục 4.4.3. Nhớ rằng *visibility* được hiển thị bằng một trong ba từ khoá *private* (hoặc dấu –), *public* (+), hay *protected* (#).



chỉ báo về tính khả kiến làm tiền tố trên tên vai trò

Hình 6.2: Visibility của kết hợp được hiển thị cùng với tên vai trò

Ký hiệu *visibility* chỉ được dùng như một tiền tố cho tên vai trò. Nghĩa là chỉ được dùng khi đã có tên vai trò. Hình 6.2 minh họa một ví dụ.

Ví dụ 6.1

Các *visibility indicator* nào được dùng trong hình 6.2?

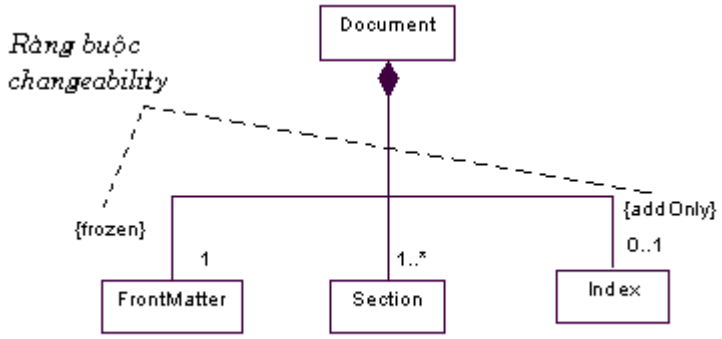
Trả lời:

Lớp *Address* có một *visibility* là *private* được nhìn từ lớp *CarSharer* trong kết hợp *CarSharer-Address* (*-homeAddress*), và có *visibility* là *public* được nhìn từ lớp *Journey* trong cả hai kết hợp *starts at* và *ends at* giữa *Journey* và *Address* (*+startAddress* và *+destinationAddress*).

Indicator đặc tả tính khả kiến của lớp khi duyệt kết hợp hướng đến vai trò lớp. Ví dụ, trong hình 6.2, *homeAddress (Address)* của một *CarSharer* nên được xem là *private*. Chỉ báo này hướng dẫn người thiết kế cài đặt kết hợp. Ví dụ này cài đặt kết hợp giữa *CarSharer* và *homeAddress (Address)* là một thuộc tính *private* của *CarSharer (homeAddress: Address)*.

6.2.2 Changeability (tính khả đổi)

Changeability của kết hợp cho biết có hay không một thể hiện của một lớp, sau khi được khởi tạo, có thể thêm hoặc huỷ các thể hiện của các lớp ở đầu mút ràng buộc *changeability*. Tính khả đổi gồm: *changeable* (có thể thay đổi), *frozen* (không thể thay đổi) hoặc *addOnly* (chỉ được thêm) và được hiển thị như một ràng buộc. Hình 6.3 minh hoạ cho ký hiệu này, dùng ví dụ về xuất bản tài liệu đã trình bày trong mục 5.3.2.



Hình 6.3: Các ràng buộc changeability *của kết hợp*

Nếu không có ràng buộc *changeability* nào được xác định, thì giá trị {*changeable*} được xem là mặc định. Ví dụ, trong hình 6.3 kết hợp *composition* giữa *Document* và *Section* không có ràng buộc *changeability*

và được mặc định là {*changeable*}. Kết hợp này chỉ được ràng buộc bởi các đặc tả *composition* và *multiplicity*.

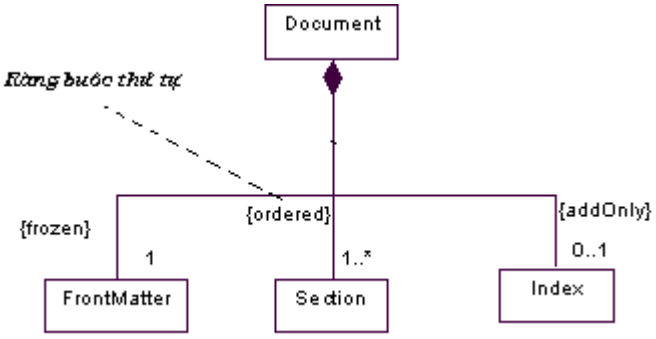
Biểu đồ lớp trong hình 6.3 cũng xác định một thể hiện *Document*, khi được tạo, phải tạo một thành phần *FrontMatter*. Thành phần này không thể bị xoá khi thể hiện của *Document* vẫn còn, ngoài ra không có thành phần *FrontMatter* nào khác được thêm vào. Thoạt tiên, ta có thể nghĩ rằng có sự trùng lắp giữa *multiplicity* 1 và changeability {*frozen*}. *Multiplicity* này cho biết một *Document* phải có một và chỉ một thành phần *FrontMatter*. Tuy nhiên, nếu không có ràng buộc {*frozen*} thì vẫn có quyền thay thể hiện *FrontMatter* bằng một thể hiện khác vì điều này không vi phạm *multiplicity*. Ràng buộc {*frozen*} xác định thể hiện *FrontMatter* là không được thay thế.

Cuối cùng, kết hợp giữa *Document* và *Index* có ràng buộc *changeability* là {*addOnly*}. *Multiplicity* và *composition* qui định một thể hiện *Document* có thể được tạo mà không cần tạo một thể hiện của *Index* đồng thời (do cận dưới *0..* của *multiplicity*). Khi một thể hiện *Document* bị huỷ, thể hiện *Index* cũng huỷ theo (do kết hợp *composition*). Ràng buộc changeability {*addOnly*} xác định một thành phần *Index* chỉ có thể được thêm vào một *Document*. Một khi được thêm, thành phần đó không thể bị gỡ bỏ. Ngoài ra cận trên (..1) của *multiplicity* này còn cho thấy thành phần *Index* sau khi được thêm vào sẽ không bị thay thế bằng thành phần *Index* khác.

6.2.3 Ordering (tính được sắp)

Khi đích của kết hợp có một *multiplicity* với cận trên lớn hơn một, có thể cần xác định tường minh tính được sắp của các thể hiện được kết hợp.

Hình 6.4 cho thấy các *Section* trong một *Document* nên được sắp thứ tự. Nếu không có ràng buộc thứ tự thì mặc định là {*unordered*}.



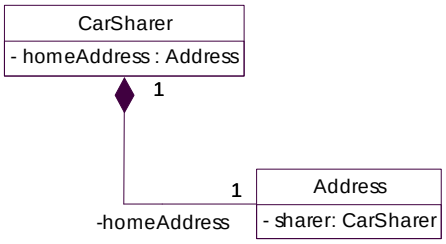
Hình 6.4: Ràng buộc thứ tự của kết hợp

Giống các ràng buộc khác, ràng buộc thứ tự nên được xem là một đặc tả. Cơ chế cài đặt thứ tự để lại cho người thiết kế *các lớp cài đặt*.

Một ràng buộc khác, {*sorted*}, có thể được dùng trong *biểu đồ cài đặt*. Ràng buộc này cho biết việc sắp thứ tự dựa trên trạng thái của các thể hiện. Một lần nữa cơ chế sắp xếp thực sự không được xác định.

6.2.4 Navigability (tính điều hướng)

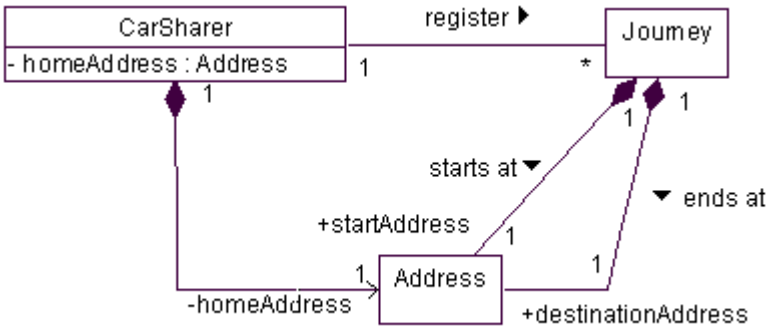
Bình thường các kết hợp được điều hướng theo hai chiều (thực ra đây là nguyên tắc của UML). Điều này có nghĩa là các thông điệp có thể được truyền theo cả hai chiều giữa hai lớp có kết nối với nhau bằng một kết hợp. Ví dụ trong hình 6.5, kết hợp giữa *CarSharer* và *Address* đặc tả một một đường đi để *CarSharer* gởi thông điệp cho *Address* và ngược lại.



Hình 6.5: Navigability của kết hợp không được đặc tả

Sự đơn giản ở đây, xét về mức quan niệm, cung cấp một hiểu biết chung về mô hình lớp. Tuy nhiên, thiết kế và cài đặt một kết hợp hai chiều sinh ra một tầng so với cài đặt của một kết hợp chỉ đi theo một chiều. Tầng này được thảo luận chi tiết trong mục hướng dẫn lập mô hình (mục 6.7.4).

Trong UML, có thể xác định hướng của một kết hợp bằng cách thêm một mũi tên vào kết hợp (hình 6.6).



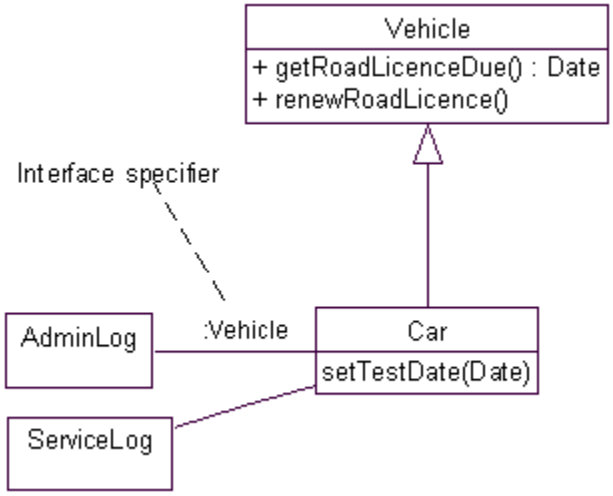
Hình 6.6: Navigability của một kết hợp tường minh

Trong hình 6.6, người phân tích cho ta thấy kết hợp giữa *CarSharer* và *Address* chỉ được phép đi từ *CarSharer* đến *Address*. Ngoài ra không cần cài đặt *navigability* từ *Address* đến *CarSharer*.

6.2.5 Interfaces Specifier *(đặc tả giao diện)*

Thông thường, sự có mặt của một kết hợp ngụ ý rằng tất cả các thao tác và thuộc tính *public* của lớp này đều được lớp kia nhìn thấy (bất kể có hay không có *navigability*).

Điều này không phải luôn luôn đúng. Người phân tích có thể muốn đặc tả một lớp chỉ cần một số thao tác *public* của một lớp khác để thực hiện các yêu cầu cộng tác. Trong UML, điều này có thể đạt được bằng cách dùng một *đặc tả giao diện* (interface specifier). Ký hiệu *interface specifier* được trình bày trong hình 6.7.



Hình 6.7: Interface specifier *trên một kết hợp*

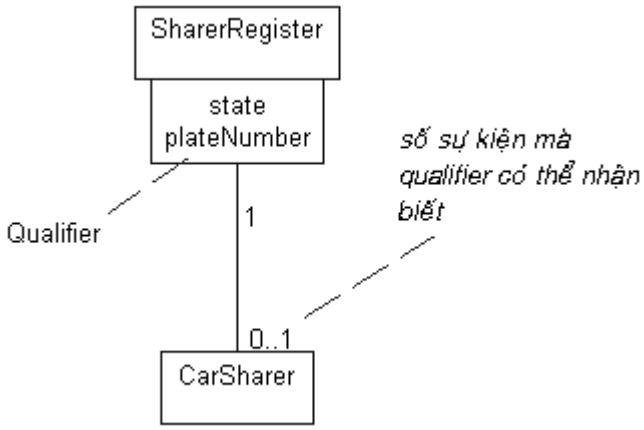
Một *interface specifier* có dạng :<*InterfaceClassName*>. Nó cho biết lớp nguồn (ví dụ *AdminLog*) chỉ yêu cầu một số thao tác *public* của lớp *InterfaceClassName* (ví dụ *Vehicle*), thay vì tất cả các thao tác *public* của lớp đích (ví dụ *Car*).

Có một hàm ý ngữ nghĩa ở lớp định nghĩa *interface specifier*, ví dụ *Car*; nó là một chuyên biệt của một lớp, hoặc là một cài đặt của một lớp giao diện (xem chương 7), hoặc là một số kiểu tinh chế khác. Việc đặt tên cho một *interface specifier* giống với lớp tham gia kết hợp (ví dụ thay :*Vehicle* bởi :*Car*) là một đặc tả thừa.

6.3 Qualifier (định lượng)

Một *qualifier ở phía đầu* của kết hợp đặc tả một phương tiện xác định số lượng *không*, *một* hoặc *nhiều* thể hiện ở *phía cuối* của kết hợp. Ký hiệu *qualifier* được mô tả trong hình 6.8.

Một *qualifier* là một danh sách các thuộc tính, mỗi thuộc tính được liệt kê bằng tên, hoặc dạng *name:type*. *Qualifier* xuất hiện trong một hộp có kích thước nhỏ hơn biểu tượng lớp và được đặt bên cạnh biểu tượng lớp.



Hình 6.8: Ký hiệu của qualifier

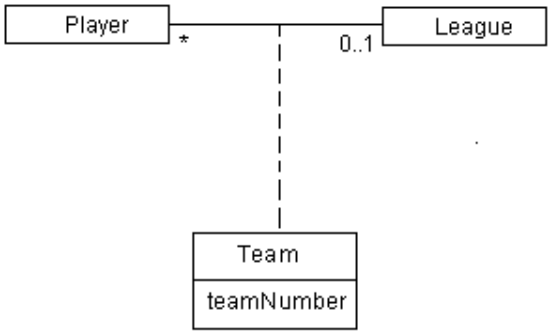
Multiplicity ở phía cuối *qualifier* cho biết số lượng thể hiện mà nó có thể xác định. Trong hình 6.8, *multiplicity* phía *CarSharer* là *0..1* còn *qualifier* gồm hai thuộc tính, *state* (bang) và *plateNumber* (biển số). Nếu biết được giá trị của hai thuộc tính này, thì *qualifier* sẽ xác định có tối đa một *CarSharer*.

Diễn giải cận dưới (*0..*) phụ thuộc ngữ cảnh của mô hình này. Ở đây *state* và *plateNumber* không bị ràng buộc là của xe hơi một thành viên nào đó.

6.4 Lớp kết hợp

Trong mục trước, ta đã biết khái niệm *qualifier* là một thuộc tính của kết hợp. Tuy nhiên đây không phải là trường hợp duy nhất lưu thông tin về kết hợp. Nhu cầu lưu giữ thông tin của một kết hợp giữa hai lớp là không phổ biến. Thực ra, trong một hệ thống hướng đối tượng, có thể cần chuyển giao chức năng (dưới dạng các thao tác) cho một kết hợp.

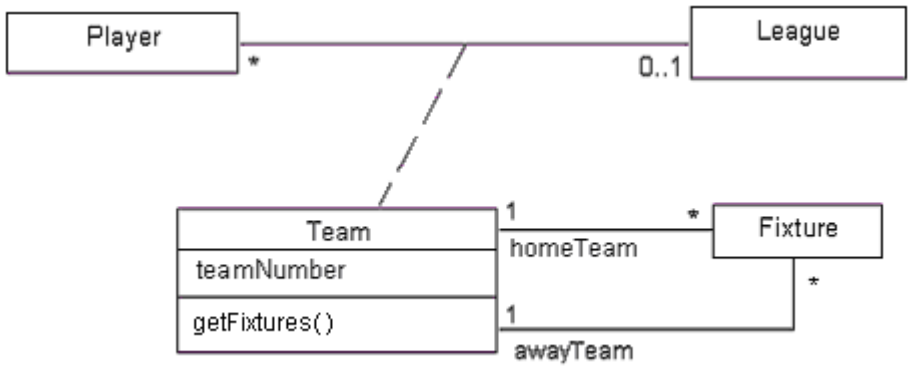
Trong UML, các thuộc tính và thao tác có thể được thêm vào kết hợp bằng cách dùng ký hiệu *lớp kết hợp*. Hình 6.9 biểu diễn ký hiệu của một lớp kết hợp. Lớp kết hợp được biểu diễn như các ký hiệu lớp trong biểu đồ lớp (xem chương 4).



Hình 6.9: Ký hiệu của lớp kết hợp

Một *lớp kết hợp* và *kết hợp* do nó mô tả là hoàn toàn giống nhau về ngữ nghĩa. Tuy nhiên lớp kết hợp và kết hợp được hiển thị riêng biệt nhau. Lớp kết hợp được gắn vào kết hợp mà nó mô tả bằng đường đứt nét.

Do lớp kết hợp giống với mô hình của kết hợp, nên tên của lớp kết hợp và kết hợp như nhau. Tên của kết hợp có thể được hiển thị trên kết hợp hoặc trên lớp (như là một tên lớp hợp lệ).



Hình 6.10: Lớp kết hợp tham gia vào kết hợp khác

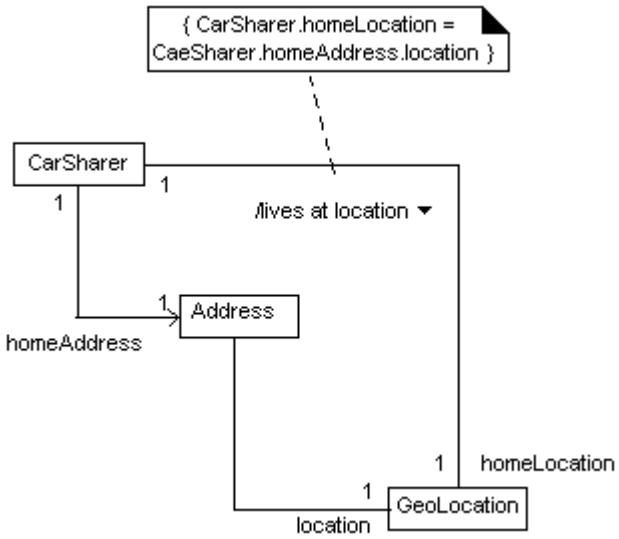
Xét theo ngữ nghĩa bên dưới của UML, một *lớp kết hợp* là một đặc tả của một lớp và một kết hợp. Do là một đặc tả của lớp nên các lớp kết hợp có thể tham gia vào các kiểu quan hệ giống với các lớp khác. Hình 6.10 minh hoạ cho điều này, lớp kết hợp *Team* đóng vai trò *homeTeam* và *awayTeam* trong hai kết hợp với *Fixture*.

6.5 Kết hợp dẫn xuất

Ký hiệu thuộc tính dẫn xuất đã được thảo luận trong chương 4. UML cũng có ký hiệu cho đặc tả của kết hợp dẫn xuất, ký hiệu này được minh họa trong hình 6.11.

Giống với thuộc tính dẫn xuất, tên của kết hợp dẫn xuất được đặt sau ký tự /. Sự dẫn xuất của kết hợp có thể được hiển thị như một ràng buộc trên biểu đồ lớp. Hình 6.11 biểu diễn một kết hợp dẫn xuất có tên */lives at location* giữa *CarSharer* và *GeoLocation*. Đặc tả của sự dẫn xuất này được hiển thị trong một ghi chú, và được gắn vào nhãn của kết hợp.

Đặc tả trong hình 6.11 cho biết vai trò *homeLocation* trong kết hợp *lives at location* được dẫn xuất từ hai kết hợp *CarSharer–Address* và *Address–GeoLocation*. Giống với thuộc tính dẫn xuất, việc dùng các tên vai trò thích hợp có thể có ích trong việc lập công thức cho một đặc tả dẫn xuất.



Hình 6.11: Ký hiệu của kết hợp dẫn xuất

Ví dụ 6.2

Multiplicity của các kết hợp tạo ra kết hợp dẫn xuất ảnh hưởng đến *multiplicity* của kết hợp dẫn xuất như thế nào?

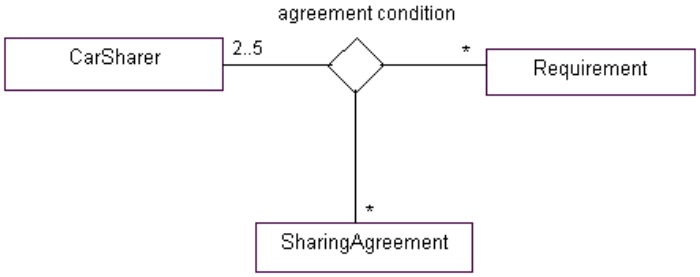
Lời giải: Nhìn từ lớp ở đầu *kết hợp dẫn xuất* đến lớp ở cuối của nó, cận trên và cận dưới của *multiplicity* này sẽ phản ánh các cận cao nhất và thấp nhất xuất hiện qua các bước của dẫn xuất này. Trong ví dụ ở hình 6.11, xét *multiplicity* của kết hợp dẫn xuất ở đầu *homeLocation* ta có *multiplicity* thấp nhất trong cả hai kết hợp *CarSharer–Address* và *Address–GeoLocation* là *1*... cũng vậy, *multiplicity* cao nhất là *..1*. Vì vậy, *multiplicity* ở đầu *homeLocation* của kết hợp *lives at location* là *1..1* (hoặc *1*). Multiplicity tương tự được dẫn xuất khi duyệt qua kết hợp theo hướng từ *GeoLocation* đến *CarSharer*.

Nếu các kết hợp thành phần có một kết hợp với cận trên là ..* và một kết hợp với cận dưới là *0..* thì *multiplicity* của kết hợp dẫn xuất sẽ là *0..** (hoặc *).

6.6 Kết hợp n-ary (kết hợp n-ngôi)

Các kết hợp mà chúng ta đã khảo sát từ trước đến nay là các kết hợp hai ngôi, liên kết hai lớp lại với nhau. UML cũng cung cấp ký hiệu đối với kết hợp n-ngôi liên kết từ ba lớp trở lên với nhau.

Ký hiệu kết hợp n-ngôi (n-ary association) là một biểu tượng hình thoi được nối với các lớp tham gia vào kết hợp. Ký hiệu này được minh hoạ trong hình 6.12.



Hình 6.12: Ký hiệu của một kết hợp n-ngôi

Hình 6.12 có một *kết hợp n-ngôi* tên là *agreement condition*. Trong ví dụ này, các yêu cầu được biểu diễn thành một lớp độc lập *Requirement*, bởi các yêu cầu này tồn tại độc lập với *CarSharers*. Một thể hiện của kết hợp *agreement condition* liên quan với các thể hiện của *CarSharer*, *Requirement* và *SharingAgreement*.

Với kết hợp *n-ngôi*, khó thiết lập và giải thích *multiplicity* hơn so với kết hợp hai ngôi. Để giải thích *multiplicity* của kết hợp *n-ngôi*, ta xét thể hiện của kết hợp liên quan đến tất cả các lớp trừ ra một lớp rồi xác định số thể hiện tham gia vào kết hợp của lớp còn lại này.

Trong hình 6.12, nếu thể hiện của kết hợp *agreement condition* liên quan đến *CarSharer* và *SharingAgreement*, thì số thể hiện của *Requirement* tham gia trong kết hợp này nằm giữa *không* và *nhiều* (là *multiplicity* * của lớp *Requirement*). Nói cách khác, một kết hợp giữa *CarSharers*, *SharingAgreements* và *Requirement* có thể có vài yêu cầu (giữa không và nhiều) đối với một người chia sẻ xe và một hợp đồng chia sẻ.

Ví dụ 6.3

Giải thích *multiplicity* của *CarSharer* và *SharingAgreement* (hình 6.12).

Lời giải: Với một *SharingAgreement* và một *Requirement*, một thể hiện của kết hợp liên quan với từ 2 đến 5 *CarSharer*. Với một *CarSharer* và một *Requirement*, một thể hiện của kết hợp sẽ liên quan 0 hoặc nhiều *SharingAgreement*.

Tên vai trò rất có ích trong kết hợp n-ngôi bởi nó làm sáng tỏ bản chất các lớp tham gia vào kết hợp. Tuy nhiên, đặc tả UML không cho phép kết hợp n-ngôi dùng *qualifier*, *aggregation* hoặc *composition* (OMG, 1999b, trang 3-74). Việc dùng các chú thích như *navigability*, *odering*, *changeability* và *visibility* là không bị cấm nhưng có thể bị hiểu sai.

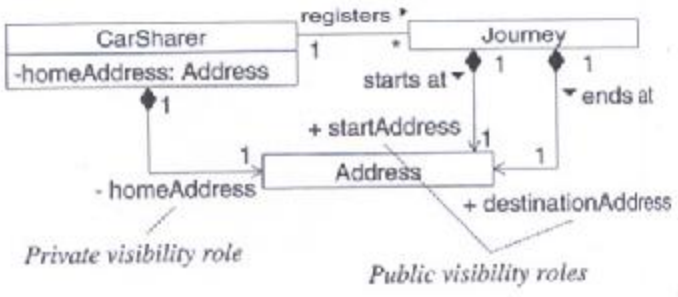
6.7 Hướng dẫn lập mô hình

Chương này gồm các ký hiệu đặc biệt được dùng trong biểu đồ lớp. Trong ngữ cảnh như vậy, một số hướng dẫn đặt trọng tâm vào việc dùng hoặc không dùng chúng. Điều này đặc biệt đúng trong mục sau đây.

6.7.1 Các đầu mút của kết hợp

Tính khả kiến (*visibility*)

Visibility ở đầu mút một kết hợp được xét tương tự như *visibility* của thuộc tính lớp. Trong các hệ thống hướng đối tượng, *visibility* rất cần để hạn chế truy cập đến cấu trúc bên trong của lớp. Hãy xét ý nghĩa của việc tạo ra một đầu kết hợp public. Hình 6.13 trình bày một biểu đồ lớp minh hoạ.



Hình 6.13: Visibility ở đầu mút của kết hợp

Hai vai trò *startAddress* và *destinationAddress* là public. Xét trường hợp có một yêu cầu thêm thao tác *printStartAddresses* vào lớp *CarSharer*. Thao tác này trả về các địa chỉ xuất phát của tất cả các lộ trình được đăng ký bởi một *CarSharer*.

Do *startAddress* là public, nên *CarSharer* có thể truy xuất trực tiếp đến thể hiện của *Address*, tức là *startAddress* của một *Journey*. Đặc tả của thao tác này có mã tương tự như sau:

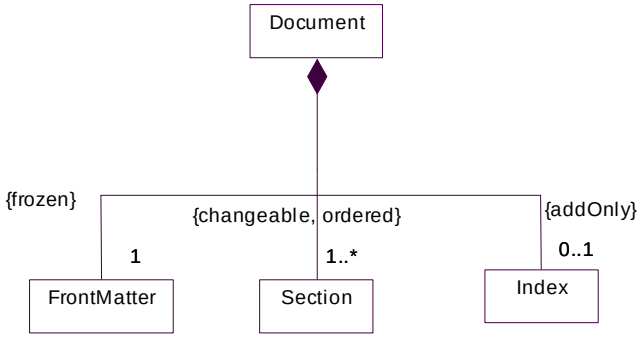
```
forAll(Journey) {
        Journey.startAddress.print();
}
```

Nói cách khác, với tất cả các lộ trình đã được đăng ký bởi một *CarSharer*, thì thao tác *printStartAddresses* sẽ gọi thao tác *print*() trên thể hiện của *Address* đang đóng vai trò *startAddress* đối với *Journey*.

Thoạt nhìn tiếp cận này có vẻ tốt. Còn một tiếp cận thứ hai là truyền thông điệp *printStartAddress* cho mỗi *Journey*, rồi đến lượt *Journey* gởi thông điệp *print*() cho *startAddress* của nó. Hướng tiếp cận thứ hai này gồm hai thông điệp trong khi đó tiếp cận đầu gọi *Address.print*() trực tiếp từ *CarSharer*.

Cách gọi trực tiếp có một hạn chế. Chuyện gì xảy ra nếu giao diện (tập các thao tác public) trên *Address* thay đổi, hoặc phương thức của thao tác *print*() thay đổi? Trong tình huống này, việc để các lời gọi trực tiếp đến thao tác *Address* trong các lớp không được kết hợp trực tiếp với *Address* là một điều khó khăn. Việc theo vết các lớp này và đảm bảo rằng tất cả các tương tác có thể có được sữa mã lại cho phù hợp và được kiểm tra lại trở nên quá tầm. Thay vào đó, tốt nhất là giữ cho *visibility* là *private*.

6.7.2 Changeability *và* Ordering



Hình 6.14: Cấu trúc của Document *được kiểm soát*

Ràng buộc *chageability* có ích trong trường hợp một cấu trúc lớp phải tuân theo theo mẫu đã được định nghĩa trước. Xem ví dụ minh họa trong hình 6.14.

Ví dụ 6.4

Giải thích các kết hợp *Document–FrontMatter*, *Document–Index* và các ràng buộc.

Lời giải:

Với *Document–FrontMatter*, một *document* phải có một và chỉ một thành phần *frontMatter*. Thành phần này không được thay thế ({*frozen*}). Với *Document–Index*, một *document* không cần phải có một *index* khi nó được tạo ra. Một khi một *index* được thêm vào thì nó không thể được thay thế. Một *document* không thể có hơn một *index*.

Ràng buộc {*Changeable*} trên *Document–Section*, với multiplicity *1..** có nghĩa là:

- Một *document* phải có một *section* một khi nó được tạo ra.
- Các *section* có thể được thêm vào, bị xoá đi hoặc bị thay thế (ràng buộc *changeability*) nhưng *document* phải luôn có ít nhất một *section*.

Ràng buộc {*ordered*} có ích trong trường hợp có một thứ tự cố hữu hoặc một thứ tự tường minh trên các thể hiện của lớp. Trong hình 6.14, ràng buộc thứ tự trên kết hợp *Document–Section* cho biết các *section* tạo ra một *document* có một thứ tự cố hữu. Bản chất của việc sắp thứ tự này không được chỉ ra. Ví dụ, nó có thể liên quan đến cấu trúc bên trong của *document* trong một môi trường soạn thảo, hoặc liên quan đến một số *section* được cấp phát cho mỗi một *section*.

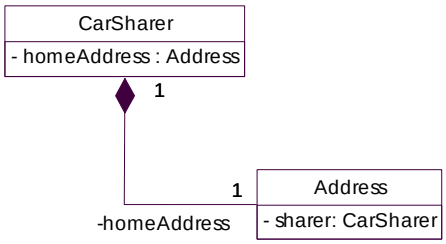
6.7.3 Các dẫn xuất

Các nguyên tắc của *ràng buộc dẫn xuất* đã được thảo luận trong ngữ cảnh của thuộc tính dẫn xuất (mục 4.4.3.3). Kết hợp với ví dụ đã thảo luận trong phần 6.5, ở đây không có gì để thảo luận thêm.

6.7.4 Navigability

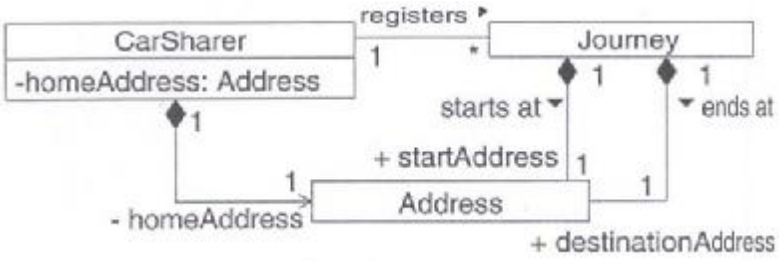
Hình 6.15 cho thấy một cách cài đặt của một kết hợp hai chiều giữa *CarSharer* và *Address*, được cài đặt thành hai thuộc tính trên hai lớp. Thuộc tính *homeAddress* trên *CarSharer* cho phép nhận ra thể hiện *Address* kết hợp với *CarSharer*. Thuộc tính *sharer* trên *Address* cho phép xác định thể hiện *CarSharer* kết hợp với *Address*. Như vậy hai thuộc tính phải được cài đặt cho một kết hợp. Thực ra, mỗi hướng cần

một thuộc tính. (Lưu ý rằng các thiết kế khác nhau có thể được dùng để cài đặt cho cùng một kết hợp).



Hình 6.15: Navigability *của kết hợp vô danh*

Giống với *tầng thiết kế* này, có một *tầng thi hành* để hỗ trợ các kết hợp hai chiều. Khi một thể hiện *Address* được tạo ra, như *homeAddress*, nó cần được hướng dẫn để trỏ trở lại thể hiện *CarSharer* đã tạo ra nó. Nếu địa chỉ nhà của một *CarSharer* thay đổi (một thể hiện của *Address* bị thay thế bởi một thể hiện khác), thì *Address* mới phải được kết hợp với *CarSharer* và thể hiện *Address* mới sẽ phải được hướng dẫn để trỏ ngược trở lại *CarSharer* một lần nữa. Việc quản lý kết hợp này sẽ trở nên phức tạp hơn nếu như *multiplicity* không phải là *1..1*.



Hình 6.16: Navigability của kết hợp tường minh

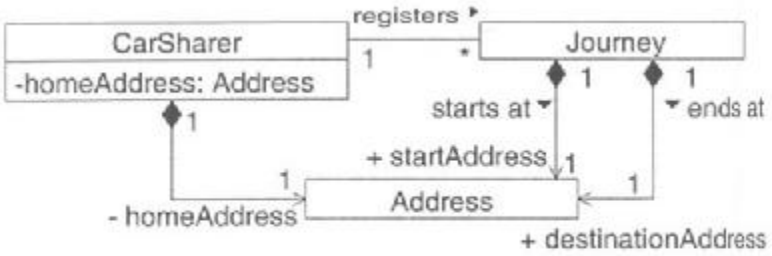
Việc xác định *navigability* của một kết hợp có thể làm đơn giản qui trình thiết kế và cài đặt. Ví dụ, nếu kết hợp giữa *CarSharer* và *Address* được xác định là chỉ có thể di chuyển từ *CarSharer* sang *Address* thì không cần cài đặt thuộc tính *sharer*:*CarSharer* trên *Address* (hình 6.16).

Ví dụ 6.5

Hãy cho biết các kết hợp khác trên biểu đồ lớp của hình 6.16 mà bạn nghĩ có thể là kết hợp một chiều.

Lời giải:

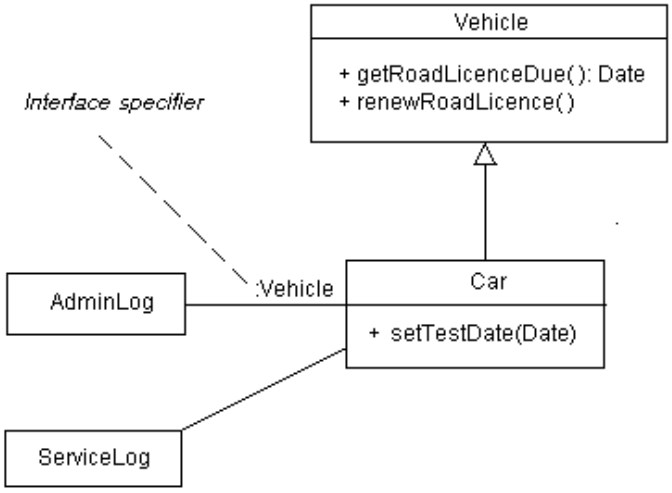
Hình 6.17 trình bày một giải pháp.



Hình 6.17: Các kết hợp một chiều khác

Có thể giả định rằng kết hợp *starts at* và *ends at* giữa *Journey* và *Address* chỉ luôn có duy nhất một hướng di chuyển là từ *Journey* sang *Address*.

6.7.5 Interface Specifier



Hình 6.18: Interface specifier *trên một kết hợp*

Interface specifier có thể cung cấp thông tin có ích trên một biểu đồ lớp bởi vì nó giúp xác định cách cư xử của một lớp đối với một lớp khác.

Ví dụ, hãy khảo sát kết hợp giữa *ServiceLog* và *Car* trong biểu đồ lớp được tạo trong hình 6.18 (theo mục 6.2.5). Mô hình lớp chỉ ra rằng lớp *ServiceLog* mong muốn truy xuất bất kỳ thao tác nào trên lớp *Car*. Đặc

biệt, lớp *ServiceLog* có thể truy xuất các thao tác *setTestDate, getRoadLicenceDue,* hoặc *renewRoadLicence*.

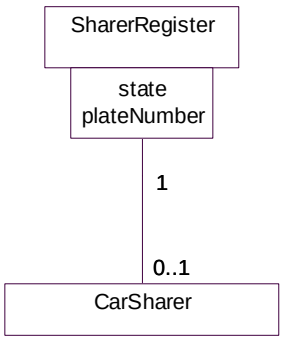
Trong kết hợp giữa *AdminLog* và *Car*, interface specifier đặc tả là :*Vehicle* ngụ ý lớp *AdminLog* chỉ yêu cầu truy xuất đến các thao tác public của lớp *Vehicle* khi nó cộng tác với lớp *Car*. Nói cách khác, *AdminLog* muốn lớp *Car* cung cấp các thao tác *getRoadLicenceDue* và *renewRoadLience* dưới dạng public.

Có hai điểm cần lưu ý ở đây. Trước nhất, *AdminLog* không mong muốn sử dụng tất cả các thao tác public của *Car*. Nó chỉ muốn sử dụng các thao tác được chỉ định trong lớp *Vehicle*. Thứ hai, việc sử dụng interface specifier ngụ ý rằng *Car* phải cung cấp các thao tác public giống với *Vehicle*. Không thể chấp nhận việc *Car* không cung cấp một thao tác nào đó, chẳng hạn như *getRoadLicenceDue*, bằng cách ghi đè lên visibility của thao tác đó làm cho nó trở thành private.

6.7.6 Qualifier

Qualifier được dùng như là một phương tiện tìm tập các đối tượng kết hợp với một lớp bằng cách dùng các giá trị thuộc tính thay vì dùng các định danh cụ thể. Có ba biến đổi phổ biến khi dùng qualifier.

Hình 6.19 trình bày một kết hợp giữa một *SharerRegister* và một *CarSharer*. *SharerRegister* là danh sách *CarSharer* đã đăng ký của CarMatch. Qualifier bao gồm hai thuộc tính, *state* và *plateNumber*.

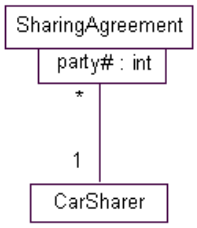


Hình 6.19: Một qualifier *với hai thuộc tính*

Qualifier này và multiplicity *0..1* ở đầu *CarSharer* nói rằng nó có thể không xác định hoặc xác định một *CarSharer*. Cận dưới (*0..*) cho biết một *state* và một *plateNumber* không thể nhận ra một *CarSharer*. Cận trên *(..1*) cho biết tối đa một *CarSharer* được nhận biết.

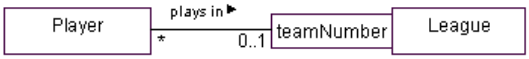
Biểu đồ lớp trong hình 6.20 cho biết các thành viên trong bản thoả thuận được xác định bởi mã số thành viên của họ (*party#*). Trong ví dụ này, *party#* có kiểu *int*.

Cận dưới của multiplicity ở đầu kết hợp là *1*, nghĩa là một *party#* luôn xác định ít nhất một *CarSharer*. Tương tự, nếu cận trên cũng là *1* thì có nghĩa là không thể có nhiều hơn một *CarSharer* sẽ được xác định bởi một *party#*. Tóm lại, kết hợp đặc tả một *party#* luôn xác định một và chỉ một *CarSharer* trong một *SharingAgreement*.



Hình 6.20: Unique qualifier

Cuối cùng, hình 6.21 minh họa ví dụ một *qualifier* được dùng để xác định một nhóm các thể hiện. Trong trường hợp này, một *teamNumber* (mã số của đội) được dùng để xác định *không*, *một* hoặc *nhiều* vận động viên.



Hình 6.21: qualifier *phân nhóm*

Các thuộc tính tạo ra *qualifier* của kết hợp không phải là các thuộc tính của các lớp ở hai đầu kết hợp. Ví dụ, trong hình 6.21 *teamNumber* không phải là một thuộc tính của *League* lẫn *Player*.

Trực giác cho thấy *teamNumber* không thể là thuộc tính của *League*. Nếu là thuộc tính của *League*, nó sẽ giữ nhiều giá trị cho tất cả các đội trong một *liên đoàn* (League); và để tạo thể hiện cho liên kết giữa liên đoàn và vận động viên, cần tạo ra một số loại liên kết đến các đội hoặc đến các vận động viên trong các đội này.

Trường hợp *teamNumber* không thuộc lớp *Player* thì khó hình dung hơn. Tuy nhiên, multiplicity của *plays* cho phép có vận động viên không chơi trong liên đoàn. Trong trường hợp này, một thuộc tính *teamNumber* trên *plays* là không thích hợp.

Thực tế giá trị *teamNumber* là dữ liệu về một vận động viên chơi trong một liên đoàn. Nói cách khác, nó là thuộc tính của chính kết hợp. Nếu một vận động viên không chơi trong một liên đoàn nào thì sẽ không có

thể hiện của kết hợp liên quan, do đó cũng không có giá trị *teamNumber*. Nếu một vận động viên đổi liên đoàn thì ngữ cảnh của kết hợp bị đổi, cho nên các thuộc tính bị thay đổi phải là thuộc tính của kết hợp.

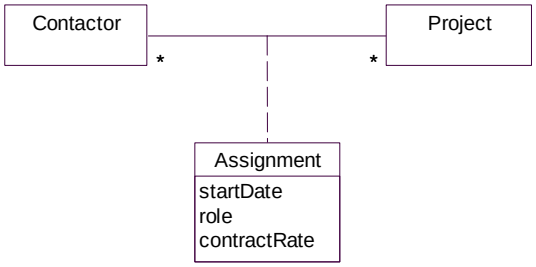
6.7.7 Lớp kết hợp

Lớp kết hợp được dùng khi kết hợp có thuộc tính hoặc thao tác và cần được biểu diễn trong mô hình lớp. Tình huống sử dụng chung nhất là để giữ lại đặc trưng lịch sử hoặc phụ thuộc thời gian của một quan hệ. Bảng 6.1 trình bày một số ví dụ về lớp kết hợp.

Kết hợp	Lớp kết hợp	Thuộc tính
Contractor–Project	*Assignment*	*startDate, role, contractRate*
CarSharer–RequirementType	*Requirement*	*priority, comment, fromDate, untilDate*

Bảng 6.1: Ví dụ về lớp kết hợp lịch sử hoặc phụ thuộc thời gian

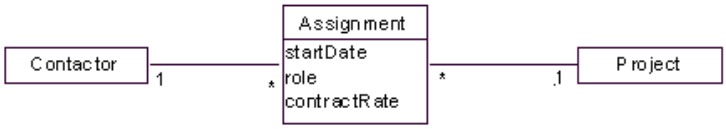
Do kết hợp và lớp kết hợp là một và cùng khái niệm, nên chỉ có một xuất hiện của một lớp kết hợp đối với mỗi xuất hiện của một kết hợp. Ví dụ, hãy xem xét kết hợp *Contract–Project* và lớp kết hợp *Assignment* được phác thảo trong bảng 6.1. Kết hợp này được biểu diễn trong hình 6.22.



Hình 6.22: Lớp kết hợp Assignment *giữa* Contractor *và* Project

Mỗi lần gán một *contractor* (người thầu) cho một *project* (dự án) sẽ đưa đến một xuất hiện của kết hợp này và tập các giá trị cho *startDate*, *role* và *contractRate*. Khi một *contractor* làm việc cho các dự án khác nhau, thì ba thuộc tính này sẽ giữ các giá trị khác nhau cho các kết hợp khác nhau đó. Khi một *contractor* được gán cho cùng một dự án (trùng với lần trước) thì vẫn chỉ có một lần xuất hiện của kết hợp. Như vậy, *startDate*, *role* và *contractRate* chỉ giữ các giá trị mới nhất đối với việc gán một *contractor* cho một *project*, nó không giữ tất cả các giá trị trước đó.

Mô hình *lớp kết hợp* như lớp bình thường cần đến ràng buộc '*một lớp-kết-hợp cho mỗi kết hợp*'. Biểu đồ lớp trong hình 6.23 không tương đương với hình 6.22 bởi vì nó có khả năng tạo ra các xuất hiện khác của lớp *Assignment* trên cùng các xuất hiện của *Contractor* và *Project*.



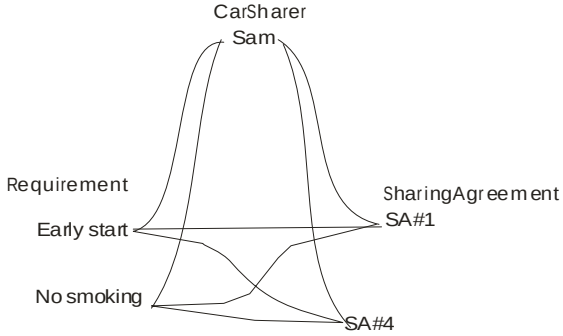
Hình 6.23: Assignment *được mô hình như một lớp*

Như vậy một lớp kết hợp có thể được dùng khi có ràng buộc '*một xuất hiện của kết hợp trên mỗi thể hiện của lớp kết hợp*'. Nếu là trường hợp khác, ví dụ như quá khứ của các lần gán một *contractor* tới một *project* được yêu cầu, thì lớp kết hợp nên được mô hình như một lớp bình thường với các kết hợp thích hợp với các lớp khác.

6.7.8 Kết hợp n-ngôi

Mục 6.6 đã giới thiệu ký hiệu kết hợp n-ngôi. Một khảo sát sẽ minh họa nơi nào ký hiệu này là có ích.

Xét trường hợp người phân tích bắt đầu mô hình các kết hợp giữa *CarSharer*, *Requirement* và *SharingAgreement*. Hình 6.24 minh họa một số xuất hiện có thể của các kết hợp cần mô hình.



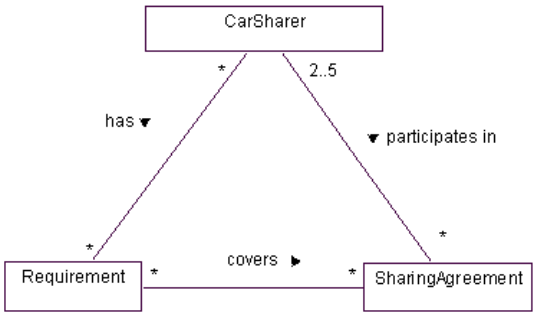
Hình 6.24: Các xuất hiện của ba cách kết hợp

Trong hình, *Sam*:*CarSharer* có 2 yêu cầu *Early start*: *Requirement* và *No smoking*: *Requirement*. *Sam* tham gia vào *SA#1: SharingAgreement* và *SA#4: SharingAgreement*. Cả hai bản thoả thuận này đều bao gồm hai yêu cầu của *Sam*.

Hình 6.25 trình bày một mô hình ban đầu của các thể hiện này. Người phân tích đã mô hình 3 *kết hợp hai ngôi* giữa ba lớp. Các kết hợp này phù hợp với các xuất hiện trong hình 6.24. Tuy nhiên, nếu như có thêm nhiều xuất hiện minh họa thì bài toán với cấu trúc kết hợp hai ngôi này sẽ trở nên rõ ràng hơn.

Hình 6.26 trình bày các thể hiện bổ sung của kết hợp giữa ba lớp. *Lou: CarSharer* với hai yêu cầu *No smoking:Requirement* và *Allergen filter:*

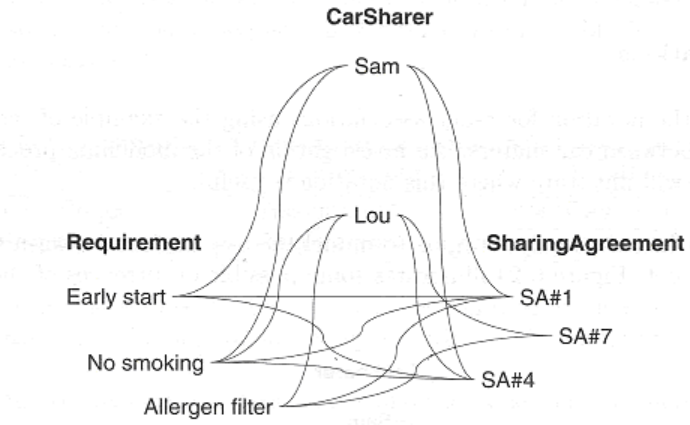
Requirerment. *Lou* tham gia vào *SA#4 : SharingAgreement* cùng với *SAM* và cũng tham gia vào *SA#7: SharingAgreement*. Cả hai thoả thuận này bao gồm đủ các yêu cầu của *Lou*.



Hình 6.25: Mô hình các kết hợp của hình 6.24

Bài toán mô hình các thể hiện này như ba kết hợp 2-ngôi không thể giữ được tính toàn vẹn ban đầu của các thể hiện gốc khi điều hướng ngang qua các kết hợp.

Hãy xem chuyện gì sẽ xảy ra khi di chuyển từ *CarSharer* rồi vòng theo ba kết hợp hai ngôi. Nếu tính toàn vẹn của các thể hiện được duy trì thì hai điều sau phải đúng. Thứ nhất, có thể di chuyển qua cả ba kết hợp và quay trở lại điểm bắt đầu. Thứ hai, không thể di chuyển đến một thể hiện lớp hoặc thể hiện kết hợp không được bao gồm bởi các thể hiện kết hợp gốc. Hình 6.27 cho thấy hầu hết thể hiện được biểu diễn trong hình 6.26.



Hình 6.26: Các thể hiện kết hợp bổ sung

Ngoài lề:

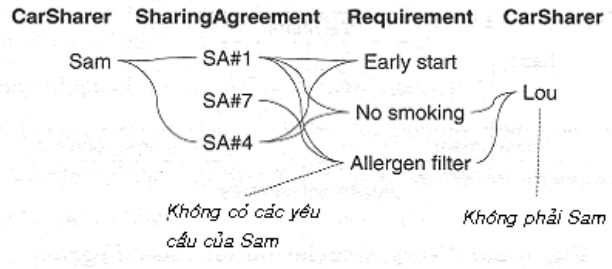
Để làm biểu đồ không bị rối rắm, các thể hiện kết hợp sau đây nên được xoá ra khỏi hình 6.27.

- *Thể hiện liên kết SAM với Early start.*
- *Thể hiện liên kết SAM với No smoking.*
- *Thể hiện liên kết Lou với SA#7.*
- *Thể hiện liên kết Lou với SA#4.*

Xoá các thể hiện này làm cho biểu đồ trong hình 6.27 dễ đọc. Các thể hiện đã bị xóa không ảnh hưởng đến các giải thích dưới đây.

Bắt đầu từ *Sam*, đi đến các bản thoả thuận có *Sam* trong đó (*SA#1*, *SA#4*). Từ đây đi đến các yêu cầu phải thoả mãn. Xuất hiện vấn đề thứ nhất, khi từ *SA#1* đi đến yêu cầu *Allergen filler* không phải là một yêu cầu của *Sam*. Cũng vậy nếu bắt đầu từ *SA#1* đến *No smoking* rồi *Lou* và không thể về *SA#1*.

Như vậy từ *Sam*:*CarSharer*, đi theo các thể hiện ta có thể lấy được một yêu cầu không phải của *Sam*, mà của *Lou*. Sự mất toàn vẹn như vậy xảy ra bất chấp điểm bắt đầu.



Hình 6.27: Lần theo các liên kết từ hình 6.26

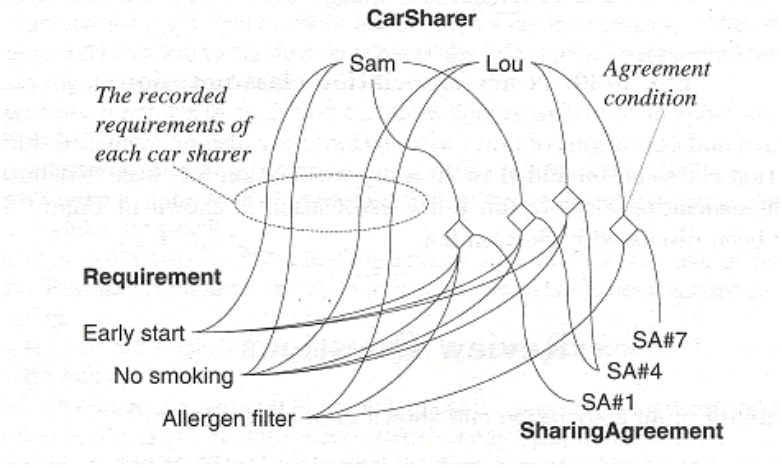
Ví dụ 6.6

Di chuyển vòng quanh các kết hợp hai ngôi trong hình 6.26, lần lượt bắt đầu từ *SA#4*:*SharingAgreement* và *Allergen filler*:*Requirement*. Bạn gặp phải vấn đề toàn vẹn nào?

Lời giải:

- *SA#4* – *Lou* – *Allergen filler* (không có trong *SA#4*) – rồi *SA#1* hoặc *SA#7*
- *Allergen filler* – *SA#1*– *Sam* (không yêu cầu *Allergen filler*) – rồi *Early start* hoặc *No smoking*

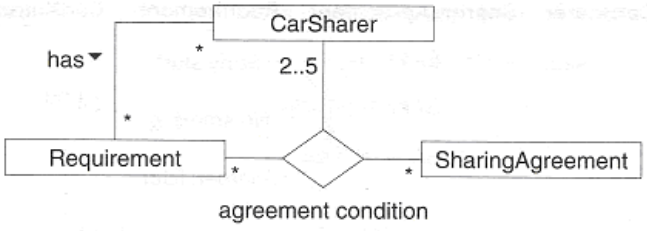
Một giải thích cho điều này liên quan đến vấn đề khái niệm. Mỗi kết hợp hai ngôi là hợp lệ. Chỉ khi ba kết hợp như vậy cùng giải thích một liên kết xảy ra giữa ba lớp mới làm phát sinh vấn đề toàn vẹn.



Hình 6.28: Các thể hiện của kết hợp n-ngôi

Nếu có một yêu cầu quan niệm liên kết ba thể hiện của ba lớp lại thì một cấu trúc kết hợp cho phép điều này là cần thiết. Hình 6-28 minh họa điều này, cùng với các liên kết hai ngôi giữa người dùng chung xe và các yêu cầu.

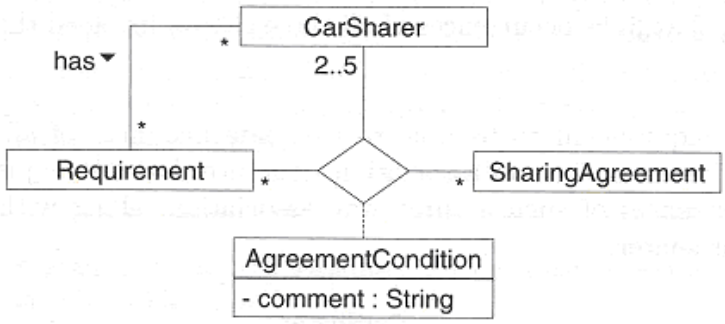
Hình 6.28 ẩn chứa một kiểu biểu diễn các thể hiện của kết hợp n-ngôi. Mỗi thể hiện này liên kết ba thể hiện của *CarSharer*, *Requirement* và *SharingAgreement* lại. Hình 6-29 mô hình thành một biểu đồ lớp.



Hình 6.29: Kết hợp n-ngôi trong biểu đồ lớp

Kết hợp *has* giữa *CarSharer* và *Requirement* cho phép mỗi *CarSharer* nắm giữ các *Requirement*. Các yêu cầu này chưa cần thoả mãn, chỉ hướng dẫn việc lập bản thoả thuận bảo đảm chỉ những thành viên có cùng yêu cầu được xếp cùng nhau.

Với mục đích rõ ràng, cần giới hạn số thể hiện. Mặt dù có thể dẫn ta đến kết luận là tất cả các yêu cầu của một thành viên đều được gặp trong bất kỳ bản thoả thuận nào mà anh ta tham gia. Thật ra mỗi thể hiện của kết hợp n-ngôi này sẽ chỉ gặp một vài chứ không phải gặp tất cả các yêu cầu của thành viên trong kết hợp. Kết hợp *agreement condition* liên kết các thành viên và các yêu cầu của họ vào bản thoả thuận.



Hình 6.30: Ký hiệu lớp kết hợp n-ngôi

Nhu cầu có các thuộc tính và các thao tác cho kết hợp n-ngôi có thể được xác định trong suốt qui trình mô hình hoá. Khi ấy một lớp kết hợp được thêm vào kết hợp n-ngôi dành cho các thuộc tính và các thao tác này. Ký hiệu lớp kết hợp cho kết hợp n-ngôi được minh họa trong hình 6.30.

Câu hỏi ôn tập

6.1 *Visibility* ở đầu mút của kết hợp được biểu diễn như thế nào?

6.2 Ba ràng buộc *changeability* nào được định nghĩa trong UML? Cho biết ý nghĩa của từng ràng buộc?

6.3 Ký hiệu gì đặc tả các đối tượng có liên quan trong một kết hợp nên có tính thứ tự?

6.4 *Navigability* của kết hợp được hiển thị như thế nào trong UML? Trong hoàn cảnh nào cần đặc tả tường minh một kết hợp có *navigability* hai chiều?

6.5 Việc thêm *interface specifier* vào một đầu kết hợp có ý nghĩa gì? Ký hiệu của *interface specifier* là gì?

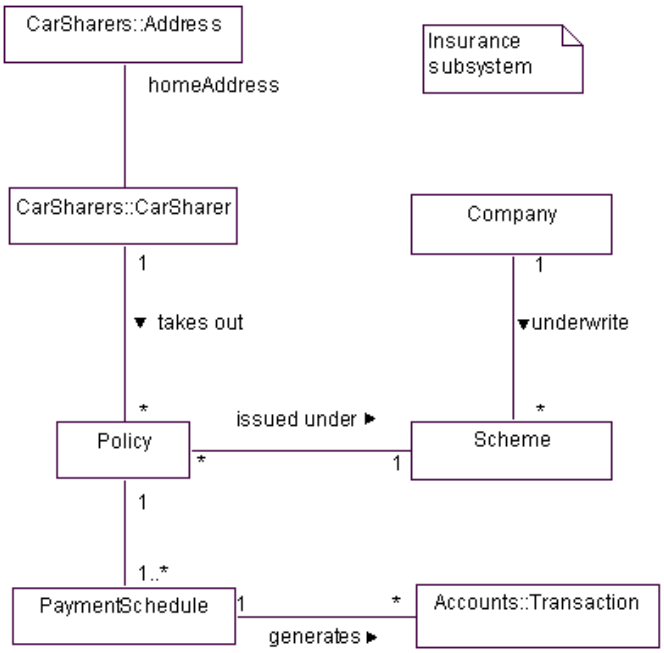
6.6 Hãy thảo luận ý nghĩa của *qualifier* khi được dùng trên biểu đồ lớp.

6.7 Thuộc tính dẫn xuất và kết hợp dẫn xuất có chung các đặc trưng gì?

6.8 Một kết hợp có liên quan với nhiều hơn hai lớp sẽ được mô hình như thế nào trong UML?

6.9 Nếu một kết hợp được tìm thấy có thuộc tính hoặc thao tác, thì chúng được mô hình như thế nào trong UML?

Bài tập có lời giải



Hình 6.31: Mô hình lớp của gói Insurance *cơ sở*

6.1 Hình 6.31 trình bày một biểu đồ lớp được sử dụng lần đầu trong bài tập 4.4.

Hãy chú thích biểu đồ lớp này bằng các ràng buộc *changeability*, *ordering* và *navigability* được gợi ý trong đoạn phỏng vấn dưới đây. (Giả sử cần chỉ rõ các *navigability* hai chiều).

Mick Perez: Ông có thể cho tôi biết thêm một ít thông tin về các lớp bảo hiểm này (Mick chỉ vào biểu đồ lớp trong hình 6.31). Tôi muốn hiểu rõ cách sử dụng một số thông tin được trình bày ở đây. Loại tham chiếu chéo nào ông muốn thực hiện.

Janet Hoffner: À, khi phân tích các *con số bán ra* (sale figure) chúng tôi cần biết những *hợp đồng* (policy) nào đã được bán dưới mỗi *lược đồ* (scheme).

MP: Cần hiểu như thế nào về lược đồ phát hành hợp đồng?

JH: Bất kỳ lúc nào nhìn vào chi tiết hợp đồng chúng tôi đều biết được lịch thanh toán của nó.

MP: Giả định lược đồ phát hành hợp đồng là không đổi phải không?

JH: Đúng vậy. Tuy nhiên lịch thanh toán thì có, bởi thỉnh thoảng người ta cần thay đổi hoặc cải tiến phương thức trả tiền.

MP: Có khi nào ông muốn tìm lịch thanh toán của một hợp đồng không? ... hoặc lịch thanh toán của một giao dịch đã thanh toán?

JH: Ồ không. Tất nhiên chúng tôi muốn liệt kê lịch thanh toán của một hợp đồng. Chúng tôi cũng cần liệt kê các giao dịch thanh toán đã được thực hiện dựa vào lịch thanh toán.

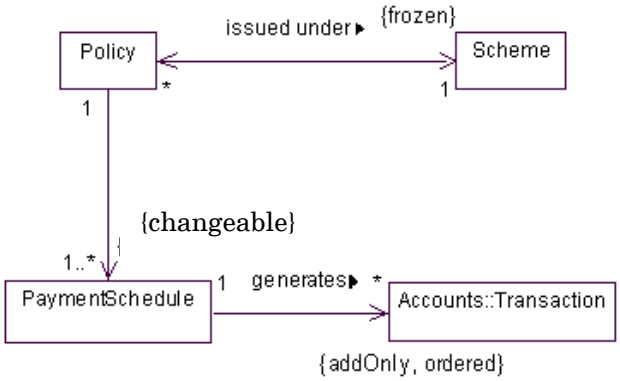
MP: Chúng ta có thể xem xét qua cách thức giao dịch được không? Chúng ta bàn đến phạm vi quy tắc tài chính. Có phải đây là một trường hợp của việc "áp dụng các luật bình thường không"?

JH: Tuỳ thuộc vào ông quan niệm "luật bình thường" là gì. Chúng ta không được phép xóa hoặc cải tiến những giao dịnh đã thanh toán. Chúng ta có thể lưu trữ chúng sau 5 năm, nhưng không được xóa chúng, nghĩa là chúng sẽ được tích lũy theo thời gian. Khi liệt kê các giao dịch chúng ta sắp xếp theo ngày tháng.

MP: Được rồi, rất tốt. Chân thành cảm ơn ông.

Lời giải:

Hình 6.32 liệt kê phần được thêm chú thích của mô hình.



Hình 6.32: Các chú thích được thêm vào mô hình của gói Insurrance.

Trong hình 6 32, các chú thích sau đây được thực hiện:

- Kết hợp *issued under* giữa *Policy* và *Scheme* được chú thích như một kết hợp hai chiều. Hơn nữa đầu *Scheme* được chú thích là {*frozen*}. Ràng buộc này ám chỉ lược đồ đã phát hành *Policy* là không thể bị thay đổi.
- Kết hợp vô danh giữa *Policy* và *PaymentSchedule* được chú thích với hướng từ *Policy* đến *PaymentSchedule*, nghĩa là không cần cài đặt hướng theo

chiều ngược lại. Hơn nữa, đầu *PaymentSchedule* được chú thích là {*changeable*}. Ràng buộc này cho biết lịch thanh toán có thể được thêm vào, thay đổi và xoá theo *multiplicity 1..**. Multiplicity này xác định tối thiểu phải có một thanh toán được gắn với một hợp đồng.

- Kết hợp *generate* giữa *PaymentSchedule* và *Accounts::Transaction* được chú thích với hướng di chuyển từ *PaymentSchedule* đến *Transaction*. Hơn nữa đầu cuối của kết hợp đã được chú thích với ràng buộc {*addOnly, ordered*}. Thành phần *addOnly* xác định rằng các giao dịch mới có thể được thêm vào, nhưng các giao dịch cũ không thể bị xoá hoặc thay đổi. Thành phần *Ordered* của ràng buộc cho biết rằng các giao dịch của *PaymentSchedule* sẽ theo thứ tự. Thực ra, thứ tự dựa trên ngày tháng không được hiển thị, nhưng có thể được đề cập bằng một ghi chú gắn vào ràng buộc *ordered*.

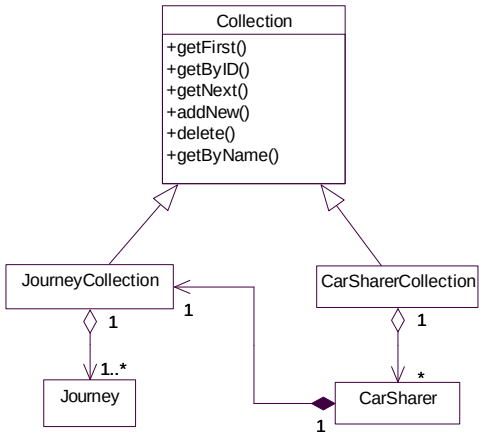
6.2 Hình 6.33 trình bày một *biểu đồ lớp đặc tả* bao gồm các lớp để đối chiếu với các thể hiện của lớp *CarSharer* và *Journey*, các thể hiện sẽ được giữ trong hệ thống *CarMatch*.

Biểu đồ lớp biểu diễn quan hệ giữa *CarSharer* và *Journey* hơi khác so với các ví dụ trước. Trong ví dụ này, quan hệ được hiện thực bởi lớp tập hợp *JourneyCollection*. Lớp *JourneyCollection* này là một thành phần của *CarSharer*.

Nó xác định *CarSharer* chỉ yêu cầu các thao tác public của lớp *Collection* trong tương tác với *JourneyCollection*. Điều này được biểu diễn như thế nào trên biểu đồ lớp?

Lời giải:

Một interface specifier là :*Collection* nên được thêm vào đầu *JourneyCollection* của kết hợp từ *CarSharer* đến *JourneyCollection*.



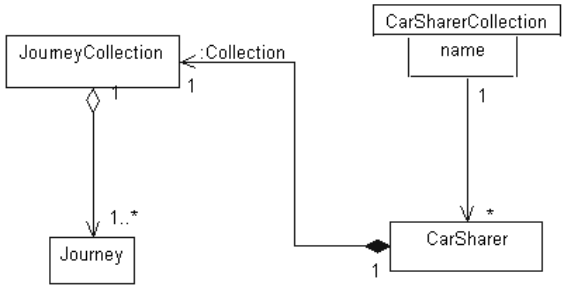
Hình 6.33: Các lớp tập hợp của Journey *và* CarSharer

6.3 Một giải thích thao tác *+getByName()* trên *Collection* trong hình 6.33 rằng *name* là một *qualifier* của kết hợp giữa

CarSharerCollection và *CarSharer*. Hãy vẽ lại kết hợp này thêm *name* vào biểu đồ. Giả sử rằng giá trị của *name* không xác định bất kỳ người thuê xe nào, hoặc xác định một vài người.

Lời giải:

Hình 6.34 là lời giải cho cả bài tập 6.2 và 6.3.



Hình 6.34: Thêm qualifier *và* interface specifier *vào biểu đồ*

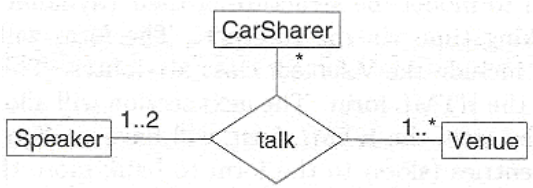
6.4 Bài tập này mở rộng case study *CarMatch* bằng cách giới thiệu các phần tử không được sử dụng ở những nơi khác. Bạn đừng lo lắng việc tích hợp mô hình lớp này với các mô hình lớp đã có.

Một dịch vụ cho thành viên, mà *CarMatch* đang đề nghị, là tổ chức các buổi hội thảo vào buổi tối. Các buổi hội thảo có mời một người dẫn chương trình. Thỉnh thoảng một buổi hội thảo có hai người tham gia dẫn chương trình. Buổi hội thảo cần đặt ra các địa điểm diễn ra hội thảo. Ví dụ, một buổi hội thảo về phòng vệ khi lái xe liên quan đến một phòng hội thảo của *CarMatch* và một bãi đậu xe để thực hành. Các buổi hội thảo khác, chẳng hạn như lái xe sao cho tiết kiệm tối đa nhiên liệu thì chỉ cần một phòng là đủ. Những người dùng chung xe đã đăng ký có thể đặt chỗ tham dự hội thảo. Khi buổi hội thảo được thiết lập lần đầu, người dẫn chương trình và nơi diễn ra hội thảo sẽ được biết trước, nhưng chưa thể biết được những thành viên nào sẽ tham gia buổi hội thảo. *CarMatch* muốn mở rộng hệ thống để hỗ trợ một sổ nhật ký cho các buổi hội thảo này.

Hãy mô hình các yêu cầu trên bằng một kết hợp n-ngôi với một lớp kết hợp, hãy thêm các thuộc tính và thao tác thích hợp vào các lớp mới này.

Lời giải:

Hình 6.35 trình bày một mô hình lớp cơ bản thể hiện multiplicity của kết hợp n-ngôi.

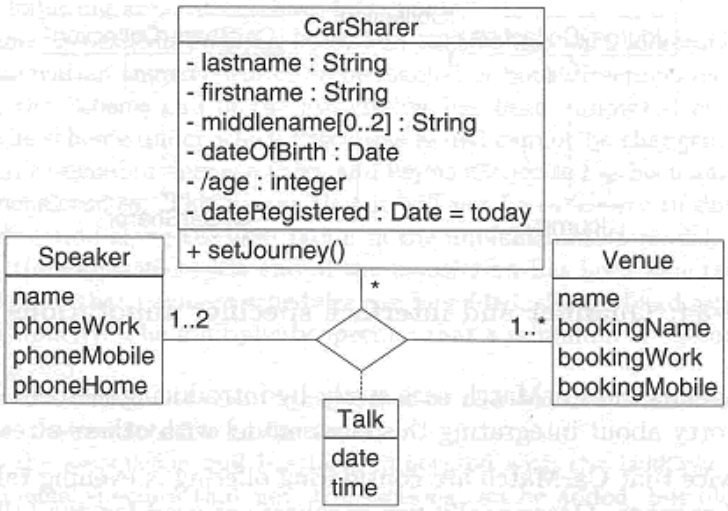


Hình 6.35: Kết hợp n-ngôi cho các buổi hội thảo của CarMatch

Một xuất hiện của *talk* có thể liên quan:

1. Giữa không và nhiều *CarSharer*. Một buổi hội thảo có thể được thiết lập mà không có *CarSharer* nào đặt chỗ (đăng ký tham gia). *CarMatch* đã không xác định là có một số lượng người tối đa có thể đặt chỗ đối với một buổi hội thảo nhưng cũng có thể là có. Thậm chí nếu một cận trên của multiplicity được thiết lập (chẳng hạn như 24), thì kết hợp vẫn được cài đặt giống với multiplicity *, nhưng nếu cận trên gần với cận dưới (chỉ hai hoặc ba), thì cài đặt sẽ khác đi.
2. Một hoặc hai (*1..2*) *Speaker*. Trường hợp này xác định rằng một buổi hội thảo có một người dẫn chương trình khi nó được sắp xếp, đồng thời cũng gợi ý rằng hai người dẫn chương trình là giới hạn bình thường.
3. Một hoặc nhiều *Venue*: Cận trên của muliplicity là * để cho phép một buổi hội thảo có thể diễn ra ở nhiều nơi. Cận dưới là *1*, bởi vì một buổi hội thảo có ít nhất một nơi khi nó được thiết lập.

Hình 6.36 trình bày bản phác thảo thứ hai của mô hình lớp.



Hình 6.36: Lớp kết hợp n-ngôi cho các buổi hội thảo của CarMatch

Các thuộc tính tích hợp cũng được thêm vào lớp *Speaker*, *Venue* và *Talk*. Các thuộc tính và thao tác của lớp *CarSharer* đã được dùng trong chương trước cũng có mặt ở đây. Kết hợp *talk* bây giờ được biểu diễn bằng một lớp kết hợp.

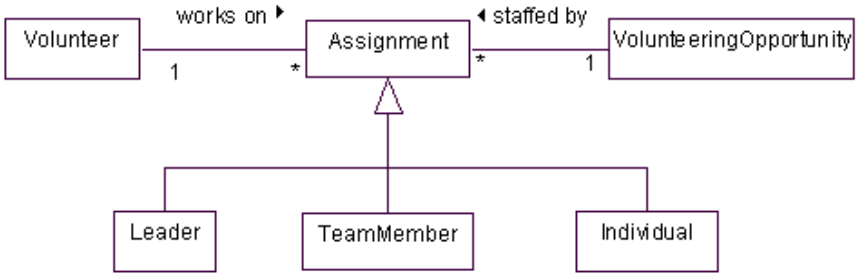
Bài tập bổ sung

6.5 VolBank yêu cầu bạn mô hình cấu trúc của giao diện dựa trên biểu mẫu HTML (động) để tình nguyện viên đăng ký qua mạng Internet. Biểu mẫu sẽ có một phần liên quan đến tình nguyện viên, bao gồm các thuộc tính của lớp *Volunteer*. Phần này phải hiện diện và không được xoá khỏi biểu mẫu HTML. Phần tiếp theo là sẽ cho phép nhập vào thời gian tình nguyện. Khi bắt đầu biểu mẫu HTML không có *slot* (chỗ) nhập *thời gian biểu gởi* nào cả. Sau đó các slot dành cho *thời gian biểu gởi* được phép thêm vào cũng như bớt đi. Giới hạn các lớp cơ bản truy xuất biểu mẫu HTML này, chỉ cần điều hướng từ biểu mẫu đến phần volunteer và đến mỗi slot nhập *thời gian biểu gởi*.

Hãy mô hình các yêu cầu trên thành một biểu đồ lớp hiển thị các ràng buộc *changeability*, *ordering* và *navigability*.

6.6 Bài tập 5.5 đã giới thiệu khái niệm của một *Assignment* của một *Volunteer* cho mộtt *Project* (*VolunteeringOpportunity*). Hình 6.37 trình bày một cấu trúc khả thi của mô hìnhh lớp này (các thao tác và các thuộc tính đã được lược bỏ).

Hãy vẽ lại hình 6.37 bằng cách dùng kết hợp n-ngôi và lớp kết hợp; thêm các thuộc tính và thao tác mà bạn tìm được từ bài tập 5.6.



Hình 6.37: Kết hợp Assignment *được mô hình như lớp*

Chương 7
BIỂU ĐỒ LỚP, CÁC KÝ HIỆU KHÁC

7.1 Giới thiệu

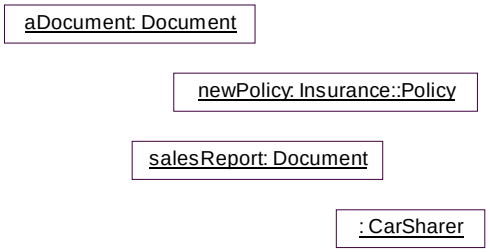
Ba chương trước đã giới thiệu các ký hiệu chủ yếu của biểu đồ lớp, những ký hiệu thường được sử dụng nhất. Chương này sẽ khảo sát các ký hiệu còn lại, những ký hiệu biểu diễn các ngữ nghĩa đặc biệt hiếm khi xuất hiện, ngoại trừ các ký hiệu đối tượng. Giống như chương trước, các ví dụ dùng trong chương này có thể biểu diễn các thành phần của các case study hơi khác so với các chương khác nhằm minh họa riêng cho các ký hiệu ở đây. Khi điều này xảy ra, chúng tôi sẽ cố gắng chỉ ra những điểm khác nhau này.

7.2 Các ký hiệu liên quan đến đối tượng

7.2.1 Các thể hiện đối tượng

Ký hiệu cơ bản biểu diễn các thể hiện đối tượng trong biểu đồ lớp đã được giới thiệu trong chương 4. Phần tên trình bày *tên* và *kiểu* lớp. Tên đối tượng có dạng *objectName : className*, tất cả đều được gạch dưới.

Hình 7.1 có vài ví dụ, *aDocument:Document* hoặc *salesReport:Document*. Có thể thêm đường dẫn, ví dụ *newPolicy:Insurance::Policy*, hoặc lược bỏ tên đối tượng, kết quả là một đối tượng vô danh, ví dụ *:CarSharer*.

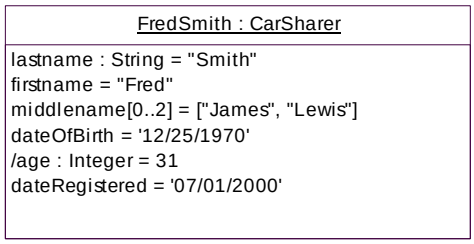


Hình 7.1: Ký hiệu cho các thể hiện đối tượng

Từ mô hình quan niệm và trong môi trường thi hành có hỗ trợ đa thừa kế, một đối tượng có thể đồng thời là thể hiện của nhiều lớp cùng lúc. Trong UML, nhiều lớp có thể được đặc tả cho một đối tượng trong một danh sách phân cách nhau bởi dấu phẩy, ví dụ *aYacht: WindPoweredBoat, MotorPoweredBoat*.

Cuối cùng dạng *objectName: ClassName[StateName]*, đặc tả đối tượng là thể hiện của một lớp trong một trạng thái nào đó (xem chương 11). Ví dụ đối tượng *:SharingAgreement[Provisional]* là thể hiện vô danh của *SharingAgreement* trong trạng thái *Provisional*.

Với đối tượng, một số hoặc tất cả các thuộc tính của nó sẽ giữ các giá trị nhằm phản ánh các tính chất của thể hiện lớp. Hình 7.2 minh họa ký hiệu xác định các giá trị thuộc tính của đối tượng. Ký hiệu có dạng *attributeName:type = value*. Hình 7.2 minh họa hai thuộc tính như vậy, *lastname* và *age*. Kiểu của giá trị phải tương thích với kiểu của thuộc tính. Kiểu thuộc tính trong đối tượng có thể được bỏ đi nếu không cần thiết. Các thuộc tính khác trong hình 7.2 sử dụng ký hiệu *name = value* cho ngắn gọn.



Hình 7.2: Ký hiệu cho thuộc tính của đối tượng

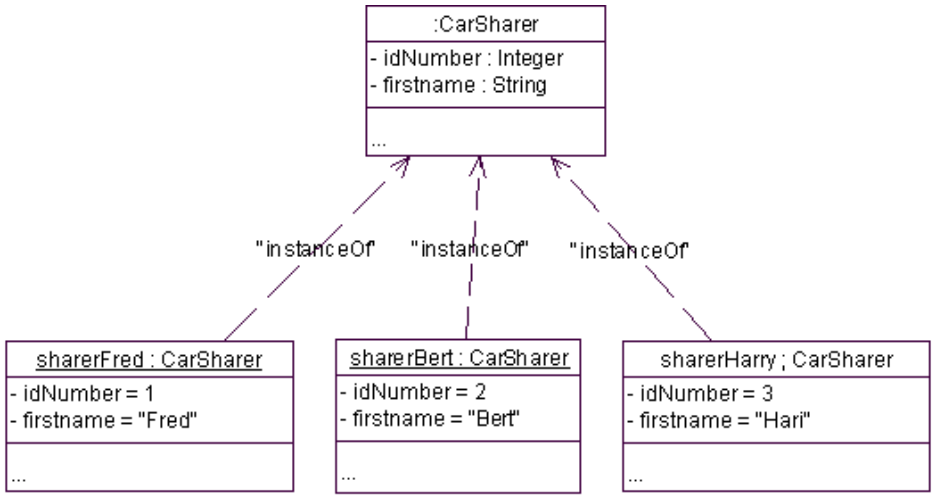
7.2.2 Quan hệ *instanceOf*

Đối tượng là thể hiện của lớp. UML cung cấp một ký hiệu để liên kết đối tượng với lớp của nó. Ký hiệu này được biểu diễn trong hình 7.3, ký hiệu là một mũi tên đứt nét có stereotype <<*instanceOf*>> (thể hiện của).

7.2.3 Liên kết

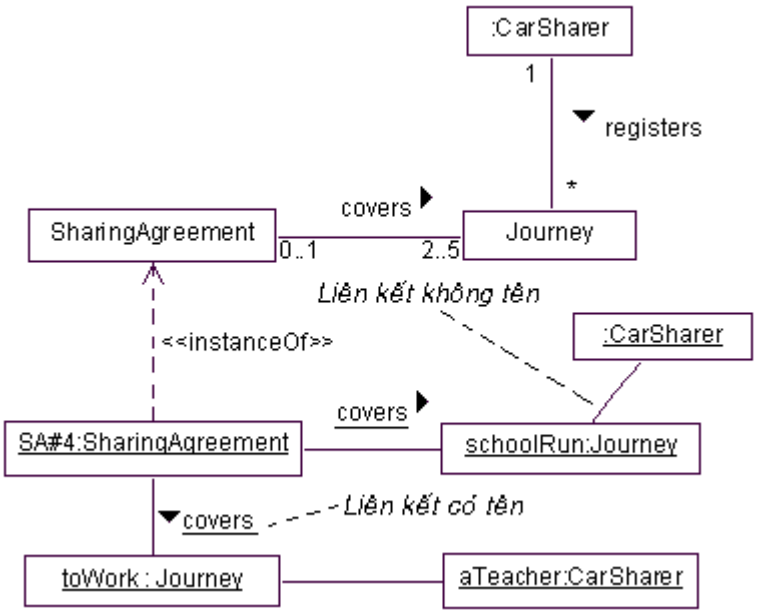
Giống thể hiện của lớp, thể hiện của kết hợp có thể được biểu diễn giữa các đối tượng. Khái niệm này đã được giới thiệu ở chương trước trong mục kết hợp n-ngôi.

Hình 7.4 minh họa ký hiệu liên kết giữa các đối tượng. Một liên kết được biểu diễn bằng một đường thẳng. Với các kết hợp n-ngôi, liên kết được biểu diễn bằng một hình thoi nối với mỗi đối tượng tham gia bằng đường thẳng liền nét (hình 7.5).



Hình 7.3: Ký hiệu <<InstanceOf>>

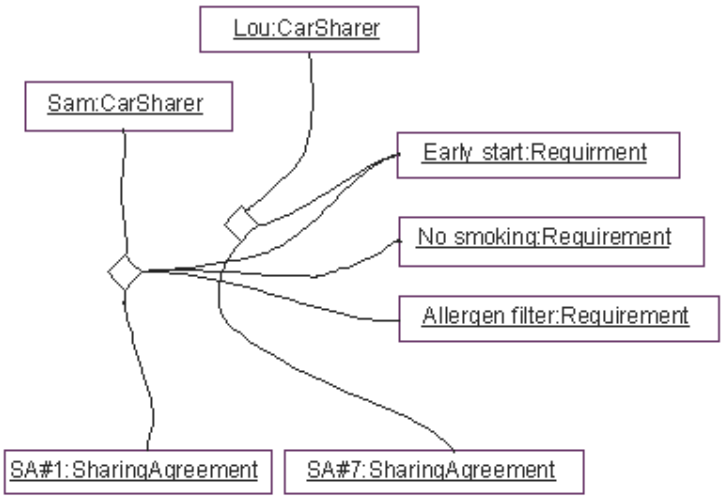
Tên kết hợp có thể được thêm vào liên kết, hình 7.4. Cũng như đối tượng, tên kết hợp được gạch dưới cho biết nó là thể hiện của kết hợp.



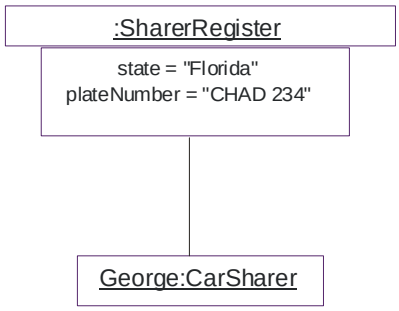
Hình 7.4: Liên kết giữa các đối tượng

Tên vai trò có thể được dùng trên các liên kết giống như khi nó được biểu diễn trên kết hợp. Các liên kết không hiển thị multiplicity, vì mỗi liên

kết nối các thể hiện đơn của đối tượng (multiplicity luôn luôn là 1-1). Các chú thích aggregation, composition và navigation có thể được biểu diễn với điều kiện là chúng nhất quán với kết hợp của liên kết. Các thể hiện *qualifier* cũng có thể được dùng (hình 7.6).



Hình 7.5: Liên kết n-ngôi



Hình 7.6: Thể hiện qualifier

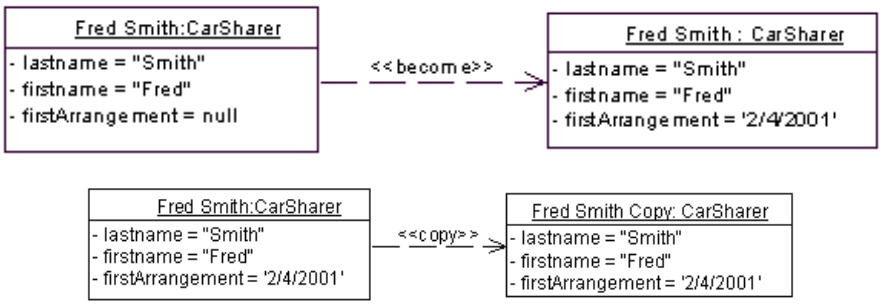
7.2.4 Biểu đồ đối tượng

Biểu đồ đối tượng được xây dựng từ các đối tượng và các liên kết. Hình 7.4, 7.5, 7.6 minh họa các biểu đồ đối tượng đơn giản. Những đối tượng và liên kết trong biểu đồ đối tượng có thể kết hợp với bất kỳ phần tử đối tượng nào đã thảo luận ở đây (ví dụ giá trị thuộc tính, tên vai trò liên kết, trạng thái lớp). Một biểu đồ như vậy sẽ hữu ích trong giai đoạn phân tích của qui trình phát triển để khảo sát trạng thái của lớp và kết hợp.

Một biểu đồ đối tượng bao gồm các đối tượng và các liên kết biểu diễn một bức ảnh của hệ thống tại một thời điểm nào đó. Vì vậy các đối tượng

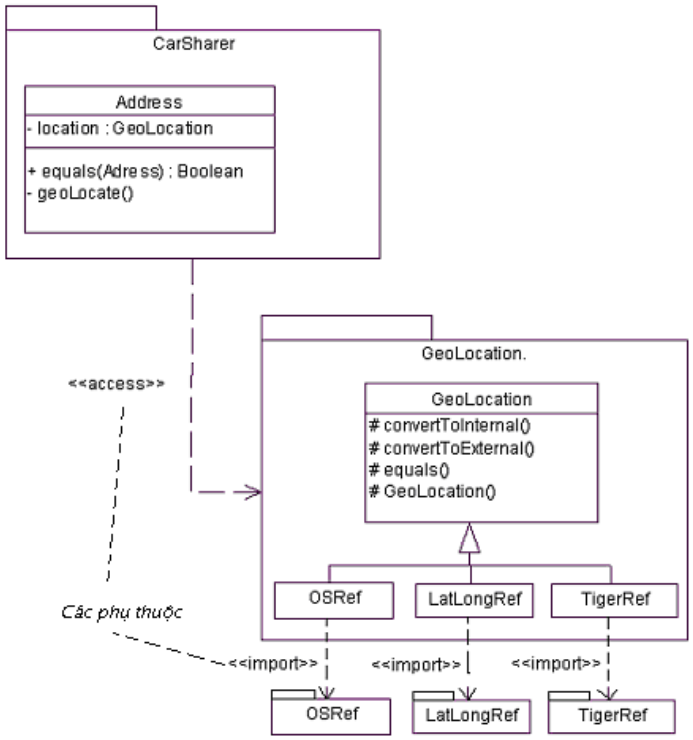
sẽ có trạng thái cố định trên biểu đồ lớp. UML cũng cung cấp một ký hiệu để cho biết ảnh hưởng của các thay trạng thái của đối tượng bằng cách sử dụng các quan hệ <<*become*>> và <<*copy*>> như trong hình 7.7.

Dù *Fred Smith* và *Fred Smith Copy* có cùng trạng thái, nhưng chúng là các thể hiện khác nhau và có tên gọi khác nhau trong hệ thống cài đặt.



Hình 7.7: Cho biết sự thay đổi trạng thái trong biểu đồ đối tượng

7.3 Phụ thuộc

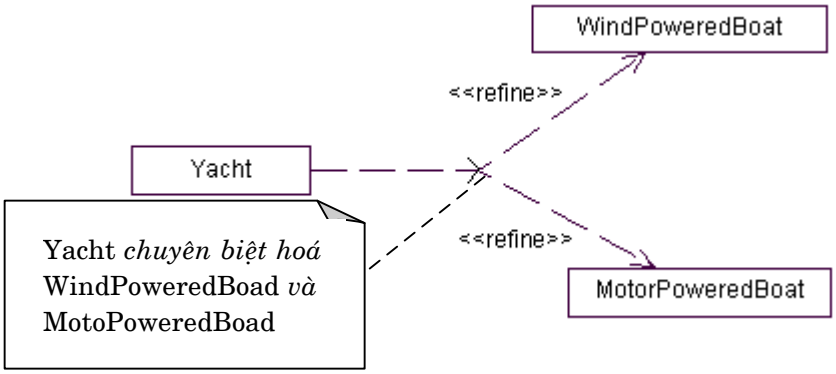


Hình 7.8: Các quan hệ phụ thuộc

Các phụ thuộc có thể được dùng để đặc tả mối quan hệ giữa các thành phần của mô hình. Ký hiệu phụ thuộc là một mũi tên đứt nét xuất phát từ thành phần phụ thuộc (hình 7.8)

Trong hình, phụ thuộc *<<access>>* giữa các gói *CarSharer* và *GeoLocation* chỉ ra rằng các lớp trong *CarSharer* sẽ sử dụng các lớp *public* trong *GeoLocation*. Không phải tất cả các lớp trong *GeoLocation* đều là public. Các kết hợp không được tạo ra giữa các lớp trong *CarSharer* và các lớp không phải là public trong *GeoLocation*.

Phụ thuộc có thể được vẽ thành các đường phân kỳ hoặc hội tụ. Hình 7.9 minh họa các đường phân kỳ. Ở chỗ phân kỳ, như *<<refine>>* của *Yacht* trên *WindPowerdBoat* và *MortorPoweredBoat*, một chú thích được gắn vào điểm phân nhánh để làm rõ phụ thuộc. Với đường đồng quy, chú thích được gắn vào điểm đồng quy.



Hình 7.9: Các đường phụ thuộc phân kỳ

7.4 Các đặc trưng phạm vi lớp

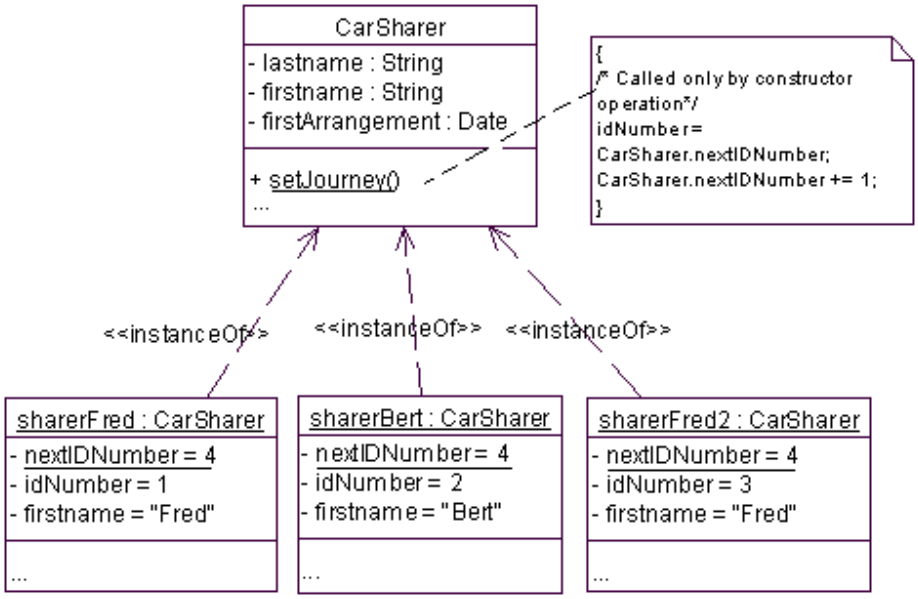
Các thuộc tính và thao tác thường được xem là các đặc trưng sẽ được biểu thị bởi đối tượng. Nói cách khác, phạm vi của chúng ở mức thể hiện. Các đối tượng khác nhau sẽ giữ những giá trị thuộc tính khác nhau. Hai đối tượng khác nhau có thể có thuộc tính *name* có cùng giá trị là *"Fred"*, nhưng thực ra chúng là hai *thể hiện chuỗi* có giá trị giống nhau.

Đôi khi chúng ta muốn một thuộc tính lớp có một giá trị chung cho tất cả các đối tượng của lớp. Hình 7.10 minh họa ba thể hiện của *CarSharer* là *sharerFred*, *sharerBert* và *sharerFred2*.

Ở đây, lớp *CarSharer* đặc tả một thuộc tính phạm vi lớp, *nextIDNumber*, và một thao tác phạm vi lớp, *setIDNumber*. Thuộc tính phạm vi lớp giữ cùng một giá trị cho tất cả các thể hiện của *CarSharer*. Thao tác *setIDNumber* dùng *nextIDNumber* gán cho *idNumber* cho mỗi thể hiện

của lớp khi nó được tạo ra. Khi thể hiện được tạo và gọi thao tác *setIDNumber*, giá trị của *nextIDNumber* được tăng lên sẵn sàng cho việc tạo ra thể hiện kế tiếp.

Hình 7.10 minh họa các thể hiện sau khi tạo ra *sharerFred2*. Lúc này, *nextIDNumber* có giá trị là 4. Khi *sharerFred* được tạo, *nextIDNumber* có giá trị 1 và được gán cho thể hiện này, sau đó nó sẽ lần lượt tăng lên qua việc khởi tạo các đối tượng *sharerBert* và *sharerFred2*.



Hình 7.10: Các đặc trưng phạm vi lớp

Trong UML, các thuộc tính và thao tác phạm vi lớp được gạch chân. Các cách dùng khác về thuộc tính lớp có thể là: đặt kích thước mặc định cho các thành phần đồ họa, như hình dạng, cửa sổ ...; hoặc cung cấp các giá trị mặc định cho các chuỗi văn bản, thay vì cài đặt trong một thao tác.

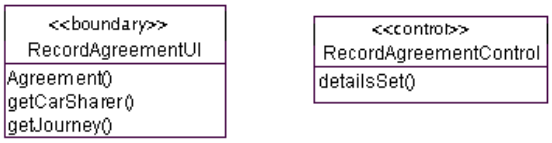
7.5 Các kiểu lớp

Stereotype được dùng trong UML như một phương tiện đặc tả một phần tử mô hình tuân theo một mẫu hành vi dễ hiểu. Nó được dùng rộng rãi trong đặc tả UML (OMG, 1999b; OMG, 1999c). Trong mục này, các stereotype chung được đưa ra trước sau đó là các stereotype đặc biệt.

7.5.1 Stereotype

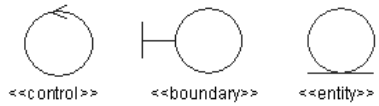
Trong ngữ cảnh của biểu đồ lớp, cơ chế *stereotype* thường được áp dụng cho các lớp, các kết hợp và các phụ thuộc khác. Hình 7.11 trình bày ký

hiệu *stereotype* cơ bản của lớp, bằng cách thêm <<*stereotype*>> vào trước tên lớp.



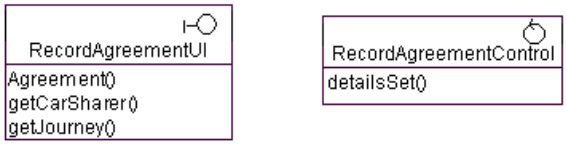
Hình 7.11: Ký hiệu stereotype

Trong hình 7.11, lớp *RecordAgreementUI* theo khuôn dạng <<*boundary*>> (như một giao diện người dùng) và lớp *RecordAgreementControl* theo khuôn dạng <<*control*>>. Hành vi của các *stereotype* này rất dễ hiểu trong mô hình hướng đối tượng và mang một ý nghĩa riêng đối với người thiết kế và người phát triển. (Ví dụ lớp *control* phối hợp sự tương tác giữa các lớp khác có liên quan với nhau).



Hình 7.12: Ba biểu tượng stereotype *của Jacobson*

Thêm *stereotype* trước tên lớp là cách phổ biến. Tuy nhiên, UML cũng hỗ trợ thêm các ký hiệu khác. Ban đầu, *Jacobson và các cộng sự* (1992) dùng ký hiệu tượng hình để biểu diễn ba khuôn dạng lớp, đó là *boundary*, *control* và *entity* như trong hình 7.12.

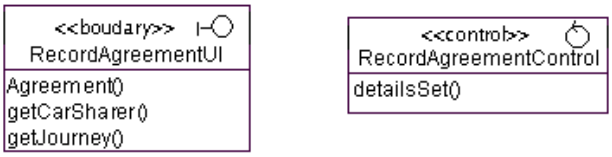


Hình 7.13: Biểu tượng stereotype *nằm trong biểu tượng lớp*

UML cho phép dùng những khuôn dạng theo kiểu tượng hình (hoặc bất kỳ biểu tượng nào do người dùng định nghĩa) để biểu diễn khuôn dạng của một lớp. Ví dụ, hình 7.13 minh họa một đặc tả khuôn dạng giống như hình 7.11, tuy nhiên *stereotype* của chúng là các biểu tượng.

Hình 7.14: Biểu tượng lớp co lại vừa với biểu tượng khuôn dạng

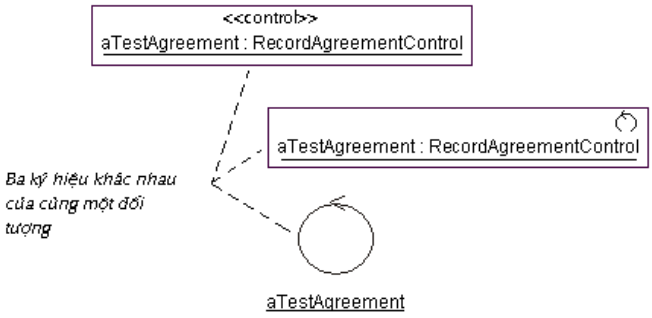
Với công việc ban đầu của Jacobson, UML cho phép biểu tượng lớp co lại ở mức tối thiểu chỉ còn biểu tượng khuôn dạng và tên lớp (hình 7.14).



Hình 7.15: Kết hợp hai ký hiệu khuôn dạng

Nếu muốn, UML cho phép ký hiệu có nhãn <<*stereotype*>> kết hợp với ký hiệu biểu tượng nhỏ (xem hình 7.15).

Ký hiệu cho một đối tượng (như giới thiệu trong chương 4, và sẽ được thảo luận chi tiết trong chương này), cũng có thể dùng các qui ước khuôn dạng mà chúng ta đang thảo luận. Ví dụ, một đối tượng có thể bao gồm tên hoặc biểu tượng nhỏ của khuôn dạng. Hình 7.16 minh họa các ký hiệu tạo khuôn dạng cho các đối tượng.



Hình 7.16: Ví dụ minh hoạ cho các ký hiệu khuôn dạng của đối tượng

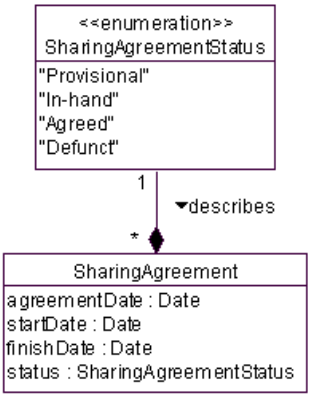
7.5.2 Kiểu liệt kê (Enumeration)

Đôi khi rất hữu ích, định nghĩa trước một tập hữu hạn các giá trị. Hình 7.17 minh họa khuôn dạng *enumeration* trong UML.

Ký hiệu cơ bản cho *enumeration* là ký hiệu lớp. Khuôn dạng <<*enumeration*>> cho biết lớp này biểu diễn một tập hữu hạn các giá trị. Các giá trị được đặt trong phần giữa của ký hiệu lớp.

Trong hình 1.17, *SharingAgreementStatus* là một *enumeration* có các giá trị được đặt tên "*Provisional*", "*In-hand*", "*Agreed*" và "*Defunct*". Nó được kết hợp với lớp *SharingAgreement* và được cài đặt như một thuộc tính của *SharingAgreement* có kiểu *SharingAgreementStatus*.

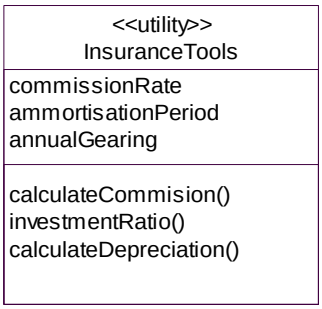
Một *enumeration* có thể có thao tác. Các thao tác được liệt kê trong phần cuối của ký hiệu lớp.



Hình 7.17: Ký hiệu <<enumeration>>

7.5.3 Tiện ích (Utility)

UML cung cấp khuôn dạng <<*utility*>> để mô hình một tập các thuộc tính và các thao tác toàn cục. Giống *enumeration*, một *utility* được đặt cơ sở trên ký hiệu lớp. Hình 7.18 minh họa cho ký hiệu *utility*.



Hình 7.18: Ký hiệu <<utility>>

Trong hình 7.18, utility *InsuranceTools* cung cấp các thuộc tính và các thao tác có thể được sử dụng trên khắp hệ thống *CarMatch*. Không cần thiết phải kết hợp một utility với các lớp dùng các thuộc tính và các thao tác của nó, bởi vì tất cả chúng là toàn cục.

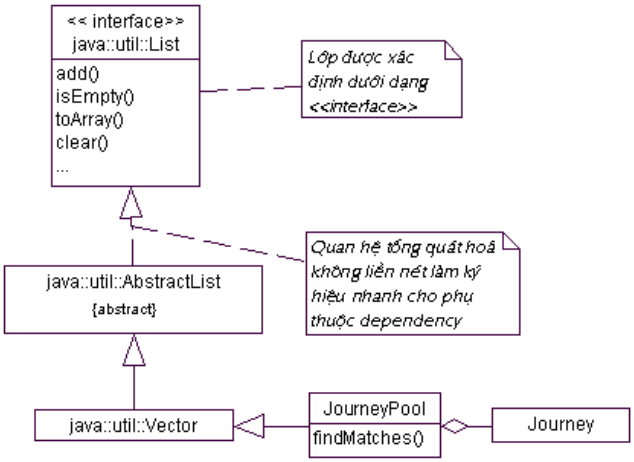
7.5.4 Interface (giao diện)

Interface là các lớp không có thuộc tính và không có các thể hiện trực tiếp (cho nên, chúng là trừu tượng). Các thao tác của một lớp *interface*

không có phương thức, nghĩa là chúng không được cài đặt. Các lớp *interface* không thể tham gia vào các kết hợp hướng đến đến lớp khác vì đây là một dạng thuộc tính của lớp. Các *interface* chỉ có thể tham gia vào các kết hợp hướng đến nó mà thôi.

Tập các thao tác không phải là *private* của lớp *interface* sẽ xác định chức năng phải được hỗ trợ bởi bất kỳ lớp nào *hiện thực* (realize) nó. Chúng ta nên nhớ rằng lớp *interface* tự nó không cung cấp cài đặt cho các thao tác của chính nó; các thao tác này đều không có phương thức.

Khái niệm hiện thực một *interface* giống với khái niệm hiện thực của tổng quát hóa đã được đề cập trong chương 5. Điểm *khác biệt chính* là một lớp hiện thực *interface* chịu trách nhiệm cài đặt các thao tác của *interface* hơn là thừa kế như trong tổng quát hóa. Hình 7.19 minh họa cách dùng *interface* trong UML sử dụng các lớp từ thư viện lớp của Java. Lớp *List* trong hình minh họa cách đặc tả khuôn dạng *interface* bằng từ khóa <<*interface*>>.



Hình 7.19: Ký hiệu interface

Hình 7.19 biểu diễn một ký hiệu UML thay cho phụ thuộc <<*realize*>>, theo đó quan hệ tổng quát hóa được vẽ bằng một đường thẳng đứt nét thay vì một đường thẳng liền nét.

Lớp giao diện *List* (danh sách) đặc tả một họ được sắp các đối tượng, nó cung cấp các thao tác nhằm hỗ trợ:

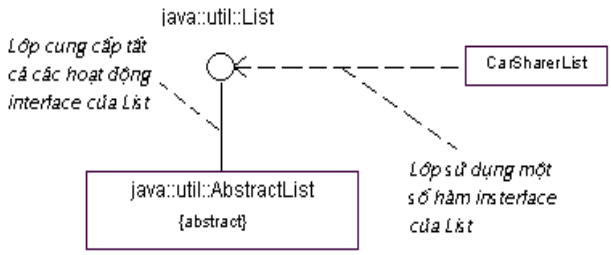
- Việc chèn các đối tượng vào danh sách (ví dụ *add*(*Object*) cho phép một đối tượng được thêm vào cuối danh sách).

- Truy vấn một danh sách liên quan nội dung của nó (ví dụ *isEmpty*() trả về giá trị *true* nếu danh sách khác rỗng).
- Chuyển giao danh sách đối tượng (ví dụ *toArray*() trả về một đối tượng kiểu *Array* chứa tất cả các phần tử của danh sách).
- Xóa tất cả các đối tượng ra khỏi danh sách (ví dụ *clear*() xóa tất cả các phần tử ra khỏi danh sách).

Nhắc lại, lớp *List* tự nó không cài đặt những thao tác này, bởi vì nó là một lớp *interface*. Trong hình 7.19, lớp *AbstractList* hiện thực *List* cho nên nó cung cấp các phương thức nhằm hỗ trợ các hàm được xác định trong lớp *List*.

Hình 7.19 minh họa sự tinh chế từ *List*, qua *AbstractList* đến *Vector*. Nó cũng minh họa làm thế nào mà một lớp trong gói *CarSharer*, đó là *JourneyPool*, có thể thừa kế hành vi của các lớp tiền bối của nó. Trong ví dụ này, lớp *JourneyPool* cung cấp các thao tác giống lớp *Vector* cộng thêm thao tác *findMatch*(). Chú ý khi thể hiện của *Journey* được đăng ký với *CarMatch* thì cũng được thêm vào vật chứa *JourneyPool* bằng cách sử dụng hành vi của *Vector*. Lớp *JourneyPool* rồi sẽ cung cấp một thao tác cụ thể *findMatches* để đối sánh các lộ trình tương tự nhau có thể chia sẻ được. Thao tác *findMatches* phải tận dụng tốt các thao tác của *Vector*, *AbstractList* để thực hiện khả năng đối sánh này.

Với một hình thức ký hiệu đơn giản hơn, UML cho phép một *interface* được biểu diễn bằng một vòng tròn nhỏ có nhãn là tên của *interface*. Hình 7.20 dùng lớp *List* đã trình bày trong hình 7.19 để minh họa ký hiệu này.



Hình 7.20: Ký hiệu khác của lớp interface

Lớp *AbstractList* cung cấp các thao tác của giao diện *List*, vì thế một đường thẳng liền nét gắn nó vào *List*. Ví dụ trong hình 7.20 cũng giới thiệu lớp *CarSharerList*, minh họa ký hiệu mũi tên đứt nét liên kết đến *List*, yêu cầu một số thao tác (không nhất thiết phải tất cả) trên *interface* này.

7.5.5 Lớp kiểu và lớp cài đặt

UML phân biệt các lớp <<*type*>> với các lớp <<*implementationClass*>>. Lớp *type* tương tự với khái niệm *interface* bởi nó định nghĩa các thao tác nhưng không cài đặt phương thức. Tuy nhiên, lớp *type* khác với lớp *interface* ở chỗ nó có thể có các thuộc tính và có thể tham gia vào các kết hợp. Các thuộc tính và các thao tác được cung cấp chỉ nhằm để hỗ trợ cho việc đặc tả thao tác.

Việc phân biệt sự khác nhau giữa lớp *type* và lớp *asbtract* rất quan trọng (lớp trừu tượng là lớp có tính chất *isAbstract = true*). Chúng đều không có các thể hiện đối tượng trực tiếp. Tuy nhiên lớp *type* khác với lớp *asbtract* ở chỗ các thao tác của một lớp *type* không được cài đặt. Một lớp *abstract* có thể cung cấp các phương thức cho tất cả các thao tác của nó, nhưng không cho phép có các thể hiện trực tiếp.

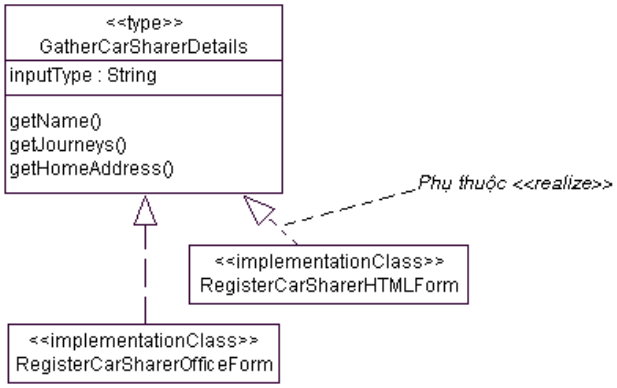
Tóm lại:

- **Lớp <<*interface*>>** không có các thuộc tính hoặc các kết hợp hướng ra bên ngoài. Các thao tác của nó không có phương thức; chúng là các thao tác giữ chỗ để bảo đảm rằng một lớp nào khác hiện thực nó sẽ cung cấp tất cả các thao tác cần thiết. Với định hướng đặc tả và cài đặt, do lớp *interface* không thực sự làm điều gì, nên nó không thể có các thể hiện đối tượng trực tiếp.
- **Lớp <<*type*>>** có các thao tác, các thuộc tính và các kết hợp hướng ra ngoài. Các thuộc tính và các kết hợp này nhằm hỗ trợ việc đặc tả các thao tác của lớp. Các thao tác của lớp *type* không có phương thức - chúng là các thao tác giữ chỗ nhằm đảm bảo rằng một lớp khác cài đặt lớp này sẽ cung cấp tất cả các thao tác cần thiết. Với định hướng đặc tả và cài đặt, một lớp *type* không thực sự làm điều gì nên nó không có các thể hiện trực tiếp.
- **Lớp {*abstract*}** có thể có thuộc tính, thao tác và kết hợp. Các thao tác được cài đặt. Một lớp trừu tượng có thể hỗ trợ các thể hiện đối tượng trực tiếp bởi vì nó có các thuộc tính, các thao tác có phương thức, và các kết hợp. Tuy nhiên, các lớp trừu tượng không có các thể hiện đối tượng trực tiếp đơn giản vì chúng là trừu tượng.

Ký hiệu UML cho lớp *type* dựa trên ký hiệu lớp thêm khuôn dạng <<*type*>>. Lớp *GatherCarSharerDetails* trong hình 7.21 minh họa cho ký hiệu này.

Căn cứ vào thảo luận bên trên, không có gì ngạc nhiên khi thấy rằng lớp <<*implementationClass*>> là hiện thực của lớp *type* bằng cách cài đặt các

phương thức của lớp *type*. Hình 7.21 minh họa hai lớp *implementationClass* là các lớp *RegisterCarSharerOfficeForm* và *RegisterCarSharerHTMLForm* (biểu mẫu văn phòng và biểu mẫu HTML). *RegisterCarSharerOfficeForm* cho phép nhân viên *CarMatch* thu thập chi tiết về người muốn chia sẻ xe khi người ấy đến vào văn phòng *CarMatch* đăng ký. *RegisterCarSharerHTMLForm* được dùng trên *Internet* cho phép đăng ký trực tuyến. Hình 7.21 minh họa hai phụ thuộc <<*realize*>> giữa các lớp *implementationClass* và lớp *type*.



Hình 7.21: Ký hiệu cho lớp <<type>> *và lớp* <<implementation>>

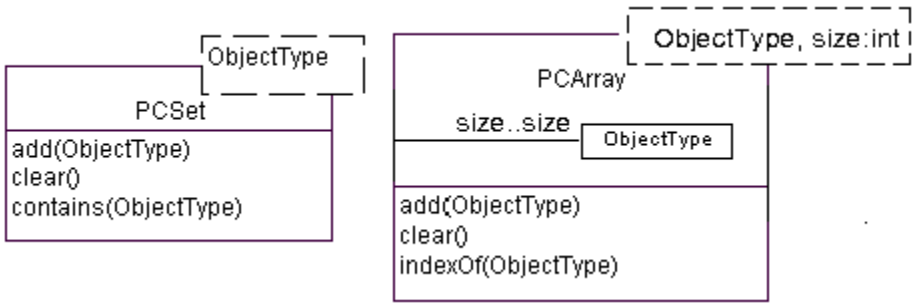
Ví dụ trong hình 7.21, lớp type *GatherCarSharerDetails*, minh họa chức năng chung được chờ đợi trong bất kỳ giao diện nào được dùng để nhập chi tiết thành viên. Giống như các thao tác, thuộc tính *inputType* cũng được hiển thị. Điểm lưu ý ở đây là bất cứ phương tiện nào được dùng để nhập chi tiết thành viên cũng sẽ được lưu ý thông qua thuộc tính *inputType*. Với mô hình lớp như trên cho phép *CarMatch* lấy thông tin đăng ký trực tuyến cùng với đăng ký ở văn phòng.

Cài đặt của *GatherCarSharerDetails* được biến đổi thông qua hai lớp *cài đặt* để phù hợp với các yêu cầu của môi trường riêng mà nó chạy (qua *Internet* hoặc cửa sổ màn hình). Mỗi lớp *cài đặt* có thể có các chức năng bổ sung để đáp ứng các yêu cầu của môi trường hoặc để cung cấp thêm giá trị để tô điểm cho giao diện. Ví dụ việc cài đặt biểu mẫu cửa sổ cho phép dùng trực tiếp chức năng của cửa sổ, chẳng hạn như kéo và thả các biểu tượng một cách đơn giản mà biểu mẫu dựa trên HTML không có.

7.6 Lớp tham số và các thành phần bị buộc

Ký hiệu UML cho *lớp tham số* (parameterized), hay *lớp mẫu* (template), tương tự như ký hiệu lớp cơ bản thêm các tham số được đặt trong một hộp có đường viền đứt nét đặt ở góc trên bên phải của ký hiệu lớp. Hình

7.22 minh họa ký hiệu lớp tham số của hai lớp khác nhau, *PCSet* và *PCArray*.



Hình 7.22: Ký hiệu lớp tham số cho hai lớp khác nhau

UML cũng cho phép chúng ta lập mô hình đồ họa các thành phần tham số của *lớp tham số*. Ví dụ, trong hình 7.22 lớp *PCArray* minh họa rằng nó có một kết hợp *composite* với *ObjectType* được truyền như tham số. (*Composition* được trình bày bằng cách dùng đối tượng chứa dạng đồ họa). *Multiplicity* của kết hợp này được định nghĩa bởi tham số *size*. Tóm lại, định nghĩa lớp trong hình 7.22 chỉ định rằng số lần xuất hiện của lớp *ObjectType*, được chứa trong lớp *PCArray*, là con số nằm giữa *size* và *size*, nói khác đi đây là một mảng size..size các *ObjectType*.

Ngoài lề

Đặc tả UML không định nghĩa tường minh miền size..size *được rút gọn thành* size.

Vậy *lớp tham số* dùng để làm gì? Ứng dụng thông dụng nhất của các *lớp tham số* là để hỗ trợ các *collection* đối tượng. Hình 7.23 minh họa cho điều này, cụ thể hai *lớp tham số* của hình 7.22.

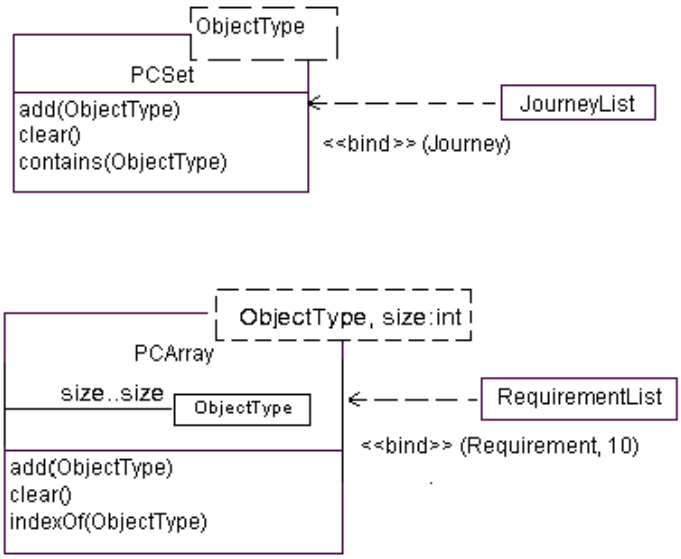
Trong hình 7.23, lớp *JourneyList* bị buộc với lớp tham số *PCSet* bằng phụ thuộc <<*bind*>>. Liên kết này cụ thể hoá tham số *ObjectType* là *Journey*. Như vậy các thao tác của *PCSet* cũng được cụ thể, ví dụ *add*(*Journey*). Do đó, hiệu quả cuối cùng là lớp *JourneyList*, là một lớp *PCSet* cụ thể, cư xử như một *tập hợp* các phần tử kiểu *Journey*.

Ví dụ 7.1

Ý nghĩa của liên kết giữa *RequirementList* và *PCArray*.

Lời giải:

RequirementList bị buộc với *PCArray* đặt *ObjectType* là *Requirement* và *size* là 10. Kết quả được lớp *RequirementList* như là một mảng có thể giữ tối đa 10 lần xuất hiện đối tượng kiểu *Requirement*.

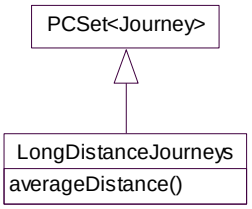


Hình 7.23: Ví dụ về lớp tham số

Với ví dụ trong hình 7.23, *JourneyList* và *RequirementList* được đề cập trong UML như là các *phần tử bị buộc (bound element)*. Chúng có thể được biểu diễn bằng cách dùng ký hiệu như trong hình 7.23. Một ký hiệu tương đương với ký hiệu này được biểu diễn trong hình 7.24.

Hình 7.24: Ký hiệu khác để biểu diễn các phần tử bị buộc

Ký hiệu tương đương này gọn hơn ký hiệu đầy đủ dùng trong hình 7.23. Tên lớp có dạng *templateName <parameterList>*. Tuy nhiên, ký hiệu này lại không cho phép phần tử bị buộc có tên riêng.



Hình 7.25: Chuyên biệt hóa một lớp bound

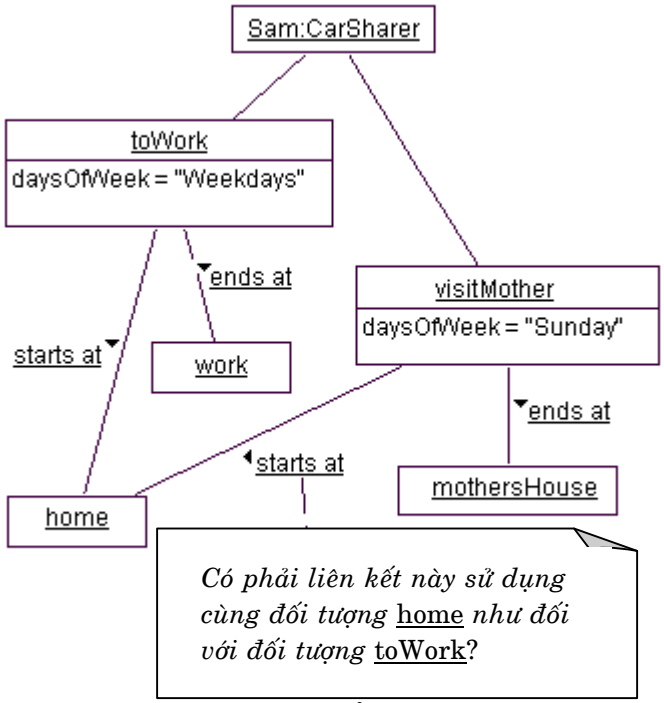
Khi lớp *bound* được đặc tả đầy đủ, nó không thể mở rộng chức năng của lớp tham số bằng cách bổ sung các thuộc tính và các thao tác riêng. Tuy nhiên, hoàn toàn có thể chuyên biệt hoá một lớp *bound* (xem hình 7.25)

7.7 Hướng dẫn lập mô hình

7.7.1 Các ký hiệu liên quan đến đối tượng

Vẽ biểu đồ đối tượng là cách rất hữu ích để thấu hiểu hệ thống. Các đối tượng và liên kết được sử dụng để khảo sát các cấu trúc kết hợp tiềm năng bằng cách minh họa các cấu hình có thể có của các đối tượng và liên kết này. Hình 7.26 minh họa thử nghiệm đầu tiên trong việc mô hình các lớp và các kết hợp chính trong *CarMatch*.

Các chương đầu đã đưa ra mô hình lớp cho *CarSharer*, *Journeys* và *Addresses*. Tuy nhiên, mục đích của mục này là minh họa cho lợi ích của biểu đồ đối tượng trong các giai đoạn đầu của qui trình lập mô hình. Như vậy, khía cạnh này được xem xét như là một thử nghiệm đầu tiên.

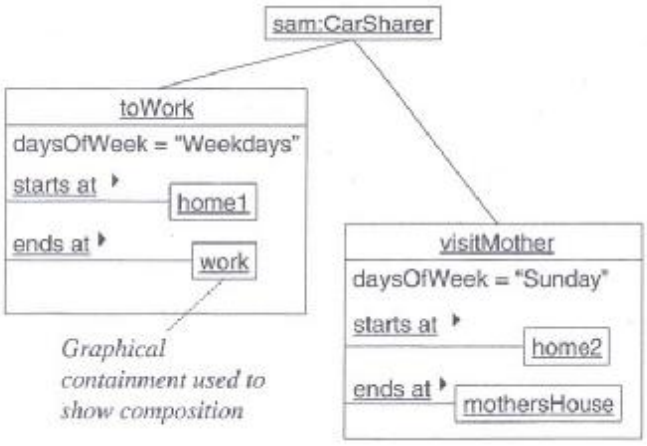


Hình 7.26: Phác thảo đầu tiên biểu đồ đối tượng *của CarMatch*

Biểu đồ phác thảo đầu tiên này nổi bật một vấn đề trong cách giải thích mục tiêu của chuyến đi là đi từ đâu hoặc đi đến đâu. Người phân tích đã

dùng một ghi chú để cho thấy họ không chắc có hay không cùng một đối tượng *home* được sử dụng trong liên kết *from* cho hai lộ trình khác nhau.

Người phân tích vẽ lại biểu đồ đối tượng dùng một cấu trúc hơi khác. Hình 7.27 minh họa bản phác thảo thứ hai này.

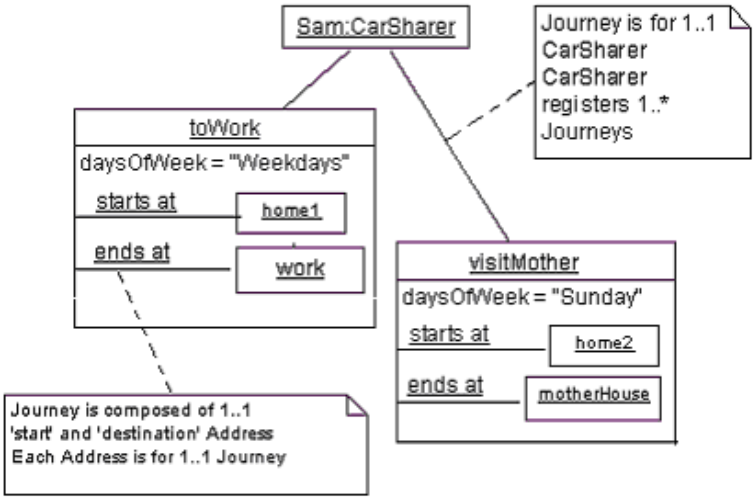


Hình 7.27: Phác thảo lần hai biểu đồ đối tượng *của CarMatch*

Biểu đồ thứ hai chỉ ra các vị trí *from* và *to* như là các thành phần của mỗi lộ trình dùng biểu diễn *composition* dạng đồ họa. Dùng ký hiệu này, cần phải dùng các đối tượng riêng biệt cho hai lần xuất hiện của vị trí *home* đối với hai lộ trình khác nhau. Việc vẽ lại các đối tượng bằng cách này làm cho người phân tích nghĩ về trạng thái tự nhiên của việc tồn tại các vị trí khác nhau, sẽ làm rõ vai trò của chúng trong phạm vi bài toán và trình bày chúng trong phạm vi lời giải. Dựa vào đây người phân tích sẽ thấy rằng phác thảo thứ hai này mô tả chính xác hơn phác thảo thứ nhất. Hay nói một cách khác, mối quan hệ giữa *trip* và *location* trong phác thảo thứ hai chính xác hơn trong phác thảo thứ nhất.

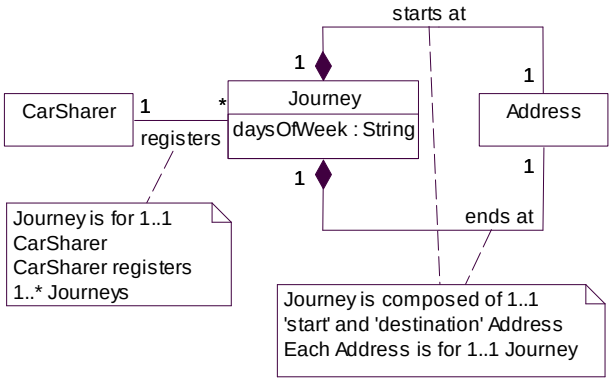
Bước tiếp theo là chọn tên kiểu (tên lớp) thích hợp để đặt cho các thể hiện. Người phân tích đã dùng tên kiểu *CarSharer* và theo cách đó dùng tiếp các thuật ngữ không chính thức *trip* và *location*. Sau khi xem xét biểu đồ đối tượng chi tiết hơn cùng với khách hàng và dùng ngôn ngữ của khách hàng (*CarMatch*), người phân tích sẽ chọn lựa tên kiểu (tên lớp) thích hợp cho các đối tượng. Trong case study này, *trip* trở thành *Journey* và *location* trở thành *Address*. Hình 7.28 trình bày một biểu đồ lớp đã được cải thiện, mỗi đối tượng đều đã có kiểu lớp. Ở đây người phân tích cũng lưu ý thêm về *multiplicity* của các kết hợp được biểu diễn bằng các liên kết trong biểu đồ đối tượng.

Cuối cùng, người phân tích sẽ chuyển từ thể hiện sang kiểu, tức là chuyển biểu đồ đối tượng sang biểu đồ lớp. Anh ta vẽ một biểu đồ lớp phản ánh các kiểu chung được minh họa trong biểu đồ đối tượng. Hình 7.29 trình bày biểu đồ lớp được vẽ từ biểu đồ đối tượng trong hình 7.28.



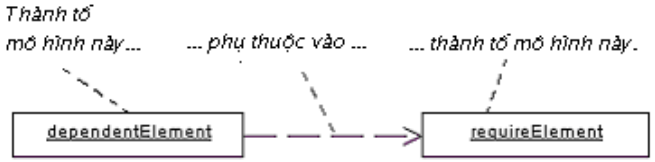
Hình 7.28: Biểu đồ đối tượng trình bày các kiểu lớp cho CarMatch

Như vậy, biểu đồ đối tượng, với các đối tượng và liên kết, là một công cụ cực kỳ hữu ích trong qui trình phân tích để khảo sát các khía cạnh đặc biệt của *lĩnh vực bài toán*. Việc sử dụng biểu đồ đối tượng để đưa ra các cấu trúc có thể có trong *lĩnh vực bài toán* là một công cụ cho phép giao tiếp hiệu quả hơn với người dùng.



Hình 7.29: Biểu đồ lớp trình bày các kiểu lớp cho CarMacth

7.7.2 Phụ thuộc *(dependency)*

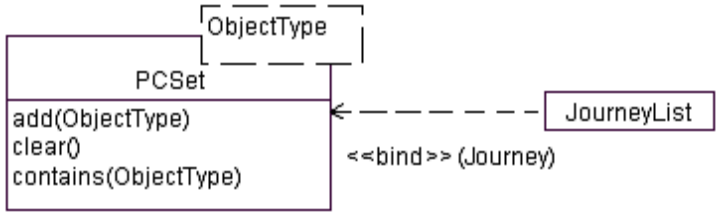


Hình 7.30: Minh họa cho phụ thuộc giữa các phần tử mô hình

Phụ thuộc cho biết một thành phần mô hình cần sự có mặt của một thành phần mô hình khác. Ý nghĩa của phụ thuộc là nếu thành phần bên trên của phụ thuộc thay đổi thì thành phần phụ thuộc cũng cần được thay đổi.

Thực chất, yêu cầu của phụ thuộc được xác định bằng khuôn dạng. Các phụ thuộc rơi vào 4 phạm trù chính là *binding* (buộc), *abstraction* (trừu tượng), *usage* (sử dụng) và *permission* (thẩm quyền).

Binding: Các phụ thuộc *binding* chỉ có một khuôn dạng, đó là <<*bind*>>. Phụ thuộc này buộc một phần tử đến lớp mẫu (lớp tham số). Phụ thuộc nên có một danh sách các giá trị truyền như tham số cho lớp mẫu. Hình 7.31 minh họa một phụ thuộc <<*bind*>>.



Hình 7.31: Phụ thuộc binding

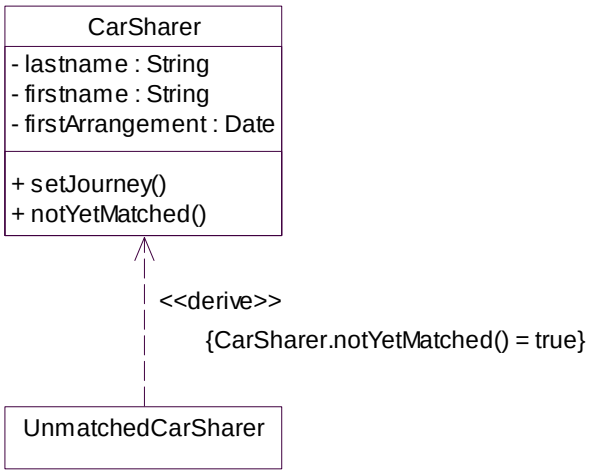
Abstraction: Phụ thuộc *abstraction* dùng để định nghĩa mối quan hệ giữa hai thành phần hoặc một tập các thành phần biểu diễn cùng một khái niệm ở các mức trừu tượng khác nhau. Bao gồm:

<<derive>>: Một phần tử được dẫn xuất (*derive*) là một đặc tả dư thừa trong mô hình vì giá trị của nó được tính từ các phần tử khác. Đặc tả của nó phải được cung cấp kèm theo phụ thuộc. Nó có thể được cài đặt để cải thiện tính hiệu quả hoặc tính rõ ràng của giải pháp. Hình 7.32 minh họa một phụ thuộc dẫn xuất.'

<<realize>>: Phụ thuộc *realize* mô tả quan hệ giữa một phần tử trong mô hình cài đặt với một phần tử trong mô hình đặc tả.

<<refine>>: Phụ thuộc *refine* mô tả quan hệ giữa các phần tử ở các mức trừu tượng khác nhau, ví dụ quan hệ giữa mức thiết kế và mức cài đặt.

<<trace>>: Phụ thuộc *trace* mô tả quan hệ giữa hai phần tử biểu diễn cùng một khái niệm nhưng trong các mô hình khác nhau.



Hình 7.32: Phụ thuộc dẫn xuất

Usage: Phụ thuộc *usage* mô tả một yêu cầu của một phần tử với sự có mặt của một phần tử khác. Bao gồm:

<<call>>: Phụ thuộc *call* đặc tả một cộng tác giữa hai thao tác, thao tác này *gọi* thao tác kia.

<<create>>: Phụ thuộc *create* đặc tả việc tạo thể hiện của lớp phụ thuộc là kết quả của việc tạo thể hiện của lớp bị phụ thuộc.

<<instantiate>>: Phụ thuộc *instantiate* đặc tả các thao tác trên lớp phụ thuộc sẽ tạo các thể hiện của lớp bị phụ thuộc.

<<send>>: Phụ thuộc *send* là phụ thuộc giữa một thao tác và một tín hiệu, cho biết thao tác gởi tín hiệu.

Permission: Phụ thuộc *permission* mô tả khả năng truy xuất của một phần tử trong một namespace (xem mục 4.4.1) với một phần tử trong một namespace khác. Bao gồm:

<<access>>: Phụ thuộc *access* mô tả gói phụ thuộc sử dụng các phần tử của gói bị phụ thuộc.

<<import>>: Phụ thuộc *import* mô tả gói phụ thuộc sát nhập các phần tử từ gói bị phụ thuộc.

<<friend>>: Phụ thuộc *friend* thay thế các ràng buộc về tính khả kiến chuẩn, cho phép phần tử phụ thuộc truy xuất phần tử bị phụ thuộc bất chấp tính khả kiến đã đặc tả.

7.7.3 Các đặc trưng phạm vi lớp

Chúng có ích trong quá trình theo dõi trạng thái qua một nhóm các lớp. Ví dụ trong hình 7.10 dùng thuộc tính *nextIDNumber* như một biến đếm. Khi các thể hiện mới được tạo, chúng được gán giá trị hiện tại của *nextIDNumber*. Sau đó giá trị của *nextIDNumber* được tăng lên, sẵn sàng gán cho thể hiện kế tiếp. Các thuộc tính phạm vi lớp có thể được dùng để kiểm soát trạng thái dựa trên hành vi, sẽ thảo luận chi tiết ở chương 11.

Các thao tác phạm vi lớp thường thao tác trên các thuộc tính phạm vi lớp hoặc xử lý các thể hiện của lớp. Ví dụ, *setIDNumber* trong hình 7.10 có trách nhiệm tăng biến đếm *nextIDNumber*. Một ví dụ khác, một phương thức *constructor* khởi tạo các thể hiện của lớp.

7.7.4 Các kiểu lớp

Dùng stereotype không là mối quan tâm chính của các nhà phân tích chưa có kinh nghiệm. Khi họ am hiểu hơn về mô hình đối tượng, về các mẫu và các kỹ thuật đối tượng thì stereotype có thể được đưa như một phương tiện làm giàu thêm nội dung thông tin của biểu đồ lớp.

Stereotype cung cấp một đặc tả ngắn có ích đối với các phần tử. Hai cách dùng stereotype cơ bản là cho lớp và cho phụ thuộc. Mục 7.7.2 đã khảo sát với phụ thuộc. Với lớp, khi đặc tả stereotype nhà phân tích đã xác định vai trò tự nhiên của lớp trong toàn bộ mô hình. Ví dụ đặc tả <<*control*>> chỉ rõ mục tiêu của lớp là kiểm soát các tương tác giữa các lớp khác.

Các lớp tham số và các phần tử bị buộc đều được các nhà phân tích có kinh nghiệm quan tâm nhiều. Ký hiệu này đã được xem xét trong cuốn sách này chủ yếu vì mục đích đầy đủ. Fowler & Scott (1997) lưu ý đến việc tái sử dụng code của lớp mẫu. Tuy nhiên, bất tiện của việc phình lên

các đoạn mã và việc gia tăng độ phức tạp có thể làm giảm tính tiện ích này.

Đặc tả UML (OMG, 1999a) có hai *stereotype* hỗ trợ trực tiếp đến sự mở rộng của chính UML. Chúng là <<*metaclass*>> và <<*powertype*>>. Một lớp *metaclass* có các thể hiện là các class. Một *powertype* là một phần tử meta có các thể hiện là các class trong một mô hình. Cả hai đều ít liên quan đến công việc phát triển mà chỉ được quan tâm bởi các nhà thực hành ở mức tinh thông UML; cho nên chúng vượt quá phạm vi của cuốn sách này.

Câu hỏi ôn tập

7.1 Hãy thảo luận tình huống mà ở đó các biểu đồ đối tượng có thể có ích cho qui trình phân tích.

7.2 Giải thích sự khác biệt giữa quan hệ *instanceOf* và quan hệ liên kết.

7.3 Trên một biểu đồ đối tượng, chú thích kết hợp nào được phép và chú thích kết hợp nào không được phép đối với các liên kết.

7.4 Trên biểu đồ đối tượng, các quan hệ *become* và *copy* dùng để làm gì?

7.5 Ký hiệu phụ thuộc *realize* được dẫn xuất từ hai ký hiệu UML khác. Chúng là những ký hiệu nào?

7.6 Trên một biểu đồ lớp, làm thế nào để phân biệt đặc trưng phạm vi lớp với đặc trưng phạm vi thể hiện?

7.7 Sự khác biệt về ngữ nghĩa giữa đặc trưng phạm vi lớp và đặc trưng phạm vi thể hiện là gì?

7.8 Ba ký hiệu chung được trình bày trong chương này để đặc tả các stereotype trong UML là gì?

7.9 Hãy trình bày mục đích của lớp *enumeration*.

7.10 Hãy trình bày mục đích của lớp *utility*.

7.11 UML hỗ trợ hai ký hiệu để đặc tả một lớp interface. Đó là hai ký hiệu nào? Sự khác biệt chính giữa hai ký hiệu này là gì?

7.12 Lớp *implementationClass* khác với lớp *type* ở chỗ nào ?

7.13 Ký hiệu nào được dùng để liên kết lớp *implementationClass* với lớp *type* của nó?

7.14 Ký hiệu cơ bản cho một lớp *parameterized* (lớp mẫu) là gì?

7.15 Hai ký hiệu để biểu diễn các phần tử bị buộc là gì? Ngữ nghĩa khác biệt chính giữa hai ký hiệu này là gì?

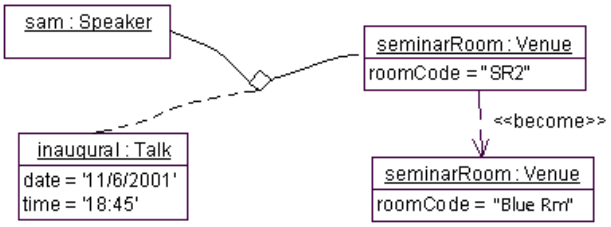
7.16 Giải thích khác biệt chính giữa các kiểu lớp <<*type*>>, {*abstract*} và <<*interface*>>.

Bài tập có lời giải

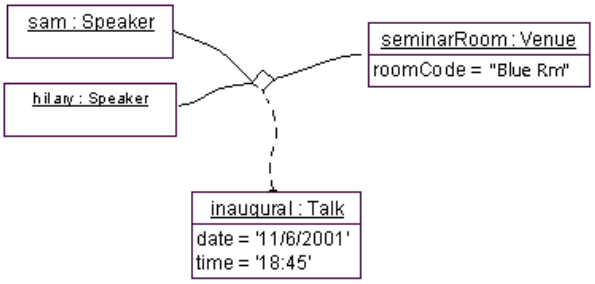
7.1 Hãy vẽ một biểu đồ đối tượng để minh họa cho biểu đồ lớp '*evening talk*' trong hình 6.36 trong bài tập 6.4.

Lời giải:

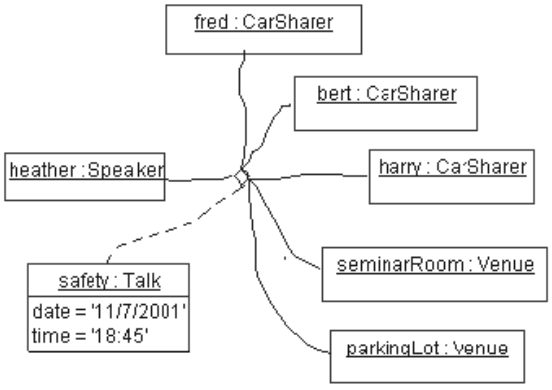
Ba biểu đồ lớp khác nhau được trình bày. Rõ ràng là có nhiều lựa chọn.



Hình 7.33: Một cuộc nói chuyện chưa ai đăng ký



Hình 7.34: Một cuộc nói chuyện với hai diễn giả



Hình 7.35: Một cuộc nói chuyện với ba người đăng ký

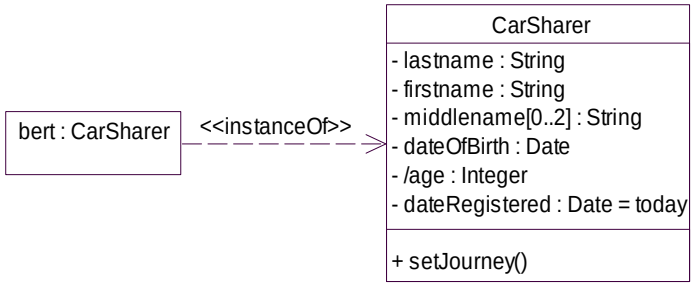
Hình 7.33 trình bày một cuộc nói chuyện (*talk*) được sắp xếp vào một ngày và một giờ cụ thể. Diễn giả (*speaker*) và địa điểm (*venue*) được xác định. Giá trị *roomCode* của địa điểm bị đổi và được minh họa bởi phụ thuộc *become*. Biểu đồ còn cho thấy sự thay đổi *roomCode* phản ánh sự thay đổi trạng thái của đối tượng *SeminarRoom*:*venue*. Tuy vậy, nó vẫn là đối tượng được bao hàm trong mối kết hợp này.

Hình 7.34 minh hoạ một cuộc nói chuyện với hai diễn giả. Cuối cùng, hình 7.35 cho thấy một cuộc nói chuyện với hai địa điểm và nhiều thành viên.

7.2 Với mỗi đối tượng trong bài tập 7.1, hãy vẽ một biểu đồ lớp mô tả phụ thuộc *instanceOf* giữa đối tượng và lớp của nó.

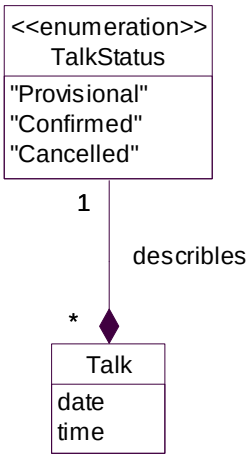
Lời giải:

Hình 7.36 minh họa một phụ thuộc instanceOf.



Hình 7.36: Minh họa phụ thuộc <<instanceOf>>

7.3 Hãy thêm một lớp *enumeration* vào mô hình '*evening talk*' trong bài tập 7.1. Lớp *enumeration* có thể được dùng để gán tình trạng của một *Talk*. Một *talk* có thể là "*Provisional*", "*Confirmed*" hoặc "*Cancelled*".



Hình 7.37: Lớp enumeration *cho trạng thái* evening talk

Lời giải:

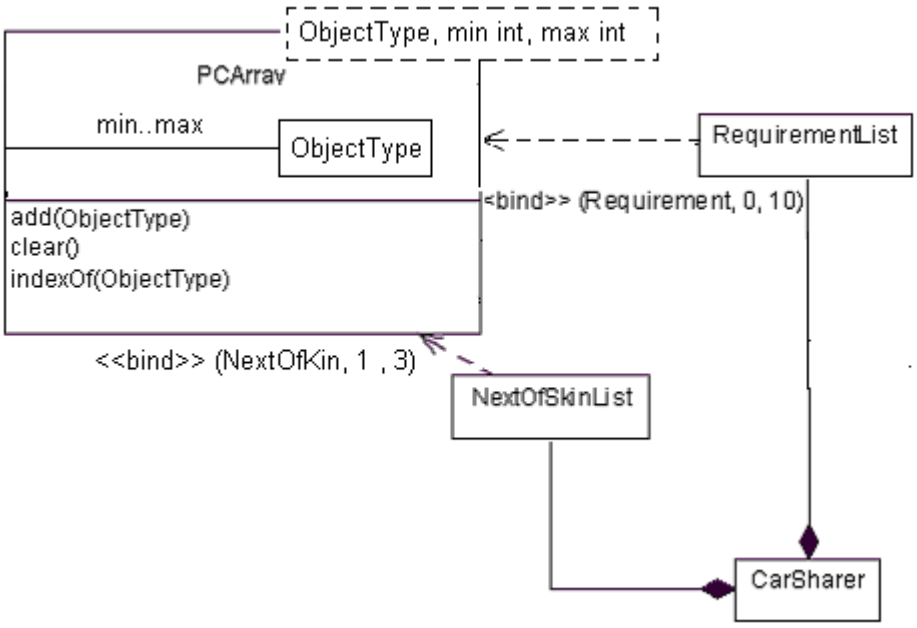
Hình 7.37 trình bày các lớp liên quan.

7.4 Vẽ một biểu đồ lớp biểu diễn các yêu cầu sau với một lớp *parameterized* có hai phần tử bị buộc. Dùng ký hiệu đầy đủ hơn với lớp *parameterized* và các quan hệ *bind*.

> **Janet Hoffner**: Chúng tôi cần phải giữ danh sách các yêu cầu của người chia sẻ xe. Một người chia sẻ xe có thể không có yêu cầu nào, nhưng chúng tôi muốn liệt kê được 10 yêu cầu đối với mỗi người. 10 là một giới hạn tuyệt đối. Chúng tôi cũng cần giữ một danh sách họ hàng của người chia sẻ xe, chúng tôi sẽ giữ tối thiểu là 1 người và tối đa là 3 người.

Lời giải:

Hình 7.38 trình bày một giải pháp. Các chú thích khác đã được lược bỏ.



Hình 7.38: Lớp tham số và các phần tử bị buộc

Bài tập bổ sung

7.5 Hãy vẽ một biểu đồ đối tượng để minh họa cho các lớp, kết hợp n-ngôi và lớp kết hợp trong mô hình lớp mà bạn đã xây dựng trong bài tập 6.6.

7.6 Chỉ ra quan hệ giữa một đối tượng bất kỳ và lớp tương ứng của nó, dùng phụ thuộc <<*instanceOf*>>.

7.7 *VolBank* sẽ dùng *getTimeInterval*(*fromTime*:*Time*, *toTime*:*Time*) và *getDateInterval*(*fromDate*:*Date*, *toDate*:*Date*) trong nhiều thao tác khác nhau trên toàn hệ thống. Ví dụ, để tính thời gian làm việc trên một dự án, thời gian ký gởi, khoảng thời gian đăng ký của nhu cầu, của tình nguyện viên và của tổ chức. Hai thao tác này được mô hình hóa như thế nào để chúng trở thành toàn cục? Hãy mô hình một lớp thích hợp cho thấy các ký hiệu được sử dụng như thế nào.

7.8 Đối với một tổ chức, *VolBank* muốn lưu giữ một danh sách địa chỉ liên lạc hơn là một địa chỉ duy nhất. Hãy mô hình hóa điều này dùng cả hai loại ký hiệu, rút gọn và đầy đủ, cho các lớp tham số và các thành phần bị buộc.

Chương 8
BIỂU ĐỒ CỘNG TÁC

8.1 Giới thiệu

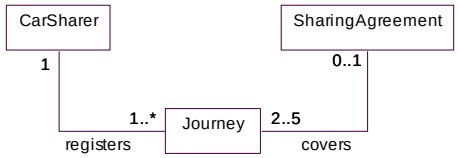
Trong chương 3, chúng ta đã nói use case được *hiện thực* bởi cộng tác. Trong chương này chúng ta sẽ tìm hiểu xem cộng tác là gì và cách đưa ra một biểu đồ cộng tác.

8.2 Cộng tác là gì?

Trong các hệ thống hướng đối tượng, các đối tượng phối hợp hoạt động với nhau tạo ra chức năng mà người sử dụng yêu cầu. Mỗi đối tượng độc lập chỉ cung cấp một phần nhỏ của chức năng, khi phối hợp hoạt động sẽ tạo ra chức năng ở mức cao mà con người có thể sử dụng. Để phối hợp hoạt động, các đối tượng cần giao tiếp với nhau bằng cách gởi các thông điệp. Hoạt động phối hợp như vậy, nhằm đưa ra một kết quả hữu ích nào đó, được gọi là *cộng tác* (collaboration).

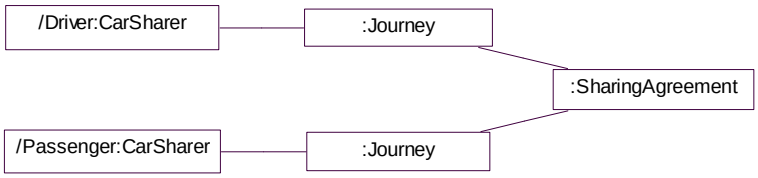
Trong đặc tả UML (OMG, 1999a) cộng tác được mô tả như một cái gì đó nhằm '*định nghĩa các thành phần và quan hệ có ý nghĩa đối với các mục đích đề ra*'. Để dễ hiểu, chúng ta hãy khảo sát một ví dụ, sau đó quay lại định nghĩa này.

Chúng ta muốn xác định các thành phần trong use case *Record sharing agreement* (ghi nhận thoả thuận chia sẻ). Khi có một thoả thuận đạt được giữa hai hoặc nhiều thành viên cùng chia sẻ một lộ trình, thì một đối tượng *SharingAgreement* mới được tạo ra. Đối tượng này có một kết hợp với *Journey* hơn là với *CarSharer*, vì mỗi thành viên có thể có vài lộ trình cần chia sẻ với các thành viên khác. Use case này bao gồm tối thiểu 5 đối tượng như trong hình 8.1 (hai thể hiện của *CarSharer*, hai thể hiện của *Journey*, và một thể hiện của *SharingAgreement*).



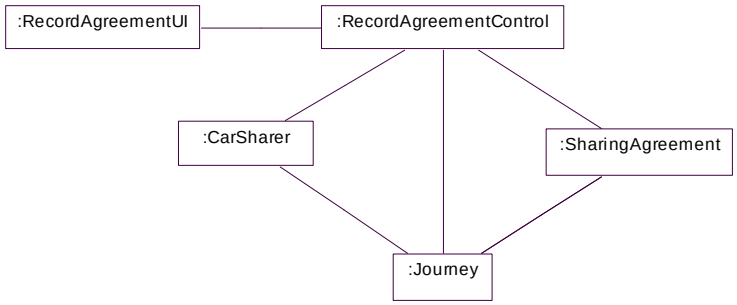
Hình 8.1: Các lớp trong use case Record Sharing agreement

Thực ra, trong thuật ngữ UML chúng ta đang làm việc với vai trò hơn là với lớp hoặc với đối tượng (xem mục 14.2.13, chương 14). Chúng ta có thể dùng tên vai trò trong các cộng tác. Nếu mỗi thành viên là tài xế hoặc hành khách, ta có thể tạo ra hai vai trò cho các đối tượng *CarSharer* trong cộng tác này, như trong hình 8.2. Tuy nhiên, hệ thống này khuyến khích mọi người thay nhau lái xe nếu được, nên không cần đến các vai trò */Driver:CarSharer* và */Passenger:CarSharer*.



Hình 8.2: Các vai trò trong cộng tác Record Sharing agreement

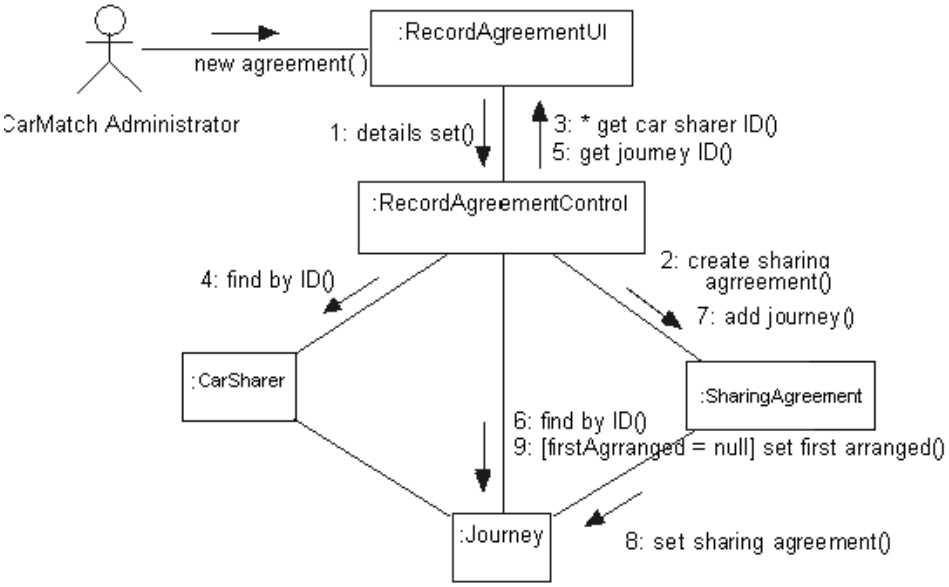
Các liên kết giữa các vai trò trong hình 8.2 tương ứng với kết hợp giữa *CarSharer* và *SharingAgreement* trong hình 8.1. Use case này có bổ sung một vài lớp, hình 8.3. Ở giai đoạn này, chúng ta cố giữ cho cộng tác đơn giản: chúng ta muốn có một ý niệm về các lớp *chính* mà không quan tâm đến cách cài đặt chi tiết. Ví dụ, ta không thêm các lớp *collection* (tập hợp) hoặc các lớp quản lý *persistence* (tồn tại lâu dài). Vì không có sự phân biệt vai trò (role) giữa các đối tượng *CarSharer*, ta biểu diễn như hình 8.3.



Hình 8.3: Các lớp trong cộng tác này

Các lớp được bổ sung ở đây là lớp interface *RecordAgreementUI* và lớp control *RecordAgreementControl*. Các vai trò kết hợp và các *multiplicity* có thể được thêm vào, nếu cần. Các chi tiết của tương tác (interaction) cũng có thể được thêm vào. Theo định nghĩa UML, '*Tương tác đặc tả giao tiếp giữa các thể hiện đang thực hiện một tác vụ nào đó … trong ngữ cảnh của một cộng tác*'. Trong các giai đoạn đầu của dự án, các thông

điệp tạo ra giao tiếp của tương tác có thể được định nghĩa một cách không hình thức. Sau này, chúng sẽ là các tín hiệu hoặc các thao tác được gởi giữa các thể hiện đối tượng. Hình 8.4 chỉ ra một vài thông điệp trên biểu đồ cộng tác của hình 8.3, kể cả tác nhân gây ra tương tác.



Hình 8.4: Tương tác được thêm vào cộng tác của hình 8.3

Biểu đồ này mô tả các lớp tham gia vào việc hiện thực use case *Record sharing agreement* và các mối quan hệ có ý nghĩa giữa chúng đối với việc tạo ra một thoả thuận dùng chung xe. Các quan hệ với lớp *RecordAgreementControl* ở đây sẽ không có ý nghĩa trong ngữ cảnh của use case khác, như use case *Register car sharer* chẳng hạn. Ngoài ra quan hệ giữa *CarSharer* và *SharingAgreement* cũng sẽ không thích hợp đối với các use case khác. Như vậy cộng tác được mô tả như là một cái gì đó '*định nghĩa các thành phần và quan hệ có ý nghĩa đối với các mục đích đề ra*'.

8.3 Mục đích của kỹ thuật

Biểu đồ cộng tác được dùng trong quá trình phác thảo tỉ mỉ biểu đồ lớp nhằm giúp người phân tích hiểu được các nhóm đối tượng tham gia hiện thực một use case. Biểu đồ cộng tác được sử dụng khi biểu đồ lớp không diễn đạt được hết ý nghĩa tương tác giữa các đối tượng. Ngoài ra, biểu đồ cộng tác còn được dùng để xác định các đối tượng có liên quan trong các

thao tác. Luôn nhớ rằng trong nhiều dự án hướng đối tượng, quá trình phát triển hệ thống là một quá trình lặp. Việc mô hình hoá các cộng tác và tương tác bằng cách dùng biểu đồ cộng tác hoặc biểu đồ tuần tự có thể cần đến các lớp mới, các thuộc tính và thao tác mới. Chúng được thêm vào biểu đồ lớp. Điều này dẫn đến việc thay đổi các biểu đồ tuần tự và biểu đồ cộng tác đã có hoặc bổ sung các biểu đồ mới để mô hình các thao tác mới. Cuối cùng, một giới hạn phải được đưa ra để kết thúc quá trình lặp này; ở đâu và bằng cách nào là do người quản lý dự án hoặc chính qui trình phát triển xác định. Trong cuốn sách về lập trình *Extreme Programming*, *Kent Beck 2000* tác giả chứng tỏ rằng hệ thống nên được mang đi *sản xuất* ngay khi có thể, thậm chí mới được phát triển từng phần.

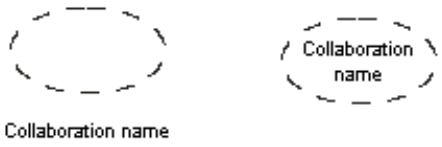
Các mục đích chính của việc tạo ra biểu đồ cộng tác bao gồm:

- Mô hình cộng tác giữa các đối tượng hoặc các vai trò nhằm thực hiện chức năng của một use case.
- Mô hình cộng tác giữa các đối tượng hoặc các vai trò nhằm thực hiện chức năng của một thao tác.
- Mô hình các cơ chế bên trong *thiết kế kiến trúc* (architectural design).
- Biểu đồ cộng tác được chú thích với các tương tác. Các tương tác này trình bày các thông điệp giữa các đối tượng hoặc vai trò bên trong cộng tác.
- Biểu đồ cộng tác được dùng để mô hình các *scenario* bên trong một use case hoặc một thao tác liên quan đến sự cộng tác của các đối tượng khác nhau và các tương tác khác nhau.
- Biểu đồ cộng tác được dùng trong các giai đoạn đầu của dự án để xác định các đối tượng (do đó các lớp) tham gia vào một use case. Mỗi cộng tác biểu diễn một góc nhìn của biểu đồ lớp, và được tổ hợp trong mô hình của toàn bộ hệ thống.
- Biểu đồ cộng tác đươc dùng để mô tả các đối tượng tham gia trong một mẫu thiết kế (xem chương 16).

8.4 Ký hiệu của cộng tác

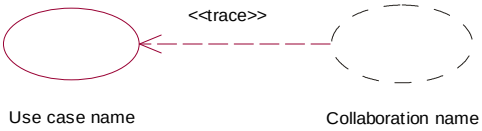
Trước khi thảo luận ký hiệu của biểu đồ cộng tác, chúng ta tìm hiểu ký hiệu của cộng tác mặc dù ký hiệu này ít được sử dụng.

Cộng tác có thể được biểu diễn bằng một hình oval đứt nét với tên của cộng tác được đặt bên trong hoặc bên dưới (xem hình 8.5).



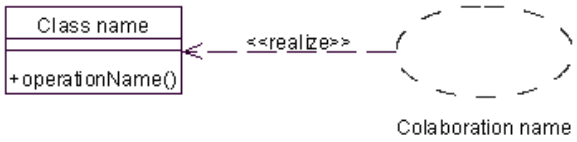
Hình 8.5: Ký hiệu cộng tác

Cộng tác cũng được dùng để biểu diễn mối quan hệ giữa cộng tác và use case mà chúng hiện thực, xem hình 8.6.



Hình 8.6: Quan hệ cộng tác «trace»

Tiếp cận này cũng được dùng để mô tả mối quan hệ giữa một cộng tác và một thao tác, như trình bày trong hình 8.7.

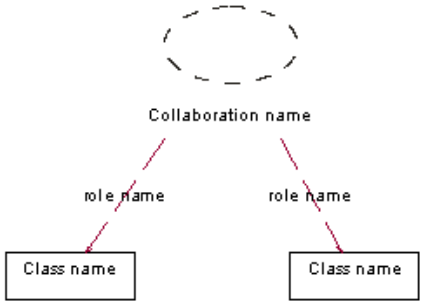


Hình 8.7: Quan hệ cộng tác <<realize>>

Tuy nhiên, những quan hệ này không thường được biểu diễn tường minh trong biểu đồ ngoại trừ các liên kết giữa các biểu đồ trong một công cụ CASE. Đặc tả UML (OMG, 1999a) xem chúng như là các *siêu liên kết ẩn*. Ta cũng có thể biểu diễn một cộng tác với tên của thao tác được hiện thực:

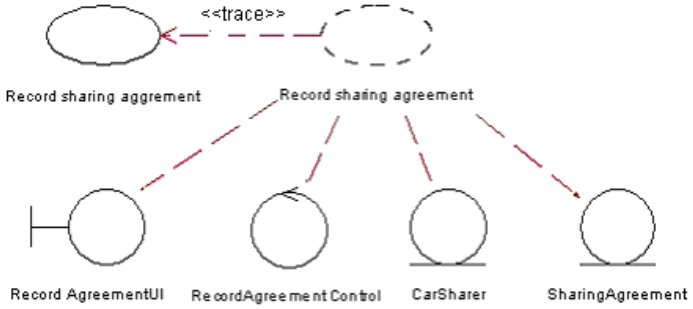
Hình 8.8: Cộng tác hiện thực thao tác

Ký hiệu hình oval đứt nét được dùng để mô tả một cộng tác qua các đối tượng tham gia. Trong trường hợp này, cộng tác được liên kết đến các lớp của chúng bằng các đường đứt nét có nhãn là tên các vai trò (hình 8.9).



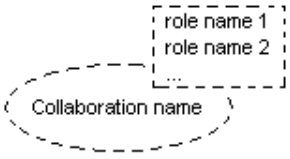
Hình 8.9: Các vai trò cộng tác

Ký hiệu này có thể được dùng với các lớp *khuôn dạng* và phụ thuộc «*trace*» để chỉ rõ các lớp tham gia hiện thực một use case, xem hình 8.10.



Hình 8.10: Các lớp hiện thực use case Record Sharing Argeement

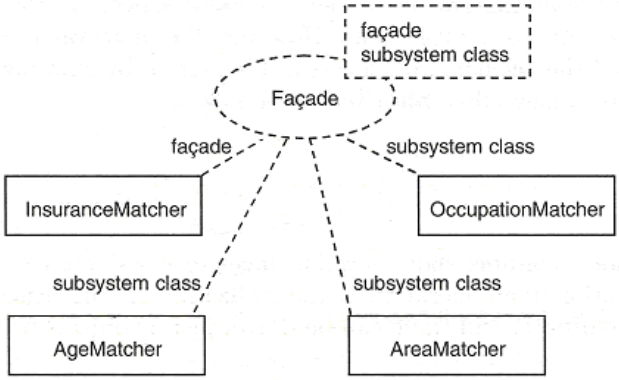
Một mẫu (pattern) thiết kế cũng có thể được biểu diễn theo cách này (xem chương 16). Khi ấy, tên vai trò đươc biểu diễn bằng cách dùng ký hiệu *tham số* (parameterized) hoặc *mẫu* (template), như trong hình 8.11.



Hình 8.11: Cộng tác mẫu *(template/ parameterized)*

Khi mẫu được dùng, các *lớp thực* phải bị buộc vào các vai trò này. Ví dụ, mẫu *Façade* được sử dụng rộng rãi trong các hệ thống mà người phát triển muốn che dấu sự phức tạp của các hệ thống con đằng sau một lớp

duy nhất cung cấp một giao diện đến các chức năng của chúng. Hình 8.12 trình bày mẫu này với vai trò *façade* buộc vào lớp *InsuranceMatcher* và vai trò *subsystem class* buộc vào các lớp *AgeMatcher*, *AreaMatcher* và *OccupationMatcher*.



Hình 8.12: Ví dụ dùng cộng tác để sưu liệu cho cách dùng mẫu Façade

8.5 Ký hiệu của biểu đồ cộng tác

Trong khi các cộng tác hiếm khi được sử dụng thì các biểu đồ cộng tác được sử dụng một cách rộng rãi. Thường được sử dụng nhất là một tương tác được đặt trên biểu đồ cộng tác để biểu diễn chi tiết giao tiếp giữa các thể hiện đối tượng tham gia vào tương tác.

Các cộng tác có thể được mô hình ở hai mức: *đặc tả* và *thể hiện*. Ở mức đặc tả, biểu đồ sẽ hiển thị các vai trò *classifier* (xem chương 14), các vai trò kết hợp và các thông điệp. Ở mức thể hiện, biểu đồ sẽ biểu diễn các đối tượng, các liên kết và các tác nhân kích thích. Lớp là loại *classifier* thường được mô hình nhất trong biểu đồ cộng tác.

8.5.1 Biểu đồ cộng tác mức đặc tả

Biểu đồ cộng tác mức đặc tả dùng để biểu diễn trường hợp tổng quát của một cộng tác. Bao gồm cả các *nhánh*, các *vòng lặp* và các *vai trò lớp*.

8.5.2 Biểu đồ cộng tác mức thể hiện

Biểu đồ cộng tác mức thể hiện được dùng để biểu diễn một thể hiện cụ thể của một tương tác đang diễn ra và các thể hiện đối tượng liên quan.

Sự khác biệt giữa biểu đồ cộng tác mức đặc tả và mức thể hiện giống với sự khác biệt giữa sự mô tả tổng quát của một use case và mô tả các *scenario,* là các thể hiện riêng của use case.

8.5.3 Biểu đồ cộng tác làm ngữ cảnh của một tương tác

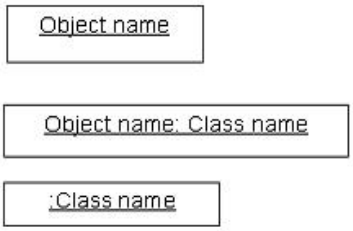
Biểu đồ cộng tác cũng được phác thảo mà không có tương tác nhằm biểu diễn ngữ cảnh của một tương tác, là tập con của mô hình lớp hoặc mô hình đối tượng bao hàm trong tương tác.

Khi được sử dụng theo cách này, biểu đồ cộng tác mô tả *khía cạnh cấu trúc* của cộng tác – các vai trò lớp liên quan và các kết hợp giữa chúng. Tuy nhiên, giá trị chính của biểu đồ là để mô hình *khía cạnh hành vi* của hệ thống bằng cách chỉ ra tương tác giữa các vai trò lớp nhằm đạt được các mục tiêu của hệ thống.

8.5.4 Thể hiện đối tượng

Biểu đồ cộng tác mức thể hiện mô tả cách các thể hiện cộng tác với nhau để đạt được một vài mục tiêu – như cài đặt một thao tác hoặc hiện thực một use case. Các thể hiện thông dụng nhất là các đối tượng.

Các đối tượng tham gia vào cộng tác được mô tả bằng ký hiệu của thể hiện (chương 4). Chúng là các thể hiện của lớp, có hoặc không có tên, có hoặc không có vai trò. Điều quan trọng cần nhớ là các thể hiện luôn luôn được gạch chân, trong khi tên lớp và tên vai trò thì không (hình 8.13).



Hình 8.13: Các thể hiện đối tượng

Bảng 8.1 trình bày các dạng khác nhau mà tên thể hiện có thể nhận.

Cú pháp	Giải thích
o	Một đối tượng tên o.
o:C	Đối tượng o của lớp C.
:C	Đối tượng vô danh của lớp C.
/R	Đối tượng vô danh đóng vai trò R.
/R:C	Đối tượng vô danh của lớp C đóng vai trò R.
o/R	Đối tượng o đóng vai trò R.
o/R:C	Đối tượng o của lớp C đóng vai trò R.

Bảng 8.1: Cú pháp của tên thể hiện

8.5.5 Lớp và vai trò

Thay cho các thể hiện đối tượng, các vai trò lớp có thể được dùng trong biểu đồ cộng tác. Trong một biểu đồ cộng tác, các lớp tham gia vào một cộng tác có thể được mô tả bằng ký hiệu lớp. Tên của lớp được đặt sau dấu hai chấm cho biết vai trò lớp đang được mô hình ở đây. Nếu vai trò có tên thì thêm dấu chéo thuận (/) vào trước tên và đặt trước dấu hai chấm, như trong hình 8.14.

Hình 8.14: Tên vai trò

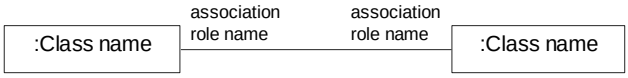
Bảng 8.2 trình bày các dạng khác nhau mà các tên vai trò có thể nhận.

Cú pháp	Giải thích
/R	Một vai trò tên R.
:C	Một vai trò không tên với lớp cơ sở C.
/R:C	Một vai trò tên R với lớp cơ sở C.

Bảng 8.2: Cú pháp của tên vai trò

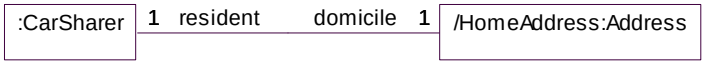
8.5.6 Kết hợp

Các kết hợp giữa các lớp trong biểu đồ lớp được thêm vào biểu đồ cộng tác để hỗ trợ cho cộng tác. Trong UML, chúng được gọi là các *vai trò kết hợp* (association role). Nếu đặt nhãn, thì tên của các vai trò lớp trong kết hợp này đươc thêm vào (xem hình 8.15), multiplicity của các kết hợp cũng có thể được mô tả.



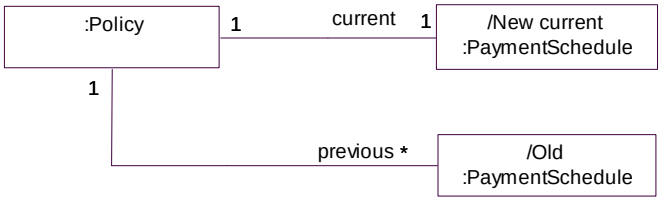
Hình 8.15: Ký hiệu vai trò kết hợp

Hình 8.16 minh họa một ví dụ về vai trò kết hợp đối với *CarSharer* và địa chỉ nhà của họ.



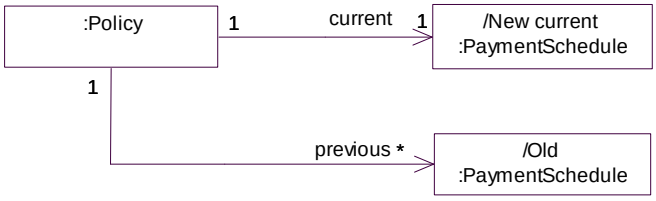
Hình 8.16: Ví dụ dùng vai trò kết hợp *trong biểu đồ cộng tác*

Khi thêm vai trò vào, biểu đồ trở nên bừa bộn. Tốt nhất chỉ nên sử dụng chúng khi cần phân biệt các vai trò trong nhiều kết hợp khác nhau; khi cùng một thể hiện tham gia vào cùng một cộng tác với các vai trò khác nhau, như trong hình 8.17.



Hình 8.17: Ví dụ cần thiết dùng vai trò kết hợp trên biểu đồ cộng tác

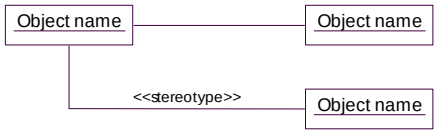
Cũng có thể biểu diễn khả năng điều hướng của các kết hợp, như trong hình 8.18. Hình này xác định rằng vai trò :*Policy* cần có khả năng gởi thông điệp đến các vai trò :*PaymentSchedule*.



Hình 8.18: Ví dụ về khả năng điều hướng
của các vai trò kết hợp trên một biểu đồ cộng tác

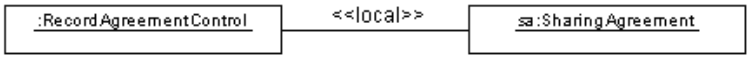
8.5.7 Liên kết (link)

Có thể thêm liên kết giữa các thể hiện đối tượng vào biểu đồ cộng tác. Liên kết có thể là các thể hiện của các kết hợp có trong biểu đồ lớp, hoặc là các liên kết tạm thời giữa chúng để gởi thông điệp cho nhau. Ví dụ, một *đối tượng điều khiển* tạo ra một thể hiện của một lớp nào đó như một phần của cộng tác, nhưng liên kết giữa chúng chỉ là tạm thời (không được hiển thị trong biểu đồ lớp), nó cho biết có một thể hiện đối tượng được giữ trong một biến cục bộ của *đối tượng điều khiển*. Các liên kết có thể được tạo khuôn dạng là «*local*» hoặc «*parameter*», xem hình 8.19.

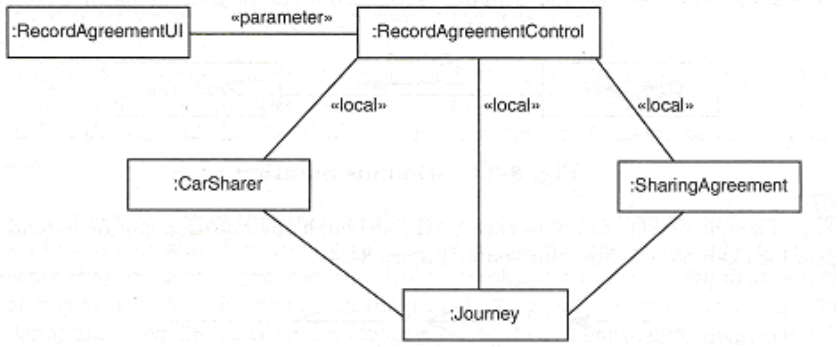


Hình 8.19: Liên kết với khuôn dạng

Hình 8.20 minh họa cách dùng «*local*» để chỉ thể hiện mới của *SharingAgreement* do thể hiện *RecordAgreementControl* tạo ra. Thể hiện mới này được tạo như một biến cục bộ và liên kết xuất hiện chỉ là tạm thời – liên kết sẽ bị mất khi thể hiện *RecordAgreementControl* bị huỷ khi kết thúc chương trình, hoặc khi tạo mới một thể hiện khác của *SharingAgreement* (trong trường hợp này là *sa*).



Hình 8.20: Liên kết «local» trên một biểu đồ cộng tác



Hình 8.21: Các liên kết trên một biểu đồ cộng tác

Chúng ta hãy so sánh liên kết này với các liên kết trong hình 8.21. Các liên kết giữa thể hiện mới của *SharingAgreement* và thể hiện của *Journey* là các liên kết lâu dài – các liên kết này phải được quản lý bằng cách dùng một loại cơ chế lưu trữ lâu dài (các tập tin hoặc một cơ sở dữ liệu) sau khi chương trình kết thúc. Các liên kết này là các thể hiện của kết hợp *covers*, mô tả trong biểu đồ lớp ở hình 8.1. Liên kết giữa thể hiện của *RecordAgreementControl* và thể hiện của giao diện người dùng, *RecordAgreementUI* là «*parameter*». Trong trường hợp này, đối tượng điều khiển sẽ tạo ra một thể hiện của lớp giao diện người dùng và truyền cho thể hiện mới một số loại tham chiếu đến chính nó như một tham số (ví dụ, một con trỏ trong C++, một tham chiếu trong Java hoặc một tên trong Smalltalk) để giao diện người dùng gởi các biến cố ngược trở lại đối tượng điều khiển.

8.5.8 Thông điệp

Thêm các kết hợp và liên kết vào biểu đồ cộng tác để chỉ rõ đường đi của các thông điệp được gởi từ thể hiện này đến thể hiện khác. Có thể lấy tên của thông điệp làm nhãn cho liên kết. Trong các phiên bản UML

trước, thông điệp là thuật ngữ duy nhất được sử dụng. Từ phiên bản 1.3, có sự phân biệt giữa *thông điệp* (message) trong biểu đồ mức đặc tả và *stimulus* (tác nhân kích thích) trong biểu đồ mức thể hiện. Thông điệp là đặc tả của *stimulus*, và *stimulus* biểu diễn một thể hiện nào đó của việc gởi thông điệp với các tham số cụ thể. Thông điệp trong biểu đồ cộng tác mức đặc tả được thêm vào ở giai đoạn đầu trong qui trình phát triển hệ thống, trước khi tên của các thao tác và các biến cố được quyết định và được sưu liệu trong mô hình.

Một *stimulus* được định nghĩa như '*một sự giao tiếp giữa hai đối tượng nhằm truyền tải thông tin với kỳ vọng một tác động sẽ xảy ra*' (OMG, 1999a). Một stimulus có thể là:

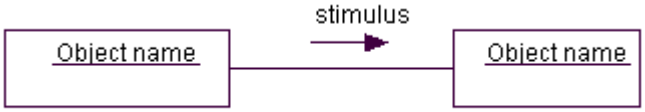
- Một biến cố được gởi từ một đối tượng đến một đối tượng khác, ví dụ một hành động nhấn nút trong đối tượng giao diện người dùng được gởi đến đối tượng điều khiển.
- Một thao tác được gọi, ví dụ một đối tượng đang gọi thao tác *getName*() của một đối tượng *CarSharer* để truyền tên cần hiển thị đến giao diện người dùng.
- Việc tạo hoặc huỷ một thể hiện đối tượng, ví dụ một đối tượng điều khiển tạo ra một thể hiện của *Sharing Agreement*.

Thuật ngữ *message* (thông điệp) được sử dụng rộng rãi và dễ hiểu hơn *stimulus*, và chúng ta sẽ ám chỉ đến các thông điệp thông qua mô tả cách hoạt động của nó dưới đây.

Thông điệp được biểu diễn bằng một mũi tên chỉ hướng gởi, và một nhãn xác định (xem hình 8.22).

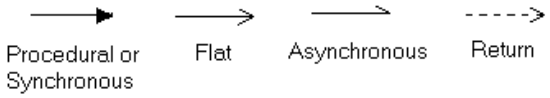
Hình 8.22: Ký hiệu của thông điệp

Stimulus dùng ký hiệu tương tự như thông điệp, nhưng stimulus được gởi từ đối tượng đến đối tượng (xem hình 8.23).



Hình 8.23: Ký hiệu của stimulus

Trong UML cơ sở (core UML), mũi tên thuộc một trong bốn dạng (xem hình 8.24). Tuy nhiên, ta cũng có thể mở rộng ký hiệu cho mũi tên bằng cách dùng stereotype.



Hình 8.24: Ký hiệu thông điệp

- **Procedural or Synchronous** (thủ tục hoặc đồng bộ)**:** Một thông điệp được gởi bởi một đối tượng đến một đối tượng khác và đối tượng đầu sẽ đợi cho đến khi hành động hoàn tất. Kể cả việc đợi hoàn tất các hành động do đối tượng thứ hai cầu viện đến các đối tượng khác nữa để hoàn tất hành động này.
- **Flat** (phẳng)**:** Mỗi mũi tên biểu diễn từng bước của một tiến trình. Bình thường thông điệp là không đồng bộ. Một thông điệp *flat* có thể được dùng khi không biết rằng thông điệp thực sự là có đồng bộ hay không.
- **Asynchronous** (không đồng bộ)**:** Một thông điệp được gởi bởi một đối tượng đến đối tượng khác nhưng đối tượng đầu tiên không đợi hành động hoàn tất mà sẽ thực hiện bước kế tiếp trong chuỗi hành động của nó.
- **Return** (trả về)**:** Biểu diễn việc trả điều khiển tường minh về đối tượng gởi thông điệp. Return không thường được biểu diễn trên biểu đồ cộng tác nhưng lại thường được biểu diễn trên các biểu đồ tuần tự (xem chương 9).

Mũi tên biểu diễn dòng điều khiển. Nếu dòng là *Procedural or Synchronous*, thì chỉ có một *tiểu trình* (thread) thi hành, và hoạt động được truyền từ một đối tượng đến đối tượng khác. Nếu dòng là *Asynchronous*, thì nhiều đối tượng có thể hoạt động tại một thời điểm.

Chi tiết của thông điệp kèm theo mũi tên rất đáng quan tâm. Ở mức đơn giản nhất chỉ là tên và thường kèm theo thứ tự thực hiện, như trong hình 8.25.

Hình 8.25: Thông điệp trên biểu đồ cộng tác

Phức tạp nhất, thông điệp có cú pháp như sau:

predecessor guard-condition sequence-expression return-value := message-name argument-list

Tất cả những thành phần này có thể bỏ đi, ngoại trừ *message-name* (tên thông điệp). *Return-value*, *message-name* và *argurment-list* (giá trị trả về, tên thông điệp, danh sách đối số) được gọi là *signature* (chữ ký) của thông điệp.

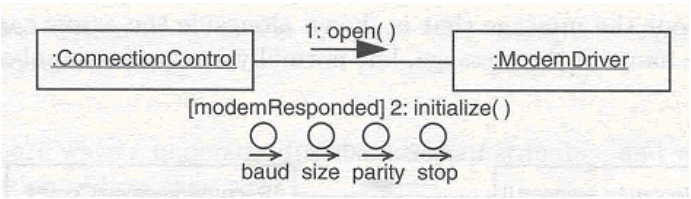
- **Message-name**: Tên thông điệp là biến cố được gởi đến đối tượng đích. Đây có thể là tên của một biến cố hoặc của một thao tác trong lớp của đối tượng đích. Ví dụ *SetFirstArranged* được gởi đến thể hiện của *Journey*.

  Nếu một biến cố được sử dụng, thì đối tượng nhận biến cố sẽ thực hiện các thao tác cần thiết, tên của biến cố và tên của thao tác không giống nhau. Thuật ngữ *signal* (tín hiệu) được dùng để ám chỉ đến các biến cố được gởi một cách không đồng bộ từ đối tượng này đến một đối tượng khác. Một *signal* có thể có các thuộc tính. Ví dụ, tín hiệu *MouseDown* có thể có các thuộc tính như *button, xyCoordinate* và *timestamp* (nút, tọa độ, thời nhãn). Các *signal* có thể được tổ chức trong một cây phân cấp giống với các lớp.

- **Argument-list**: Thao tác được xác định duy nhất nếu đặc tả đầy đủ danh sách đối số, như trong trường hợp nạp chồng các phiên bản khác nhau có cùng tên nhưng khác danh sách đối số. Các biến cố cũng có thể có các đố số. Danh sách đối số là một danh sách các tên đối số phân cách nhau bởi các dấu phẩy, ví dụ *SetArranged* (*dateArranged*) hoặc *MouseDown* (*button, xyCoordinate, timestamp*). Các đối số thường là các tham chiếu đến các đối tượng hoặc đến các biến chứa các kiểu dữ liệu gốc. Chúng cũng có thể là các biểu thức mã giả hoặc là một ngôn ngữ lập trình, ví dụ *setArranged* (*Date.getCurrentDate*()). Tên đối số có thể là tên thuộc tính của đối tượng gởi, giá trị trả về từ các thao tác trước hoặc là các đường dẫn điều hướng đến đối tượng liên quan, ví dụ *journeyName* := *getJourneyName*(*i*) sau đó là *journey* := *find*(*journeyName*).

- **Return-value**: Biến cố là không đồng bộ và không có giá trị trả về. Các thao tác đồng bộ có thể có giá trị trả về, giá trị này được gởi trở lại cho đối tượng gởi thông điệp, ví dụ *journeyName* := *getJourneyName* (*i*) và *journey* := *find*(*journeyName*). Các giá trị trả về là một danh sách các tên được trả về khi kết thúc thao tác. Nếu một thao tác không trả về một giá trị, thì trị trả về và dấu ':=' được bỏ đi – không cần phải dùng từ *void* như trong một số ngôn ngữ lập trình.

Thay vì dùng văn bản làm các đối số và giá trị trả về, chúng ta có thể dùng các *data token*, chúng là các hình tròn nhỏ có một mũi tên đính kèm và tên của giá trị trả về đi cùng, như trong hình 8.26.



Hình 8.26: Data token được sử dụng như các đối số

- **Sequence-expression** (biểu thức tuần tự): Biểu thức tuần tự định nghĩa thứ tự xảy ra của các tương tác. Nó là danh sách các *sequence-term* (số thứ tự) phân cách bởi dấu chấm, theo sau là dấu hai chấm. Mỗi *term* (số) biểu diễn một mức *lồng* (nest) bên trong tương tác. Nếu một đối tượng nhận một thao tác, được *đánh số* là **1:**, dẫn đến việc gọi hai thao tác khác, thì chúng sẽ có *sequence-expression* là **1.1:** và **1.2:**. Nếu tất cả các điều khiển đều đồng hành, thì sẽ không có sự lồng nhau nào xảy ra và các thông điệp được đánh số tuần tự là **1:**, **2:**, **3:**,...

  Mỗi *sequence-term* có thể chứa một số nguyên, hoặc một tên; và có thể kèm một *recurrence* (số lần lặp). Đặc tả UML không giải thích điều này rõ ràng, nó cho rằng mỗi *term* là một số nguyên hoặc là một tên, nhưng lại không định nghĩa *tên* là gì, xem thảo luận trong mục 14.2.7. Ví dụ cho *sequence-expression* đối với thông điệp kết hợp với tên là **3.1a** và **A1**. Trong Booch và các cộng sự 1999, tên đối tượng được dùng làm tiền tố cho số thứ tự trong thông điệp gốc.

Hình 8.27: thêm tên đối tượng trước sequence-term

(Gomaa, 2000) gợi ý dùng các tên khởi tạo để biểu diễn use case đang được hiện thực bởi cộng tác, ví dụ *Use1.1.2*. Ông cũng gợi ý nên dùng các ký tự làm hậu tố để cho biết các thông điệp đồng hành và các thông điệp thay thế ở nơi có sự chọn lựa.

Một tên có thể được dùng thay cho số để chỉ rõ các thao tác cùng mức; nghĩa là các thao tác xảy ra tại cùng một thời điểm. Vì vậy *3.1.1a* và *3.1.1b* là đồng thời trong hoạt động của thông điệp *3.1*.

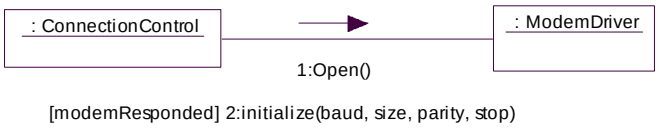
Tên cũng có thể được dùng để phân biệt các thông điệp thay thế là kết quả từ một điều kiện trên thông điệp, ví dụ *3.1.2a*[*x<=0*] và *3.1.2b*[*x>0*].

Recurrence được dùng để biểu thị thông điệp mang tính lặp hoặc phụ thuộc điều kiện. Tính lặp được mô tả bởi một dấu hoa thị (*) và một *iteration-clause* (mệnh đề lặp) được đặt trong cặp dấu ngoặc vuông, ví dụ *[*i*:*0*..*journeys.length*] hoặc *[*while not end-of-file*]. Nếu tên được dùng làm chỉ mục trong một vòng lặp (chẳng hạn *i* trong ví dụ trên), thì tên này được dùng trong *sequence-term* con, ví dụ *2.1.i*, nhưng mệnh đề lặp này không được xuất hiện một lần nữa trong mức này. Nếu các thao tác bên trong vòng lặp sẽ được thực hiện song song (trong thiết bị có cung cấp xử lý song song), thì dấu hoa thị kèm theo hai vạch thẳng đứng, *||.

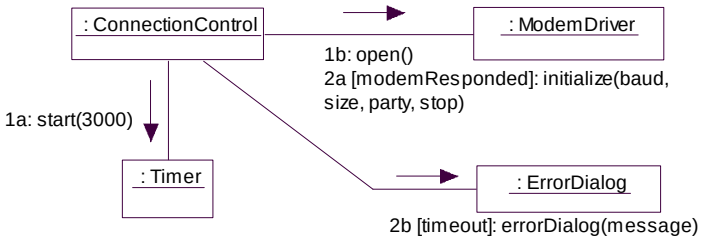
Với điều kiện, biểu diễn chỗ có sự lựa chọn, ta dùng cùng cú pháp nhưng không có dấu hoa thị, ví dụ [*dateArranged=null*].

- **Predecessor** (các thông điệp tiên quyết): của một thông điệp là một danh sách các *sequence-number* (số tuần tự) cách nhau bởi dấu phẩy kết thúc bằng một dấu chéo xuôi. Một *sequence-number* là một *sequence-expression* không có *recurrence* cũng như dấu hai chấm (ví dụ *2.1* thay vì *2.1* *[*while not end-of-file*]:). *Predecessor* liệt kê các *sequence-number* của các thông điệp phải hoàn tất trước khi thông điệp hiện hành được gởi đi, ví dụ *2.1a*,*2.1b*/*2.2*: *drawLine*(*start*, *finish*). Nếu danh sách rỗng, thì bỏ luôn dấu chéo. Theo mục *UML Semantics* (ngữ nghĩa UML) của đặc tả UML, thì các *sequence-number* phải có *activator* (mức hoạt động) giống với thông điệp hiện hành. Tác dụng của việc dùng mệnh đề predecessor là để đồng bộ hoá các tiểu trình thi hành trùng nhau trước khi xử lý thông điệp kế tiếp.
- **Guard-condition** (điều kiện bảo vệ): Một *guard-condition* là một điều kiện phải thoả trước khi thông điệp được gởi. Trong mục *ký hiệu UML* ta đã không định nghĩa *guard-condition*, nhưng nó sẽ được định nghĩa trong ngữ cảnh của *biểu đồ trạng thái* (chương 11). Một *guard-condition* là một biểu thức logic thường được viết bằng *ngôn ngữ ràng buộc đối tượng OCL* (Object Constraint Language), điều kiện này phải thoả để một chuyển đổi trong biểu đồ trạng thái xảy ra. Biểu thức được tạo từ các tham số của thông điệp, các thuộc tính của đối tượng hiện hành, các trạng thái cùng xảy ra của các đối tượng hiện hành hoặc các trạng thái của các đối tượng khác có thể đạt được thông qua các liên kết từ đối

tượng hiện hành. Trong ngữ cảnh của một biểu đồ cộng tác, *guard-condition* xuất hiện để cho biết một thông điệp sẽ không được gởi đi cho đến khi thể hiện của đối tượng thoả một số yêu cầu. Việc dùng một *guard-condition* cũng có tác dụng đồng bộ hoá các tiểu trình của điều khiển. Các tình huống sử dụng *guard-condition* trong biểu đồ cộng tác là không rõ ràng, bởi vì nó có tác dụng của việc ngăn chặn thông điệp được gởi đi cho đến khi điều kiện thoả. Làm cho hệ thống có thể bị treo nếu như điều kiện không bao giờ thoả. Điều này làm cho chúng ta phải có ý thức hơn khi dùng mệnh đề điều kiện trong *sequence-expression* để lặp hoặc để rẽ nhánh. Hình 8.28 và 8.29 minh họa rẽ nhánh.



Hình 8.28: Thông điệp với điều kiện bảo vệ



Hình 8.29: Thông điệp với điều kiện trong sequence-expression

8.5.9 Các ký hiệu khác

Có bốn ký hiệu bổ sung được biết là:

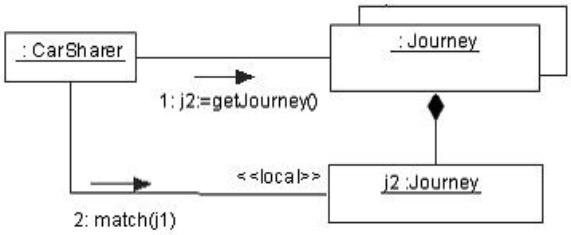
- Các đối tượng hoạt động (active object)
- Đa đối tượng (multiobject)
- Các ràng buộc (constraint)
- Các *quan hệ dòng* (flow relationship)

Active object: là đối tượng sở hữu tiểu trình đồng hành của điều khiển, bởi nó là một đối tượng độc lập đang chạy trên bộ vi xử lý riêng, chẳng hạn như một cơ sở dữ liệu đang chạy trên một server, hoặc bởi nó đang chạy trên một tiểu trình hoặc tiến trình độc lập trên một bộ vi xử lý đơn,

chẳng hạn như *print spooler* (một chương trình hỗ trợ in) trong Windows. Đối tượng hoạt động được biểu diễn với một đường viền dày hoặc với từ khóa thuộc tính {*active*}, như trong hình 8.30.

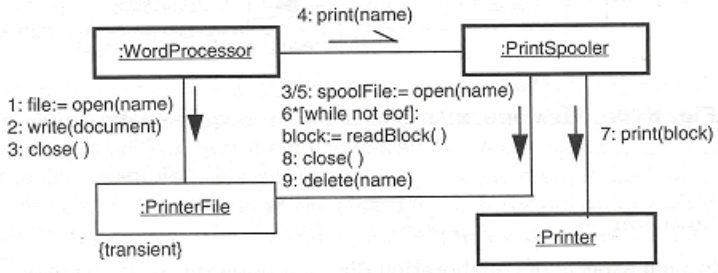
Hình 8.30: Ký hiệu cho đối tượng hoạt động

Multiobject: được mô tả bằng hai hình chữ nhật chồng lên nhau, dùng để biểu diễn một tập các đối tượng bên *many* của một kết hợp và cho biết một thông điệp gởi đến tập này để thiết lập liên kết đến từng đối tượng sao cho một thông điệp sau này có thể gởi cho từng đối tượng. Một *object* được nối với *multiobject* bằng một kết hợp *composition*, xem hình 8.31.



Hình 8.31: Ký hiệu multiobject

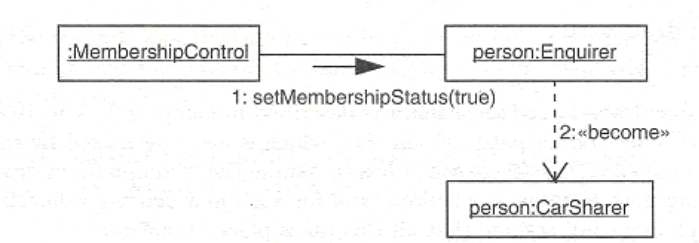
Constraint: được dùng để mô tả đặc trưng của đối tượng hoặc liên kết. Ràng buộc được dùng để đặc tả việc tạo ra như {*new*} hoặc xóa đi như {*destroyed*}. Một đối tượng đơn hoặc một liên kết đơn có thể được tạo ra và huỷ bỏ bên trong quá trình của một tương tác đơn, trong trường hợp này ta dùng ràng buộc {*transient*}, như trong hình 8.32.



Hình 8.32: Ký hiệu ràng buộc

Flow relationship được dùng để biểu diễn một phụ thuộc giữa hai phiên bản của một đối tượng tại những điểm khác nhau trong cùng một thời điểm. Phụ thuộc có thể là «*become*» hoặc «*copy*», xác định đối tượng

đã thay đổi lớp hoặc trạng thái hoặc đã được sao chép. Nhãn khuôn dạng có thể được đánh số để chỉ rõ vị trí trong dãy tương tác, xem hình 8.33.



Hình 8.33: Ký hiệu quan hệ flow

8.6 Cách tạo ra biểu đồ cộng tác

Biểu đồ cộng tác được tạo ra ở nhiều giai đoạn khác nhau của qui trình phát triển:

- Ở giai đoạn đầu, như là một phần của qui trình phát triển biểu đồ lớp, các đối tượng tham gia vào mỗi use case có thể được mô hình trong một biểu đồ cộng tác, từ đó xác định lớp cho chúng. Các lớp này được kết hợp lại phác thảo ra biểu đồ lớp đầu tiên.
- Khi biết các lớp trong hệ thống, biểu đồ cộng tác dùng để xác định tương tác xảy ra giữa các vai trò lớp nhằm hiện thực use case.
- Khi có thao tác, biểu đồ cộng tác dùng để xác định cách thực hiện của các thao tác có hành vi phức tạp.

Bất kể đó là giai đoạn hay mục đích, các bước phát triển biểu đồ cộng tác như sau:

- Dựa vào ngữ cảnh: hệ thống, hệ thống con, use case hoặc thao tác.
- Xác định các phần tử cấu trúc (vai trò lớp, đối tượng, hệ thống con) cần thiết để thực hiện chức năng của cộng tác.
- Mô hình các quan hệ cấu trúc giữa các phần tử này để tạo ra một biểu đồ nhằm biểu diễn ngữ cảnh của tương tác.
- Xem xét một số *scenario* thay thế cần thiết
- Vẽ các biểu đồ cộng tác mức thể hiện nếu cần (hoặc các biểu đồ tuần tự nếu quan tâm đến tính tuần tự của các thông điệp, xem chương 9).
- Vẽ biểu đồ cộng tác mức đặc tả để tóm tắt các *scenario* trong biểu đồ tuần tự mức thể hiện, đây là bước tuỳ chọn.

8.6.1 Dựa vào ngữ cảnh

Biểu đồ cộng tác có thể mô hình hóa tương tác ở mức hệ thống, mức hệ thống con, mức use case hoặc mức thao tác. Giai đoạn phát triển dự án và tác vụ đang được đảm nhiệm sẽ xác định mức nào được mô hình.

Ví dụ 8.1

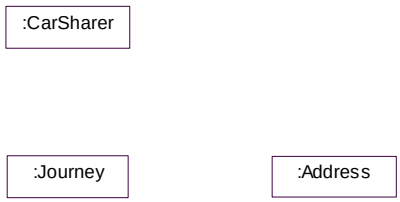
Trong trường hợp này, chúng ta đang mô hình một use case – use case *Register car sharer* của hệ thống *CarMacth*.

8.6.2 Xác định các phần tử thuộc cấu trúc

Trước hết xác định các phần tử cấu trúc sẽ tham gia vào cộng tác. Chúng bao gồm phần tử *giao diện người dùng*, phần tử *điều khiển* và các lớp hoặc đối tượng biểu diễn các thực thể nghiệp vụ của hệ thống.

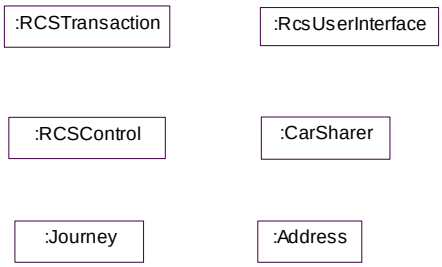
Ví dụ 8.2

Use case *Manually add car sharer***,** mô tả trong chương 3, là một chuyên biệt của *Register car sharer* (mô tả trong mục 3.3.2, chương 3). Một chuyên biệt khác là use case *Transfer car sharer from web server*. Use case *Register car sharer* phải xử lý chức năng chung của việc tạo ra một *CarSharer* mới, tạo các *Journey* mới theo yêu cầu và kiểm tra các *Address* của mỗi *Journey*. Trong ví dụ này, chúng ta giả sử tất cả những điều trên xảy ra một lần. Các lớp liên quan ở đây là *CarSharer*, *Journey* và *Address* (hình 8.34). Ngoài ra còn có một thể hiện của *lớp điều khiển* nhằm quản lý cộng tác và có một thể hiện của *lớp giao diện người dùng* hoặc một *giao dịch web* để cung cấp dữ liệu, như trong hình 8.35.



Hình 8.34: Các lớp tham gia vào cộng tác

Boot và các cộng sự (1999) gợi ý đặt các đối tượng quan trọng ở giữa và các đối tượng ít quan trọng xung quanh. Tuỳ vào cấu trúc của cộng tác, cách bố trí này không phải luôn thích hợp; cấu trúc *top-down* hoặc *left-right* có thể dễ bố trí hơn.



Hình 8.35: Thêm các lớp giao diện và điều khiển vào cộng tác

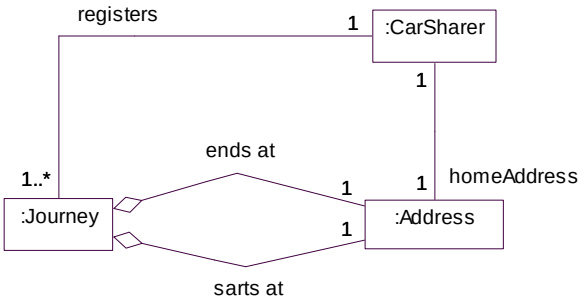
8.6.3 Mô hình hóa các mối quan hệ cấu trúc

Thêm các quan hệ kết hợp giữa các lớp vào biểu đồ cộng tác. Trong một vài trường hợp, một số lớp có thể tham gia vào các vai trò kết hợp khác nhau. Nếu vai trò lớp, trạng thái hoặc các thuộc tính của một lớp hay của một đối tượng thay đổi trong quá trình cộng tác, thì ta đặt một bản sao chứa các giá trị mới vào biểu đồ cộng tác và nối đối tượng cũ với đối tượng mới bằng một thông điệp theo khuôn dạng «*become*» hoặc «*copy*».

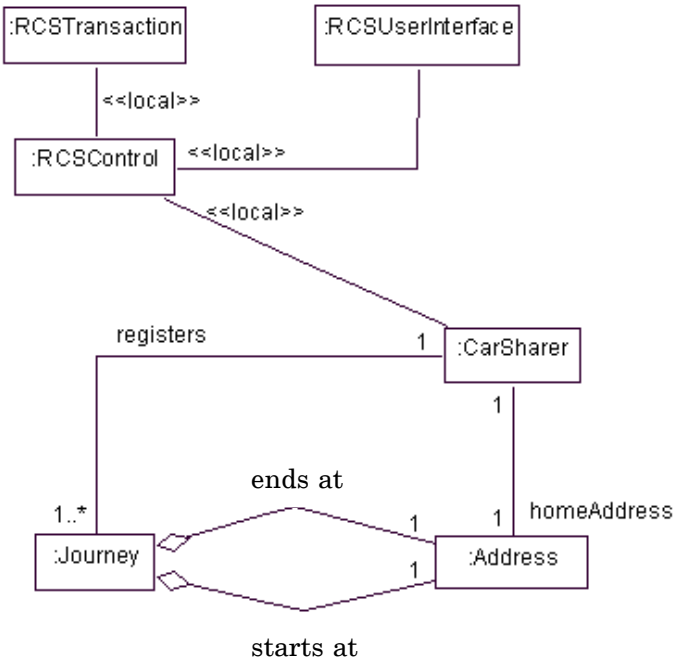
Ví dụ 8.3

Có các kết hợp giữa *CarSharer* và các lớp *Journey* và *Address*, và có hai quan hệ kết tập giữa *Journey* và *Address*, xem hình 8.36. Cũng cần các lớp *control* và lớp *interface*. Trừ phi mô hình tương tác chi tiết, còn không phải luôn quan sát được các kết hợp giữa các lớp này với các lớp chính, vì vậy một vài kết hợp được thêm vào biểu đồ (hình 8.37) và có khuôn dạng là «*local*».

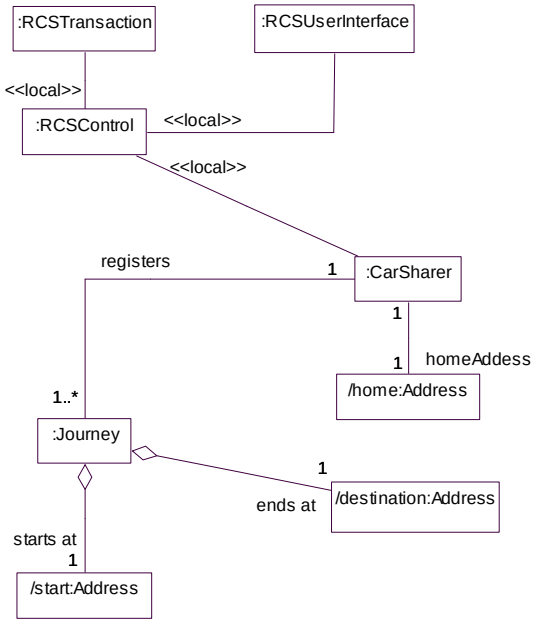
Các quan hệ cấu trúc này sẽ làm nổi bật sự cần thiết có lớp *Address* trong biểu đồ với các vai trò khác nhau. Các quan hệ này được thêm vào như trong hình 8.38.



Hình 8.36: Các kết hợp của các lớp chính



Hình 8.37: Một vài kết hợp được thêm vào lớp control *và lớp* interface



Hình 8.38: Các vai trò được bổ sung cho Address

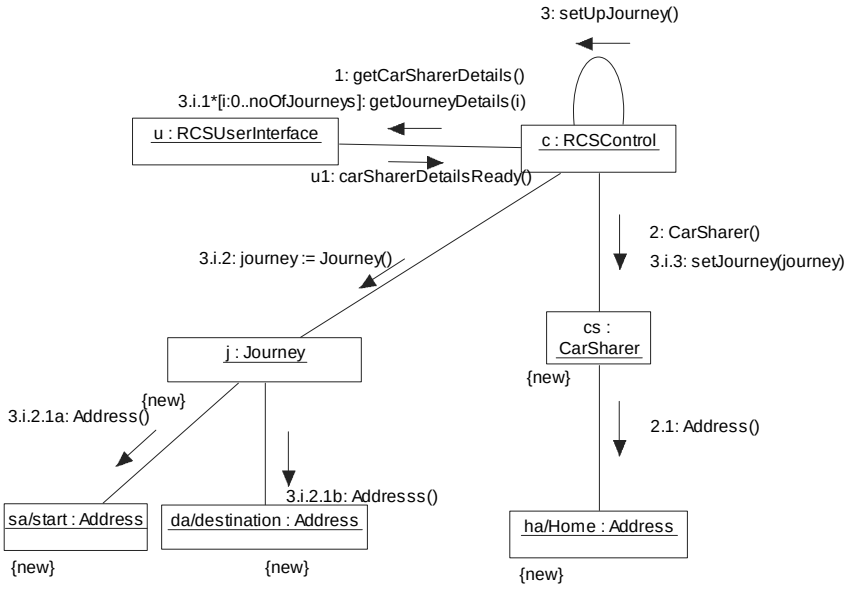
8.6.4 Xem xét các *scenario*

Có nhiều cách phát triển tương tác, tuỳ vào thông tin đầu vào của hệ thống hoặc trạng thái của các đối tượng hiện có. Chúng đã được sưu liệu trong các đặc tả hành vi trong mô hình use case, hoặc người phát triển hệ thống phải làm việc với các *scenario* này như một phần của qui trình tạo ra các biểu đồ cộng tác. Những kịch bản này liên quan đến các điểm trong tương tác, nơi xuất hiện các nhánh rẽ, hoặc liên quan đến các vòng lặp với số lần lặp cụ thể. Các biểu đồ cộng tác cùng với các tương tác sẽ được vẽ cho các *scenario* có ý nghĩa này.

Ví dụ 8.4

Hai *scenario* hiện thực use case *Register car sharer* tuỳ vào tính năng của nó được gọi từ *Manually add car sharer* hay *Transfer car sharer from web server*. Trường hợp đầu, đối tượng *control* (điều khiển) tương tác với đối tượng *interface* (giao diện người dùng) và dữ liệu địa chỉ được hệ thông tin địa lý xác nhận và một tham chiếu bản đồ được định vị. Trường hợp sau, đối tượng *control* tương tác với đối tượng *transaction* (giao dịch) web và địa chỉ được định vị (định vị một tham chiếu bản đồ). Trong cả hai trường hợp, số lần lặp phụ thuộc vào số lộ trình được nhập.

8.6.5 Vẽ các biểu đồ cộng tác mức thể hiện



Hình 8.39: Cộng tác liên quan đến giao diện người dùng

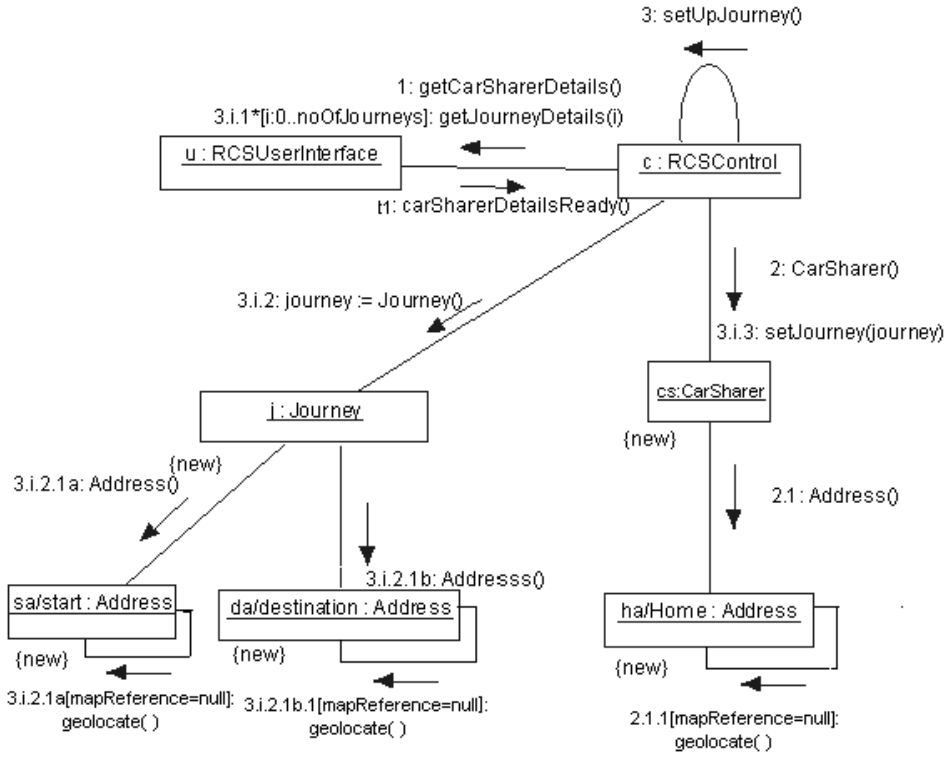
Các biểu đồ mức thể hiện dùng để biểu diễn các thể hiện đối tượng tham gia vào cộng tác. Nếu các biểu đồ cộng tác được dùng để phát triển biểu đồ lớp, thì rõ ràng các lớp của các đối tượng có thể chưa được biết ở giai đoạn này. Trong ví dụ dưới đây chúng ta biết các vai trò.

Bắt đầu với thông điệp gây ra tương tác, thêm từng thông điệp vào liên kết thích hợp và thêm một *sequence-term* vào từng thông điệp. Thêm các ràng buộc cần thiết vào các đối tượng hoặc liên kết.

Ví dụ 8.5

Hình 8.39 minh họa tình huống cộng tác liên quan đến một giao diện người dùng, hình 8.40 minh họa tình huống cộng tác liên quan đến một giao tác web.

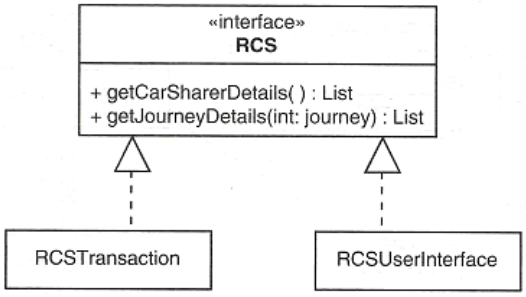
Ràng buộc {*new*} được thêm vào các đối tượng được tạo ra trong tương tác.



Hình 8.40: Cộng tác liên quan đến một giao tác web

Ở đây có một số giả thuyết. Thứ nhất, đối tượng được tạo ra bằng *constructor* (thao tác khởi tạo) có tên trùng với tên lớp. Đây là qui ước

trong một số ngôn ngữ, như Java và C++. Có thể thay bằng stereotype «*create*». Thứ hai, để giữ cho biểu đồ đơn giản, các giá trị trả về và các tham số được bỏ đi trong hầu hết các trường hợp. Ngoại trừ thể hiện của *Journey* được tạo ra bởi đối tượng *control* và sau đó được thêm vào *CarSharer*. Ví dụ, các tham số cho constructor *CarSharer*() sẽ là *firstname*, *lastname*, *dob*, *regDate*, *status* và các tham số khác cần thiết cho địa chỉ nhà. Thứ ba, giả sử *CarSharer* chịu trách nhiệm tạo ra đối tượng *Address* của riêng nó, còn các *Journey* là do đối tượng control tạo ra. Điều này nhằm tránh việc phải truyền tất cả dữ liệu tạo ra *Journeys* thông qua *CarSharer*. Thứ tư, tất cả các tương tác được kết hợp với người dùng thông qua giao diện người dùng hoặc với việc truyền tải của dữ liệu thông qua giao tác web là ngoài phạm vi của sự thực hiện use case này. Cộng tác này cung cấp chức năng được thừa kế bởi hai cộng tác khác. Cả hai lớp *RCSTransaction* và *RCSUserInterface* phải hỗ trợ thao tác *getCarSharerDetails*() và thao tác *getJourneyDetails*(). Một cách thực hiện điều này là định nghĩa một giao diện cài đặt cả hai thao tác, như trong hình 8.41.



Hình 8.41: Các lớp cài đặt giao diện RCS

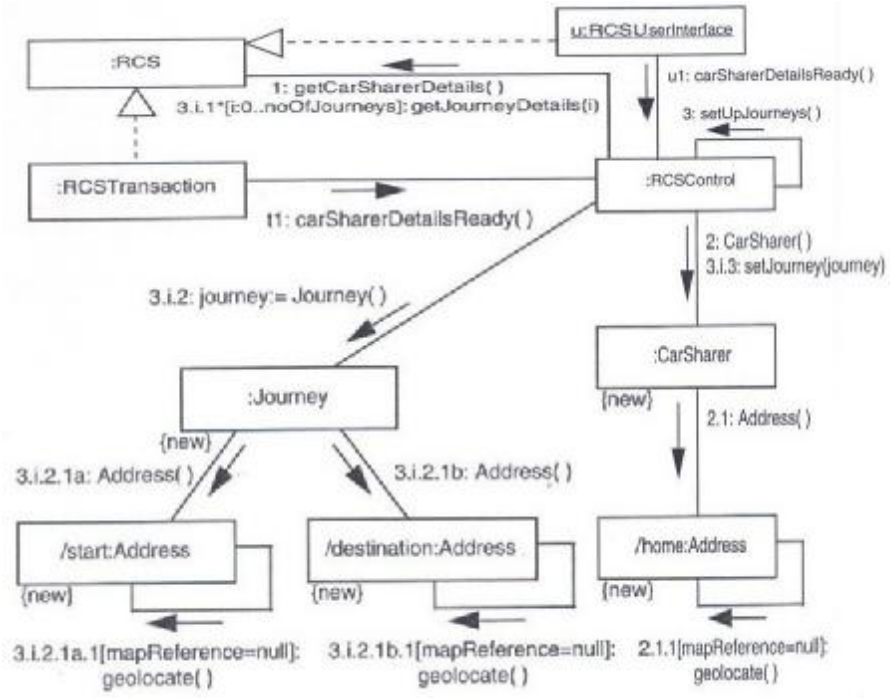
8.6.6 Vẽ biểu đồ cộng tác mức đặc tả

Biểu đồ mức thể hiện có thể được kết hợp vào một biểu đồ mức đặc tả. Để làm điều này, tất cả các nhánh (*scenario*) phải được biểu diễn trên biểu đồ. Các vai trò lớp thường được dùng nhiều hơn các thể hiện đối tượng.

Ví dụ 8.6

Thông điệp *CarSharerDetailsReady*() được phát sinh từ *RCSUserInterface* hoặc *RCSTransaction*, do cả hai đều cài đặt giao diện *RCS*. *RCSControl* không cần biết lớp phát sinh thông điệp nó chỉ gởi thông điệp trở lại đối tượng cài đặt giao diện mà nó yêu cầu.

Nếu có bất kỳ đường đi nào hợp lệ thì tất cả chúng đều được hiển thị trong biểu đồ này. Hình 8.42 là một giải pháp.



Hình 8.42: Cộng tác mức đặc tả

8.7 Lập mô hình nghiệp vụ với biểu đồ cộng tác

Trong profile của UML dành cho việc lập mô hình nghiệp vụ, có năm khuôn dạng được định nghĩa cho các đối tượng nghiệp vụ là: *Actor*, *Worker*, *Case Worker*, *Internal Worker* và *Entity*. Các Actor đã được định nghĩa trong UML. Ý nghĩa của các khuôn dạng còn lại như sau:

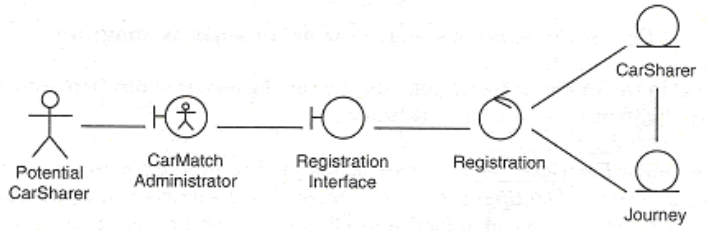
- **Worker**: Một trừu tượng hoá của một người hoạt động bên trong hệ thống tương tác với các *worker* khác và các *thực thể* (entity) trong một hiện thực hoá use case.
- **Case Worker**: Một *worker* tương tác trực tiếp với các *actor* bên ngoài hệ thống.
- **Internal Worker**: Một *worker* tương tác với các *worker* khác và các thực thể bên trong hệ thống.
- **Entity**: Một lớp thụ động được thao tác bởi các *worker*, tham gia trong các tương tác khác. Entity là các lớp nghiệp vụ, như *Address*, *Journey*, *Invoice* và *Product*.

Cùng với các lớp *control* và *interface*, các lớp *entity* đã được sử dụng và là thành phần của profile *qui trình phát triển phần mềm UML*. Hình 8.43 minh họa các biểu tượng được dùng cho các khuôn dạng này.

Hình 8.43: Các biểu tượng cho các khuôn dạng lớp nghiệp vụ

Chúng được dùng trong biểu đồ cộng tác để mô hình các tương tác giữa *worker* và *entity* trong hiện thực use case. Hình 8.44 minh hoạ cách sử dụng chúng để mô hình hoá use case *Manually add car sharer*.

Loại biểu đồ này được tạo ra ở giai đoạn đầu của qui trình phát triển hệ thống, như một phần của việc lập mô hình nghiệp vụ, trước khi thực hiện phân tích chi tiết.



Hình 8.44: Biểu đồng cộng tác nghiệp vụ

8.8 Quan hệ với các biểu đồ khác

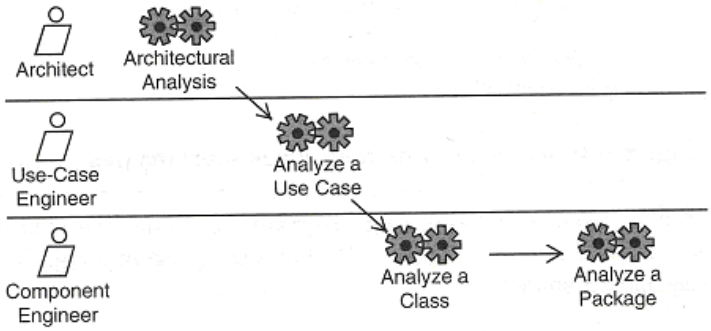
Biểu đồ cộng tác được dùng để mô hình việc hiện thực các use case hoặc các thao tác của lớp. Ở trường hợp sau, mỗi thao tác phải tồn tại trong biểu đồ lớp. Trong biểu đồ mức thể hiện, tên thông điệp phải là biến cố hoặc thao tác của lớp nhận thông điệp. Nếu các trạng thái được tham chiếu trong điều kiện bảo vệ thì chúng phải là các trạng thái hợp lệ của lớp có liên quan và nên xuất hiện trong một biểu đồ trạng thái, chương 11 sẽ trình bày về biểu đồ trạng thái.

Biểu đồ cộng tác và biểu đồ tuần tự (chương 9) mô hình các khía cạnh giống nhau của hệ thống: các đối tượng và các vai trò lớp cộng tác với nhau và các thông điệp được trao đổi giữa chúng để đạt được mục tiêu. Các biểu đồ này có thể được chuyển đổi qua lại. Một số công cụ thực hiện điều đó một cách tự động. Hầu hết thông tin đều được chuyển đổi nhưng không phải là tất cả, một số thông tin sẽ bị mất khi chuyển đổi: biểu đồ

cộng tác không hiển thị thông tin ràng buộc thời gian, biểu đồ tuần tự không hiển thị các liên kết giữa các đối tượng hoặc các kết hợp giữa các vai trò lớp.

8.9 Biểu đồ cộng tác trong UP

Biểu đồ cộng tác được sử dụng trong UP trước tiên trong dòng *Analysis*.



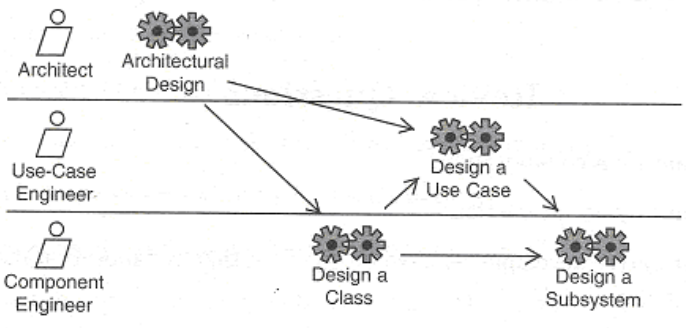
Hình 8.45: Analysis workflow

Analysis Model (mô hình phân tích) trong UP gồm có một tập các *hiện thực use case* (use case realization) và một tập các *classifier* tham gia.

Trong hoạt động *Analyze a Use Case* (phân tích một use case), từng use case được phân tích để tạo ra một hiện thực use case. Điều này được thực hiện trong hai bước chính *Identitying Analysis Classes* (xác định lớp phân tích) và *Describing Analysis Object Interactions* (mô tả tương tác đối tượng phân tích). Trước hết, các lớp tham gia được xác định và các biểu đồ lớp được tạo ra và chỉ biểu diễn các lớp có liên quan trong mỗi use case, sau đó các cộng tác được phân tích để tạo ra các biểu đồ cộng tác, lúc này các tương tác cũng được thêm vào biểu đồ. Trong *Analysis Model*, các biểu đồ cộng tác này sẽ sử dụng các tên bình thường cho các thông điệp để biểu diễn mục đích của từng thông điệp. Các tên này sau đó sẽ được thay thế bằng tên biến cố hoặc tên thao tác trong các lớp thiết kế. Trong thuật ngữ UML, use case cũng là *classifier* và cũng có thể được mô tả bởi biểu đồ hoạt động, biểu đồ trạng thái và biểu đồ tuần tự. Biểu đồ tuần tự cũng được dùng để biểu diễn cộng tác. Các cộng tác cũng được lập sưu liệu bằng cách dùng *Flow of Events Analysis*. Hoạt động này dùng văn bản để mô tả cho tương tác xảy ra bên trong một hiện thực use case. Các mô tả dạng văn bản này khác với văn bản mô tả được tạo ra để xác định hành vi của các use case trong *Requirements Workflow*, bởi vì chúng mô tả cho sự tương tác giữa các lớp bên trong hệ thống, trong khi đó các mô tả use case lại mô tả cho hành vi bên ngoài của use case (xem

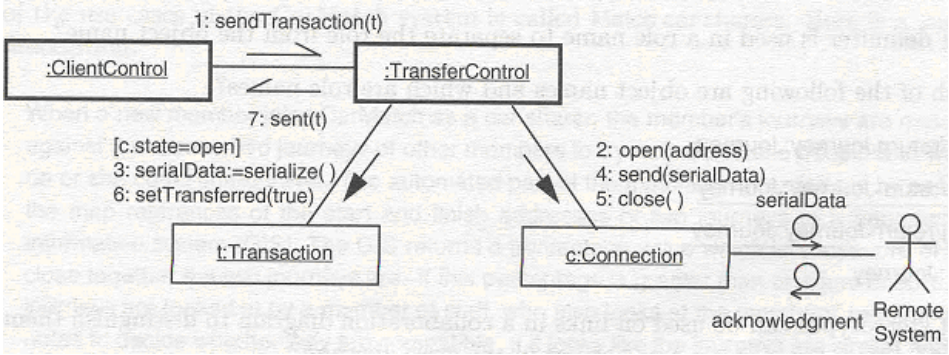
mục 3.3.2). Ở đây còn có bước thứ ba, *Capturing Special Requirements*, trong bước này tất cả các yêu cầu có liên quan đến use case đều được lập sưu liệu, kể cả các yêu cầu phi chức năng.

Biểu đồ cộng tác được tạo ra như một phần của *Analysis Model* được tinh chế trong thiết kế. Hình 8.46 trình bày dòng thiết kế *Design Workflow*.



Hình 8.46: Design Workflow

Trong hoạt động *Architectural Design*, người thiết kế tìm kiếm và thiết kế các *cộng tác tổng quát*. Các cộng tác này sẽ được đặc tả bởi các cộng tác cụ thể bên trong hệ thống. Để làm điều này, người thiết kế phải tìm các mẫu tương tác chung trong các cộng tác khác nhau. Chúng bao gồm các *vai trò actor* tương tự nhau và các vai trò lớp có cấu trúc giống nhau. Trong hệ thống *CarMatch*, use case *Process credit card payment* và use case *Notify insurer* đều được phát sinh bởi việc hoàn tất một giao dịch tài chính và liên quan đến việc chuyển giao dịch này cho một hệ thống bên ngoài, do đó là các ứng viên cho tiếp cận này (chú ý có sự khác biệt với use case *Transfer direct debit to ABTS* xảy ra như một xử lý theo lô). Ví dụ về một cộng tác tổng quát như thế được minh họa trong hình 8.47.



Hình 8.47: Cộng tác tổng quát thực hiện việc chuyển giao các giao tác cho một hệ thống từ xa

Hoạt động *Design a Use Case* giống với hoạt động phân tích cùng tên, ngoại trừ các lớp thiết kế được sử dụng và có nhiều bước hơn, vì khi một dự án tiến gần đến cài đặt thì cần nhiều chi tiết hơn. Các bước chính là: *Identifying the Participating Design Classes* (xác định lớp thiết kế), *Describing Design Object Interactions* (mô tả tương tác đối tượng thiết kế), *Identifying the Participating Subsystems and Interfaces* (xác định hệ thống con và giao diện) và *Describing Subsystem Interactions* (mô tả tương tác hệ thống con). Các lớp thiết kế có thể khác với các lớp phân tích. Ví dụ, các lớp tập hợp có thể được thêm vào trong thiết kế để quản lý các tương tác có liên quan đến một tập hợp các đối tượng. Các tương tác trên biểu đồ tương tác và biểu đồ tuần tự sẽ phải được sửa đổi để chứa các thể hiện của các lớp này cùng các tương tác của chúng. Trong UP, biểu đồ tuần tự được khuyến khích sử dụng hơn là biểu đồ cộng tác khi lập mô hình tương tác trong thiết kế. Một use case cũng có thể được xem xét dưới dạng các tương tác giữa các hệ thống con thiết kế hơn là các đối tượng thiết kế. Điều đó cung cấp một phân rã phân cấp của các tương tác, bởi vì chi tiết của tương tác bên trong một hệ thống con có thể bị che giấu trong một biểu đồ và được hiển thị trong một biểu đồ độc lập mức thấp hơn (chương 9). Để hỗ trợ tiếp cận này, giao diện của các hệ thống con phải được xác định để cho biết rõ những thao tác nào hoặc biến cố nào sẽ được xử lý bởi hệ thống con. Giống như hoạt động phân tích, còn có một bước thứ năm, *Capturing Implementation Requirements*, trong bước này các yêu cầu được xử lý trong cài đặt được lập sưu liệu.

Câu hỏi ôn tập

8.1 Cộng tác là gì?

8.2 Tương tác là gì?

8.3 Sự phát triển theo phương pháp lặp có ảnh hưởng gì đến việc phát triển các biểu đồ cộng tác?

8.4 Trình bày các mục đích chính của việc dùng biểu đồ cộng tác.

8.5 Ký hiệu của cộng tác là gì?

8.6 Những phụ thuộc nào có thể tồn tại giữa các cộng tác, các thao tác và các use case?

8.7 Ký hiệu *mẫu* (template) được dùng như thế nào để biểu diễn một mẫu thiết kế dùng một cộng tác?

8.8 Sự khác biệt giữa một biểu đồ cộng tác mức đặc tả và biểu đồ cộng tác mức thể hiện là gì?

8.9 Trình bày mục đích của việc vẽ một biểu đồ cộng tác ngữ cảnh.

8.10 Hãy phân biệt tên của các thể hiện đối tượng với tên của các vai trò trong biểu đồ cộng tác.

8.11 Ký hiệu phân cách nào được sử dụng trong một tên vai trò để tách rời tên vai trò với tên đối tượng?

8.12 Trong cách tên dưới đây, đâu là tên đối tượng và đâu là tên vai trò?

(a) /returnJourney:Journey

(b) /returnJourney:Journey

(c) j/returnJourney:Journey

(d) :Journey

8.13 Các *stereotype* nào được dùng trên liên kết trong biểu đồ cộng tác để phân biệt với những liên kết là thể hiện của các kết hợp tĩnh trong biểu đồ lớp?

8.14 Stimulus trong biểu đồ cộng tác là gì?

8.15 Hãy trình bày sự khác biệt giữa message (thông điệp) và stimulus.

8.16 Ba loại stimulus nào có thể được dùng trong một biểu đồ cộng tác?

8.17 Bốn kiểu thông điệp nào có thể được biểu diễn bởi các mũi tên khác nhau trong một biểu đồ cộng tác?

8.18 Hãy phân biệt một biến cố xuất phát từ thao tác với một stimulus.

8.19 Giá trị trả về (*Return-value*) là gì?

8.20 Sequence-term trong một thông điệp là gì?

8.21 Các tên có thể được dùng như thế nào trong các sequence-expression?

8.22 Guard-condition là gì?

8.23 Active object (đối tượng hoạt động) là gì?

8.24 Ký hiệu của đối tượng hoạt động là gì?

8.25 Ký hiệu cho một multiobject là gì?

8.26 Các ràng buộc đặc biệt nào được biểu diễn trên các liên kết hoặc trên các đối tượng trong một biểu đồ cộng tác?

8.27 Các phụ thuộc nào có thể được biểu diễn bởi flow relationship?

8.28 Trình bày các bước để tạo ra một biểu đồ cộng tác?

8.29 Các stereotype nào được định nghĩa trong profile *mô hình nghiệp vụ*?

8.30 Hai dòng công việc UP nào có sử dụng các biểu đồ cộng tác?

8.31 Hãy cho biết ba bước trong hoạt động *Analysis a Use Case*.

8.32 *Flow of Events Analysis* khác với *Use Case Description* như thế nào?

8.33 Cộng tác tổng quát là gì?

8.34 Năm bước trong hoạt động *Design a Use Case* là năm bước nào?

Bài tập có lời giải

8.1 Một trong các use case trong hệ thống CarMatch được gọi là *Macth car sharers*. Dưới đây là phần mô tả tóm tắt của nó.

> Khi một thành viên mới gia nhập vào *CarMatch*, thì các lộ trình của thành viên này được kết hợp với các lộ trình còn trống (các lộ trình chưa được kết hợp) với các thành viên khác để tìm người phù hợp nhằm chia sẻ lộ trình. Quá trình kết hợp tự động được thực hiện bằng cách gởi các tham chiếu bản đồ của địa chỉ đầu và địa chỉ đích của hai lộ trình đến hệ thống thông tin địa lý (GIS). GIS sẽ trả về giá trị phần trăm dựa trên khoảng cách xa gần của hai lộ trình. Nếu giá trị trả về từ 80% trở lên thì nhân viên của *CarMatch* sẽ xem xét đến chúng cùng cùng các yêu cầu của thành viên rồi ra quyết định xem các thành viên này có thể được kết hợp với nhau không. Nếu được, thì một lá thư sẽ được gởi đi.

Rõ ràng ngữ cảnh của cộng tác này là một use case. Các lớp domain liên quan với use case này là các lớp nào? (Bạn có thể tham khảo thêm các biểu đồ lớp trong chương 5).

Lời giải:

Có 3 lớp liên quan trong use case này là *CarSharer*, *Journey* và *Address*.

8.2 Những hệ thống con khác nào có liên quan trong use case này?

Lời giải:

Hệ thống thông tin địa lý là một hệ thống độc lập. Ở đây chúng ta không bàn đến cách làm việc của nó, chúng ta chỉ bàn đến các dịch vụ mà nó cung cấp (kết hợp lộ trình) và quan tâm đến giao diện dùng để lưu giữ dịch vụ đó. Trong trường hợp đặc biệt này, do có vị trí địa lý của các địa chỉ trong ví dụ ở bên trên nên chúng ta không cần biểu diễn nó trong biểu đồ cộng tác. Nếu chúng ta đang vẽ một biểu đồ cộng tác cho thao tác *Address*:*geolocate*() hoặc *Journey*:*matchJourney*(*journey*), thì chúng ta cần phải biểu diễn nó.

8.3 Các lớp bổ sung cần thiết cho cộng tác này là các lớp nào?

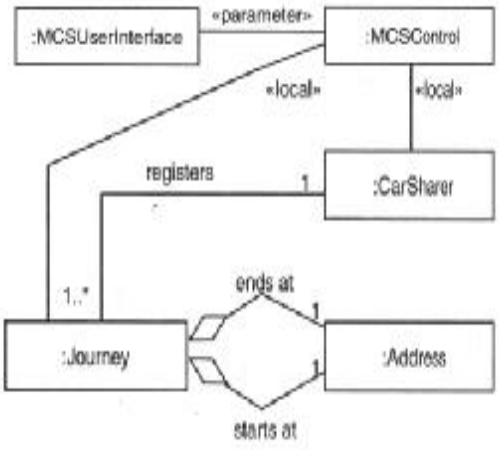
Lời giải:

Cộng tác yêu cầu một lớp control *MCSControl* và một lớp interface *MCSUserInterface*.

8.4 Hãy vẽ một biểu đồ lớp biểu diễn các kết hợp giữa các lớp, kể cả các kết hợp stereotype nếu cần.

Lời giải:

Hình 8.48 là một giải pháp. Lưu ý rằng kết hợp *homeAddress* giữa *CarSharer* và *Address* là không được yêu cầu trong cộng tác này.

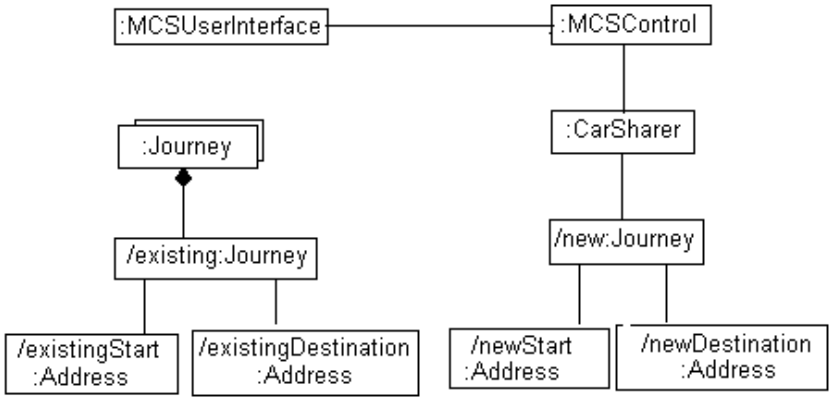


Hình 8.48: Các lớp tham gia vào cộng tác

8.5 Hãy cho biết các vai trò lớp tham gia vào cộng tác trong hình 8.48.

Lời giải:

Mỗi Journey có hai vai trò địa chỉ: */start:Address* và */destination:Address*. Use case này bao gồm việc kết hợp các *Journey* từ một tập các *Journey* chưa được kết hợp với các *Journey* dành cho *CarSharer* mới. Use case này cũng bao gồm việc sử dụng một *multiobject* cũng như các vai trò khác nhau cho các *Journey* khác nhau. Hình 8.49 trình bày các vài trò lớp có liên quan tham gia vào cộng tác.



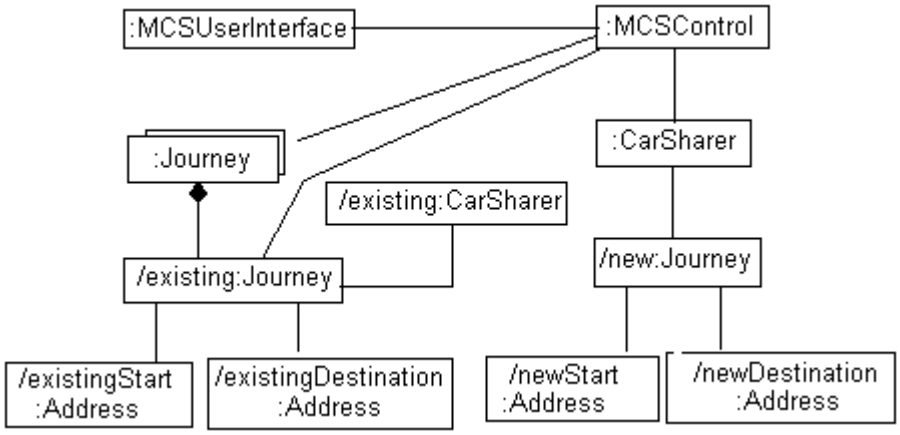
Hình 8.49: Các vài trò của các lớp tham gia vào cộng tác

8.6 Những *scenario* nào có thể xảy ra?

Lời giải:

Có hai *scenario* có ý nghĩa. Thứ nhất, tìm thấy một *Journey* khớp với các lộ trình mới và chi tiết của chúng cũng như các yêu cầu của các thành viên được hiển thị cho người quản trị. Thứ hai, không có lộ trình nào khớp với các lộ trình mới từ 80% trở lên, cho nên không có chi tiết nào được hiển thị. Xét *scenario* thứ nhất, ta nhận ra

rằng để hiển thị các yêu cầu ta cần thêm một vai trò lớp khác trong biểu đồ cộng tác. Hình 8.50 là kết quả.

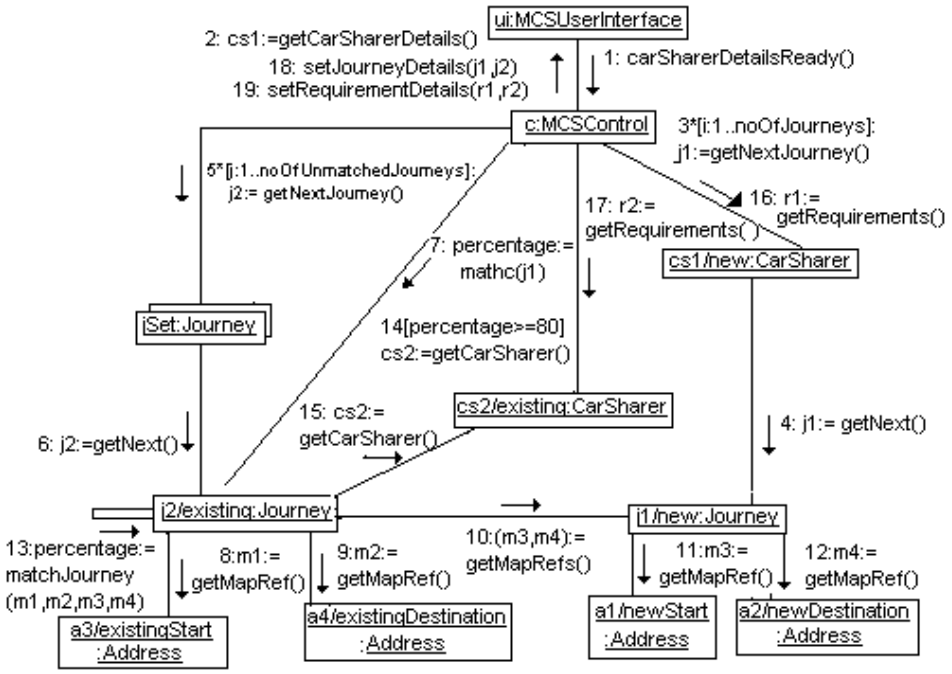


Hình 8.50: Các vai trò đã được sửa lại của các lớp tham gia vào cộng tác

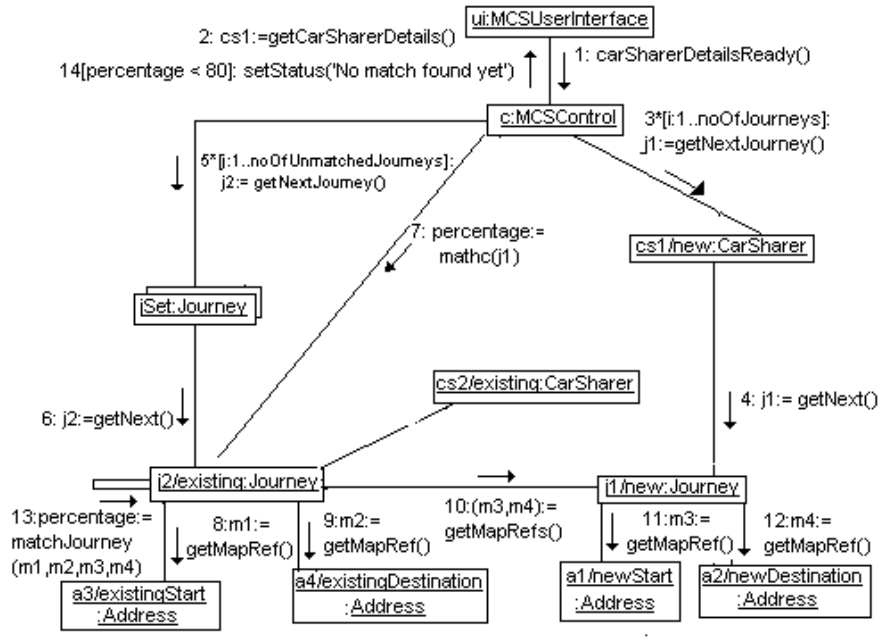
8.7 Hãy vẽ một biểu đồ cộng tác mức thể hiện cho mỗi scenario này.

Lời giải:

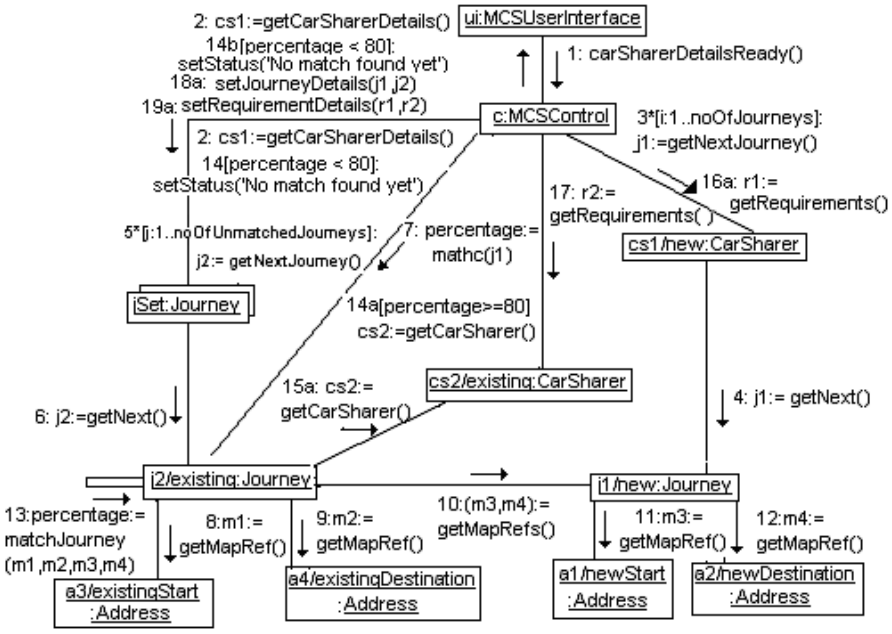
Hai scenario khác nhau được biểu diễn trong hình 8.51 và 8.52. Các thông điệp được đánh số thứ tự vì các biểu thức tuần tự rất phức tạp, ví dụ **1.2.i.j.1.1a**.



Hình 8.51: Biểu đồ cộng tác mức thể hiện cho scenario thứ nhất



Hình 8.52: Biểu đồ cộng tác mức thể hiện cho scenario thứ hai



Hình 8.53: Biểu đồ cộng tác mức đặc tả

8.8 Hãy vẽ biểu đồ cộng tác mức đặc tả kết hợp hai scenario này lại.

Lời giải:

Hình 8.51 và 8.52 đã biểu diễn hai scenario này trong hai biểu đồ riêng. Biểu đồ trong hình 8.53 trình bày một biểu đồ kết hợp của hai biểu đồ này, một tập các nhánh có nhãn chung là "a" và một nhánh có nhãn là "b".

Biểu đồ này rất phức tạp. Thường thì các biểu đồ cộng tác được vẽ từ việc trích từ một cộng tác, thay vì biểu diễn tất cả thông tin vào một biểu đồ duy nhất.

Bài tập bổ sung

8.9 Hãy vẽ một cộng tác được gọi là *Process payment* chứa các đối tượng từ các lớp *CarSharer*, *Account* và *Transaction*. Sau đó thêm lớp *control* và lớp *interface* thích hợp vào biểu đồ.

8.10 Hãy bố trí các đối tượng của các lớp *PPUserInterface*, *PPControl*, *CarSharer*, *Account* và *Transaction* và thêm các thông điệp sau đây vào biểu đồ.

- *transactionReady*()
- *itemList* := *getTransactionDetails*()
- *account* := *getCarSharerAccount*()
- *transaction* := *Transaction*(*amount*, *type*, *date*, *account*)
- *balance* := *postTransaction* (*transaction*)
- *displayBalance*(*balance*)

Biến *itemList* chứa các mục dữ liệu sau: *carSharerID*, *amount*, *type*, *date*.

8.11 Một trong các use case bạn tìm ra trong chương 3 có tên là *Record an individual's request for help*. Nếu không tìm được use case này, hãy tham khảo đoạn mô tả dưới đây.

> Các cá nhân có thể yêu cầu *VolBank* giúp đỡ, kể cả tình nguyện viên. Người quản trị *VolBank* trước hết phải nhập chi tiết của từng người. Nếu đây là một tình nguyện viên đã được lưu trong hệ thống thì tên và chi tiết của họ được hiển thị. Nếu có nhiều người có cùng tên, thì một danh sách sẽ được hiển thị cùng với địa chỉ của mỗi người, do đó người quản trị sẽ chọn được đúng người. Nếu người này là tình nguyện viên mới, thì nhập vào tên, địa chỉ và số điện thoại của người này được nhập vào. Thông tin mã vùng và mã bưu điện cũng rất quan trọng bởi vì quá trình kết hợp cũng dựa vào thông tin này. Tiếp theo nhập các chi tiết cần giúp đỡ: một đoạn tóm tắt nội dung cần giúp đỡ và mã mô tả loại công việc. Ví dụ, DES cho *decorating*, GAR cho *garden work* hoặc

PET cho *pet care*. Ngày bắt đầu và ngày kết thúc cần giúp đỡ cũng được nhập vào.

Chúng ta đã biết rằng ngữ cảnh của cộng tác là một use case, vậy những lớp chính nào liên quan trong use case này?

8.12 Các hệ thống con nào khác có liên quan trong use case này?

8.13 Các lớp bổ sung nào sẽ cần thiết cho cộng tác này?

8.14 Hãy vẽ một biểu đồ lớp biểu diễn các kết hợp giữa các lớp, kể cả các kết hợp khuôn dạng nếu cần.

8.15 Hãy cho biết vai trò của các lớp trong cộng tác này.

8.16 Các *scenario* nào có thể xảy ra?

8.17 Hãy vẽ một biểu đồ cộng tác mức thể hiện cho mỗi *scenario* này. Đối với mỗi biểu đồ, thực hiện các bước sau:

- Bố trí các đối tượng.
- Thêm các liên kết giữa các đối tượng.
- Bố trí các thông điệp bắt đầu với thông điệp phát sinh tương tác.
- Thêm các ràng buộc vào biểu đồ.

8.18 Hãy vẽ một biểu đồ cộng tác mức đặc tả để kết hợp các *scenario* này lại với nhau.

Chương 9
BIỂU ĐỒ TUẦN TỰ

9.1 Giới thiệu

Trong các hệ thống hướng đối tượng, tác vụ được thực hiện nhờ các đối tượng tương tác với nhau bằng cách gởi thông điệp. Biểu đồ tương tác được dùng để lập mô hình các tương tác. Một tương tác là một đặc tả cách gởi thông điệp giữa các đối tượng, hoặc giữa các *vai trò lớp*, nhằm thực hiện một tác vụ. Trong UML có hai loại biểu đồ được xếp vào loại *biểu đồ tương tác* (interaction diagram) là *biểu đồ tuần tự* (sequence diagram) và *biểu đồ cộng tác* (collaboration diagram). Trong chương này, chúng ta tìm hiểu về ký hiệu và cách sử dụng biểu đồ tuần tự, căn cứ vào giải thích của ký hiệu và cách sử dụng các biểu đồ cộng tác ở chương 8.

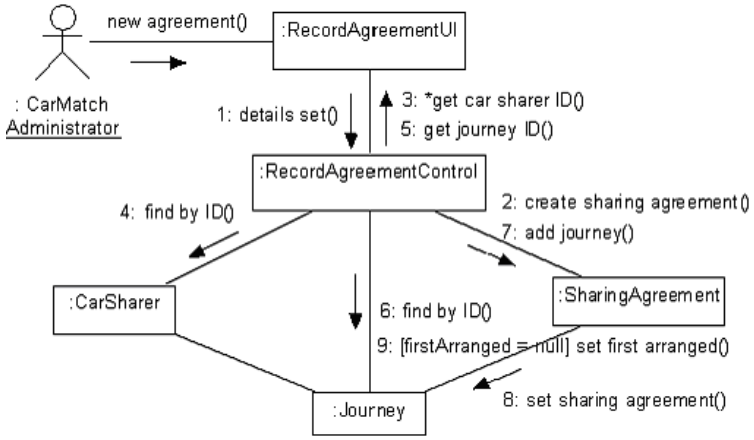
Biểu đồ lớp dùng để mô hình cấu trúc tĩnh của hệ thống; còn biểu đồ tương tác dùng để mô hình các khía cạnh động của hệ thống. Biểu đồ tương tác biểu diễn cách tương tác giữa các đối tượng để thực hiện một số chức năng ở mức cao mà một đối tượng độc lập không thể tự thực hiện được. Biểu đồ cộng tác biểu diễn sự tương tác này trong ngữ cảnh của các *vai trò lớp* tham gia vào tương tác và biểu diễn mối quan hệ cấu trúc giữa các lớp bằng cách dùng các *vai trò kết hợp*. Biểu đồ tuần tự được dùng để biểu diễn sự tương tác giống như biểu đồ cộng tác, nhưng chúng nhấn mạnh tính thứ tự của thông điệp theo thời gian.

9.2 Biểu đồ tuần tự là gì?

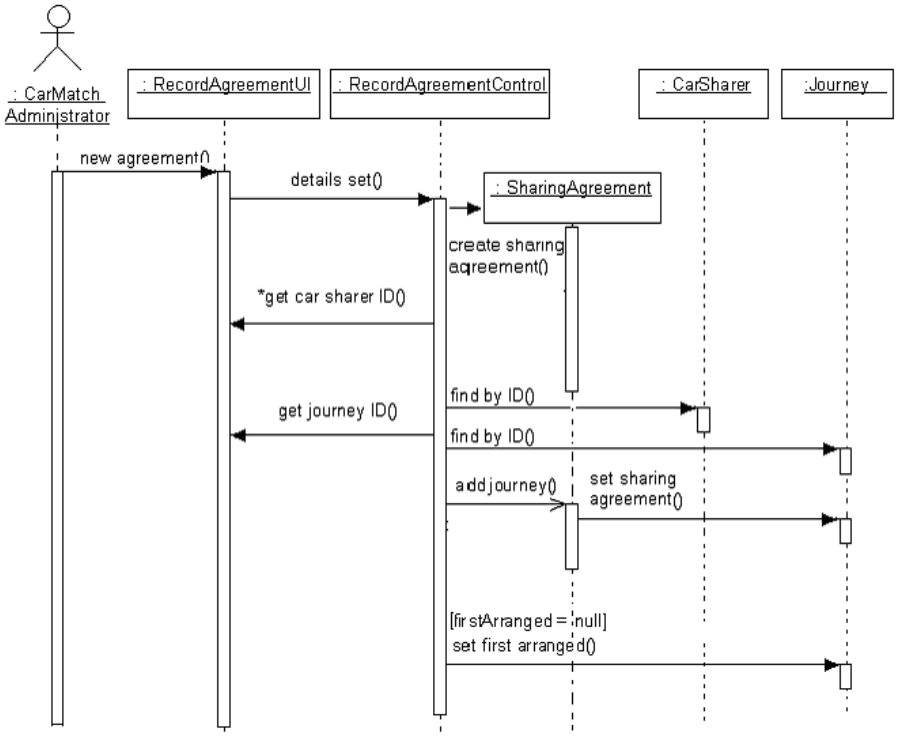
Trong chương 8, chúng ta đã biết cách thêm các tương tác, giữa các đối tượng hoặc *vai trò lớp*, vào biểu đồ cộng tác như là một chuỗi các tác nhân hoặc thông điệp. Hình 9.1 là một biểu đồ cộng tác biểu diễn sự tương tác giữa các vai trò.

Là phương tiện biểu diễn tương tác dưới dạng hình ảnh, biểu đồ cộng tác có hai đặc điểm chính: mô tả các mối quan hệ cấu trúc, giữa các vai trò lớp hoặc giữa các đối tượng dưới dạng vai trò kết hợp hoặc liên kết nhằm phản ánh cấu trúc của biểu đồ lớp, và mô tả thứ tự của tương tác bằng cách đánh số thứ tự các thông điệp. Các biểu đồ tuần tự biểu diễn một số thông tin tương tự, nhưng không phải tất cả. Chúng biểu diễn các thể hiện đóng một vai trò được định nghĩa trong cộng tác; chúng không biểu

diễn các quan hệ cấu trúc giữa các đối tượng mà biểu diễn thứ tự của tương tác một cách trực quan bằng cách dùng trục đứng của biểu đồ để biểu diễn thời gian. Hình 9.2 là biểu đồ tuần tự tương đương hình 9.1.



Hình 9.1: Biểu đồ cộng tác biểu diễn các tương tác



Hình 9.2: Biểu đồ tuần tự tương đương với hình 9.1

Hình 9.2 trình bày các tác nhân tuần tự từ trên xuống do đó không cần thiết phải đánh số. Mặc dù biểu đồ này minh hoạ cùng một cộng tác như biểu đồ cộng tác trong hình 9.1, nhưng ở đây không chỉ ra các liên kết giữa các đối tượng. Chúng ta không thể biết được khả năng của một đối tượng gởi một thông điệp đến một đối tượng khác là dựa trên sự kết hợp cố định trong biểu đồ lớp hay là một liên kết tạm thời (<<*local*>> hoặc <<*parameter*>>).

Biểu đồ tuần tự làm nổi bật thêm một vài khía cạnh. Việc tạo ra một đối tượng được mô tả tường minh bằng mũi tên hướng đến hình chữ nhật chứa tên (đối tượng :*SharingAgreement*), và *đường sinh tồn* của nó (đường thẳng đứng đứt nét) được bắt đầu tại điểm tạo. *Tiêu điểm kiểm soát* (Focus of control) được biểu diễn bằng hình chữ nhật dài và hẹp trên *đường sinh tồn*, cho biết khi nào một đối tượng là hoạt động, hoặc bởi nó đang thực hiện một số hành động hoặc khi nó gởi một thông điệp cho đối tượng khác thực hiện một hành động nhân danh nó.

9.3 Mục đích của kỹ thuật

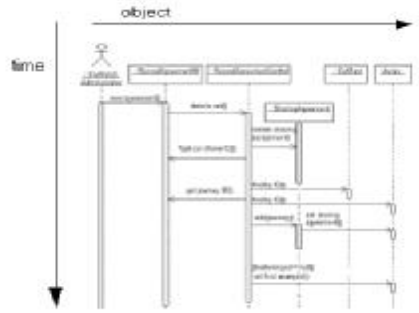
Biểu đồ tuần tự được dùng để mô hình các tương tác giữa các thể hiện đối tượng trong ngữ cảnh của một cộng tác. Cộng tác này được biểu diễn tường minh trong biểu đồ cộng tác, nhưng không tường minh trong biểu đồ tuần tự. Các thể hiện thường được sử dụng nhiều hơn các vai trò, nhưng phải nhớ rằng mỗi thể hiện đang đảm nhiệm một vai trò đã được định nghĩa trong một cộng tác.

Biểu đồ tuần tự có thể được sử dụng trong những trường hợp sau:

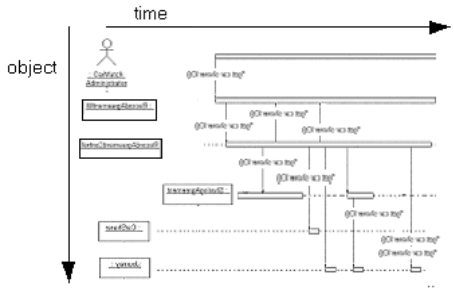
- Lập mô hình tương tác ở mức cao giữa các đối tượng hoạt động.
- Lập mô hình tương tác giữa các thể hiện đối tượng bên trong một cộng tác nhằm hiện thực một use case.
- Lập mô hình tương tác giữa các đối tượng bên trong một cộng tác nhằm hiện thực một thao tác.
- Lập mô hình các tương tác tổng thể (biểu diễn tất cả các đường đi có thể có thông qua tương tác) hoặc dùng để xác định các thể hiện của một tương tác (chỉ biểu diễn một đường đi thông qua tương tác).

9.4 Ký hiệu của biểu đồ tuần tự

Trong biểu đồ tuần tự, các thể hiện đối tượng thường được sắp xếp theo hàng ngang còn thời gian được biểu diễn trên trục đứng từ trên xuống (hình 9.3), hoặc ngược lại (hình 9.4). Trong thực tế cách biểu diễn thứ hai hiếm khi gặp.

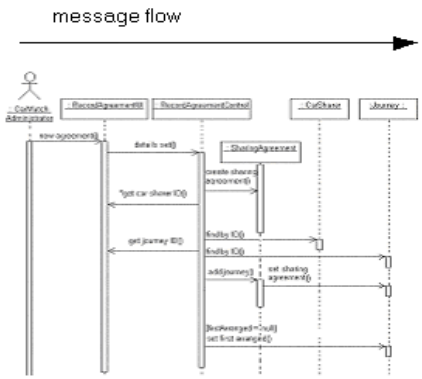


Hình 9.3: Trục thời gian thẳng đứng



Hình 9.4: Trục thời gian nằm ngang

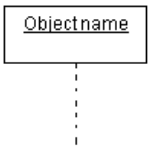
Thứ tự của các đối tượng không quan trọng, nhưng theo qui ước thì các tác nhân bên ngoài và các đối tượng giao diện được đặt bên trái và có một dòng thông điệp tổng quát đi ngang qua trang từ trái sang phải, như Hình 9.5.



Hình 9.5: Hướng của dòng thông điệp

9.4.1 Đường sinh tồn (*lifeline*) và sự hoạt động (*Activation*)

Các thể hiện đối tượng được biểu diễn trong biểu đồ tuần tự bởi một đường thẳng đứng đứt nét với ký hiệu đối tượng trên đỉnh của đường. Đường này được gọi là *đường sinh tồn* (lifeline) của đối tượng, như trong hình 9.6.



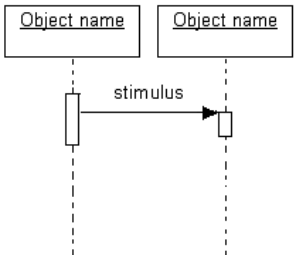
Hình 9.6: Ký hiệu của đường sinh tồn

Đường sinh tồn biểu diễn thời gian tồn tại của đối tượng. Nếu một đối tượng nào đó tồn tại trước khi sự tương tác bắt đầu và vẫn tiếp tục tồn tại sau khi tương tác kết thúc, thì *đường sinh tồn* của nó sẽ chạy từ trên xuống dưới biểu đồ.

Các qui ước về tên của các thể hiện đối tượng được áp dụng tương tự như biểu đồ cộng tác. Bảng 8.1 trong chương 8 được viết lại thành bảng 9.1.

Cú pháp	Giải thích
o	Một đối tượng tên o.
o:C	Đối tượng o của lớp C.
:C	Đối tượng vô danh của lớp C.
/R	Đối tượng vô danh đóng vai trò R.
/R:C	Đối tượng vô danh của lớp C đóng vai trò R.
o/R	Đối tượng o đóng vai trò R.
o/R:C	Đối tượng o của lớp C đóng vai trò R.

Bảng 9.1: Cú pháp tên của thể hiện

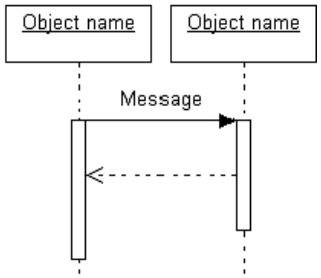


Hình 9.7: Ký hiệu của stimulus

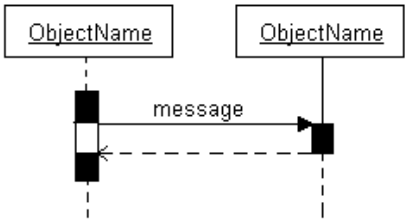
Trong biểu đồ tuần tự, một thông điệp hoặc một *tác nhân kích thích* (stimulus) được mô tả bằng một mũi tên từ đối tượng gởi đến đối tượng nhận, xem hình 9.7.

Mũi tên được đặt nhãn là thông điệp hoặc kích thích (stimulus) đang được gởi. Từ lúc này, để đơn giản chúng ta sẽ dùng chung thuật ngữ thông điệp.

Mũi tên xuất phát từ vùng *tiêu điểm kiểm soát* của đối tượng gởi đến *tiêu điểm kiểm soát* của đối tượng nhận. *Tiêu điểm kiểm soát* được biểu diễn bằng một hình chữ nhật nhỏ đặt trên *đường sinh tồn* của đối tượng. Hình chữ nhật này không phải lúc nào cũng hiển thị trong biểu đồ tuần tự. *Tiêu điểm kiểm soát* cho biết đối tượng nào hiện đang điều khiển sự tương tác bởi nó đang tự thực hiện một vài tác vụ hoặc gởi một thông điệp cho đối tượng khác. Ý tưởng *tiêu điểm kiểm soát* chỉ áp dụng khi các thông điệp được gởi là các lời gọi thủ tục từ một đối tượng đến một đối tượng khác; các lời gọi này là đồng bộ. Trong tương tác không đồng bộ, một đối tượng có thể gởi một thông điệp đến đối tượng khác, và sau đó đối tượng đầu tiên sẽ tiếp tục thực hiện tác vụ kế tiếp mà không cần phải đợi phản hồi; trong một tương tác đồng bộ, mỗi đối tượng phải đợi phản hồi. Việc trả điều khiển về trong tương tác thủ tục được biểu diễn bằng một đường mũi tên đứt nét quay về đối tượng gọi, như hình 9.8.

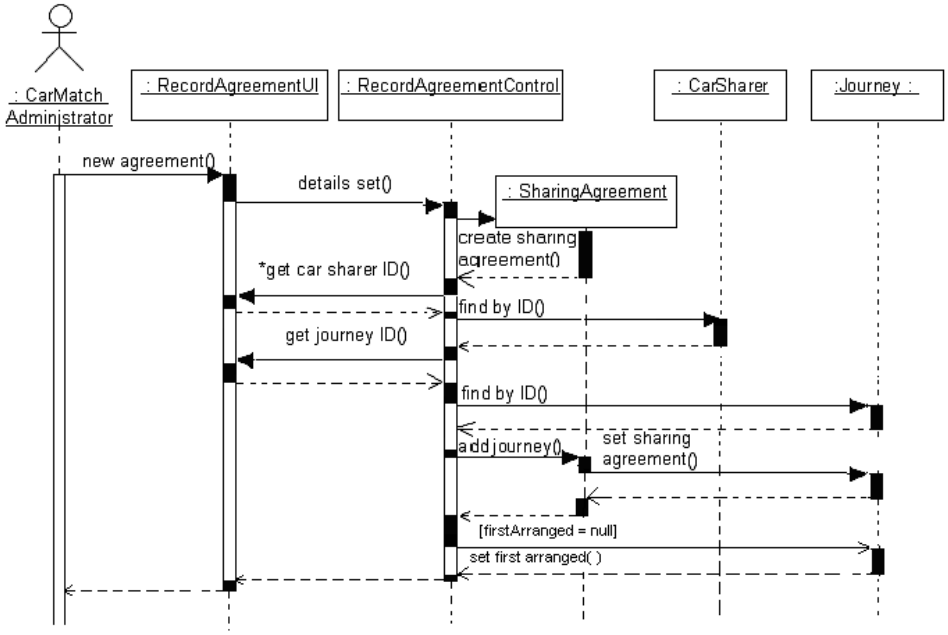


Hình 9.8: Ký hiệu trả về



Hình 9.9: Ký hiệu hoạt động

Thời gian hoạt động của một đối tượng nào đó có thể được mô tả bằng cách tô đen lên vùng *tiêu điểm kiểm soát*, như trong hình 9.9. Hình 9.10 biểu diễn biểu đồ tuần tự của hình 9.2 với các giá trị trả về tường minh và các *vùng hoạt động* được tô đen. Trong một chương trình đang chạy có dùng các lời gọi thủ tục, những vùng được tô biểu diễn các thời điểm *mã trình* đang chạy bên trong đối tượng có *đường sinh tồn* làm chủ *tiêu điểm kiểm soát* đó.

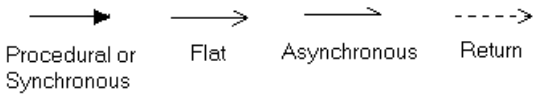


Hình 9.10: Biểu đồ tuần tự với các trả điều khiển tường minh và các vùng hoạt động được tô

Các hình chữ nhật *tiêu điểm kiểm soát* trong các đối tượng bên phải của biểu đồ không được duy trì bởi vì chúng chỉ hoạt động trong một khoảng thời gian rất ngắn; giao diện người dùng và các đối tượng điều khiển luôn hoạt động và chuyển giao trách nhiệm thực hiện những tác vụ cho các đối tượng khác.

Biểu đồ tuần tự vẽ như thế này cung cấp một hình ảnh rõ hơn nhiều so với biểu đồ cộng tác tương đương (hình 9.1). Tuy nhiên, các biểu đồ tuần tự cũng có thể trở nên rối rắm với bừa bộn các đường và trở nên khó hiểu.

Không phải tất cả các thông điệp đều đồng bộ, và giống với biểu đồ cộng tác, các kiểu mũi tên khác nhau được dùng để biểu diễn các loại thông điệp khác nhau, xem hình 9.11.



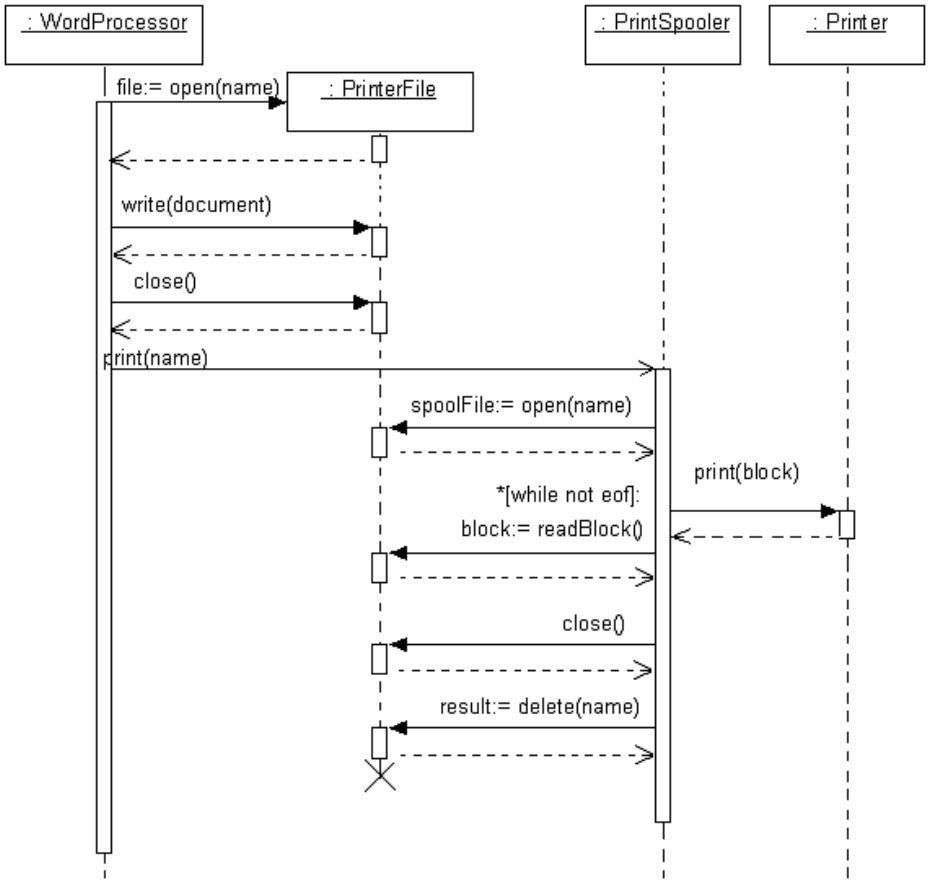
Hình 9.11: Ký hiệu dòng thông điệp

- **Procedural or Synchronous** (thủ tục hoặc đồng bộ): Một thông điệp được gởi đi từ một đối tượng đến đối tượng khác và đối tượng đầu sẽ đợi cho đến khi kết quả hoàn tất. Kể cả việc đợi cho đến khi nào hoàn tất hành động được gọi bởi đối tượng thứ hai trên những đối tượng khác.
- **Flat**: Mỗi mũi tên đại diện cho một tiến trình tuần tự từ bước này đến bước khác. Một thông điệp *flat* có thể được sử dụng khi không biết một thông điệp có là đồng bộ hay không.
- **Asynchronous** (không đồng bộ): Một thông điệp được gởi từ một đối tượng đến đối tượng khác, nhưng đối tượng đầu không đợi cho đến khi hành động hoàn tất mà nó sẽ thực hiện bước tiếp theo trong chuỗi hành động của nó.
- **Return** (trở về): Biểu diễn việc trả điều khiển tường minh đến đối tượng gởi thông điệp. Cách biểu diễn này ít được dùng trong biểu đồ cộng tác (xem chương 8)

Mũi tên biểu diễn dòng điều khiển. Khi dòng là *procedural or synchronous*, sẽ chỉ có một dòng thi hành, và hoạt động sẽ truyền từ đối tượng này đến đối tượng khác. Khi dòng là *asynchronous*, thì có thể có nhiều hơn một đối tượng được kích hoạt đồng thời (bất cứ khi nào). Hình 9.12 biểu diễn biểu đồ cộng tác của ví dụ trong mục 8.5.9 bằng biểu đồ tuần tự. Lưu ý rằng một thông điệp *asynchronous* được gởi đi giữa các đối tượng hoạt động, là :*WordProcessor* và :*PrintSpoole*, những đối tượng này được vẽ bằng một đường viền dày (ký hiệu các đối tượng hoạt động được trình bày trong hình 8.30). Đối tượng :*Printer* cũng được biểu diễn như một đối tượng hoạt động, nhưng chúng ta vẽ nó ở đây với một trả về tường minh để phản hồi rằng nó đã in một *khối* (block) trước khi nó thực hiện công việc kế tiếp.

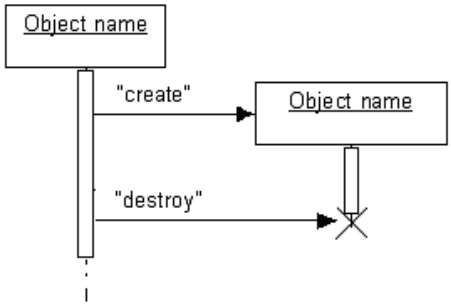
Biểu đồ này cũng minh hoạ việc huỷ một đối tượng bằng cách dùng ký hiệu 'X' ngay cuối *đường sinh tồn* của đối tượng nơi nó nhận được thông điệp *huỷ*. Đối tượng :*PrinterFile* được tạo và huỷ trong quá trình tương

tác. Trong ví dụ này, 'X' được hiển thị ở cuối vùng *tiêu điểm kiểm soát*, khi :*PrinterFile* phải thực hiện một vài hành động để tự xoá nó và giải phóng tài nguyên của hệ thống, nhưng nó có thể được đặt trên *đường sinh tồn* nếu như *hành động huỷ* chỉ đơn giản là huỷ đối tượng. Cũng trong trường hợp này, chúng ta đặc tả việc xoá là đồng bộ, khi :*PrinterFile* chuyển điều khiển trở lại cho :*PrintSpooler* một cách tường minh. Các thông điệp tạo và huỷ các đối tượng có thể được tạo khuôn dạng là <<*create*>> và <<*destroy*>>, xem hình 9.13.

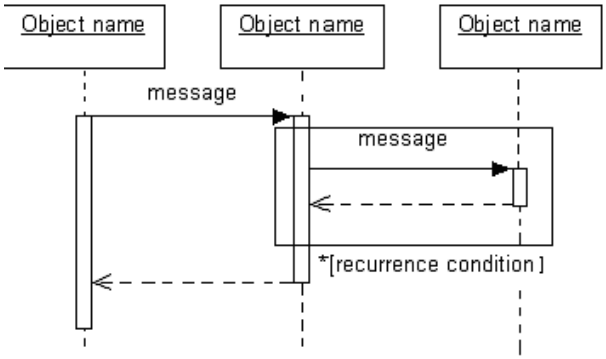


Hình 9.12: Biểu đồ tuần tự với thông điệp không đồng bộ và đối tượng hoạt động

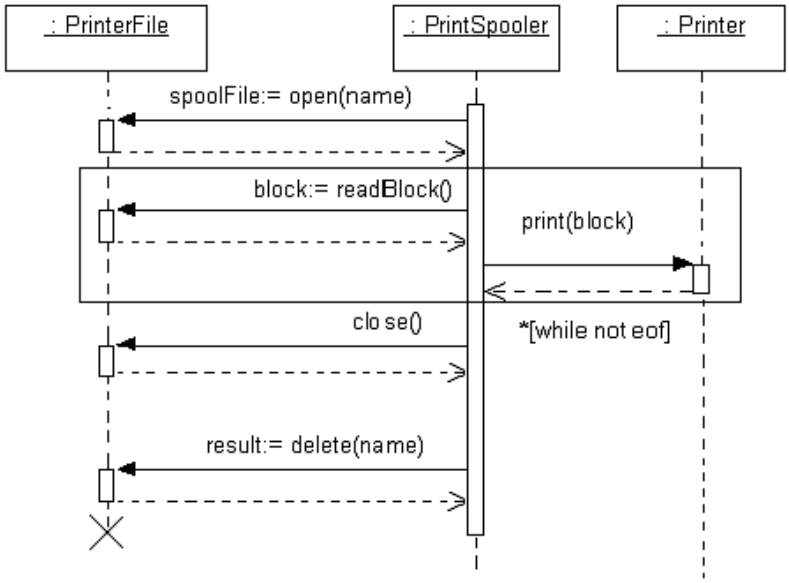
Khi có dãy thông điệp được đặt trong vòng lặp (iteration) chúng được nhóm lại trong một hình chữ nhật với điều kiện lặp được đặt ở đáy, xem hình 9.14. Đặc tả này là không rõ ràng về việc điều kiện lặp nên có dấu hoa thị hay không. Chúng ta nên thêm dấu hoa thị vào điều kiện lặp để cho biết những hành động này có thể được lặp lại.



Hình 9.13: Ký hiệu việc tạo và huỷ một đối tượng



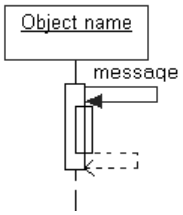
Hình 9.14: Ký hiệu lặp



Hình 9.15: Biểu đồ tuần tự lặp với một nhóm các thông điệp

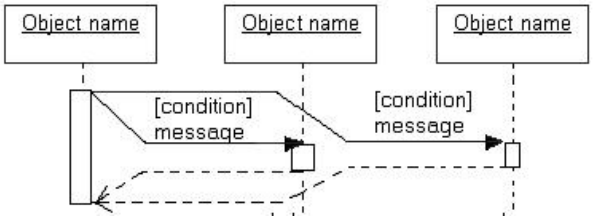
Hình 9.15 cho thấy :*PrintSpooler* sẽ lặp lại việc đọc một *khối* từ :*PrinterFile* và ghi nó vào :*Printer* cho đến khi nào đọc đến cuối tập tin.

Một đối tượng có thể gởi thông điệp cho chính nó hoặc gọi một trong những thao tác của nó. Điều này được biểu diễn tường minh trong vùng *tiêu điểm kiểm soát* bằng cách đặt một hình chữ nhật khác lên hình chữ nhật hiện có và đặt hơi lệch về phía bên phải, xem hình 9.16.



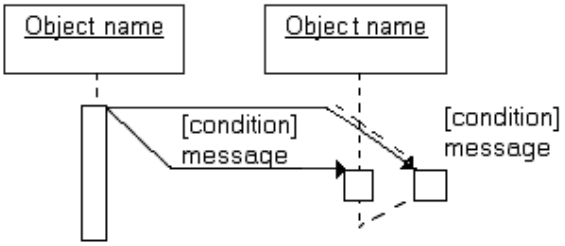
Hình 9.16: Ký hiệu đệ qui

Việc phân nhánh được hiển thị trong biểu đồ tuần tự bằng hai mũi tên tách ra từ một điểm. Một mệnh đề điều kiện được thêm vào thông điệp để hiển thị điều kiện nhằm quyết định nên theo nhánh nào, hình 9.17.



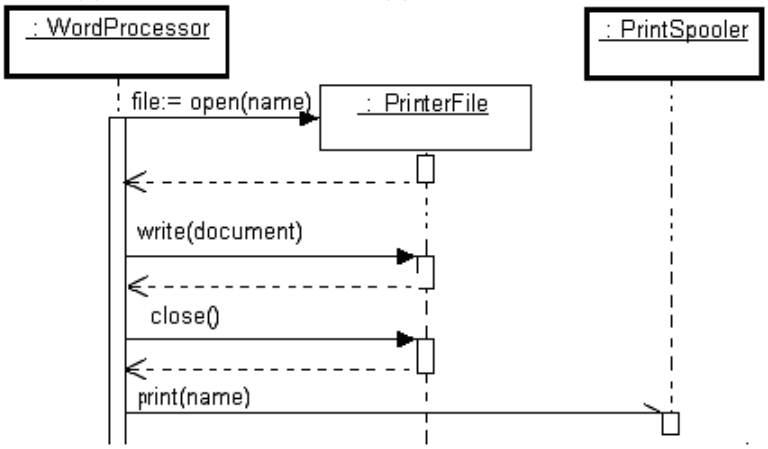
Hình 9.17: Ký hiệu rẽ nhánh

Khi một nhánh dẫn đến việc chọn lựa hai thông điệp khác nhau đều gởi đến một đối tượng, thì *đường sinh tồn* của đối tượng được chia ra thành hai vùng *tiêu điểm kiểm soát* khác nhau. Hai *đường sinh tồn* riêng biệt này sẽ được hợp lại sau khi đã hoàn tất các hành động, xem hình 9.18.

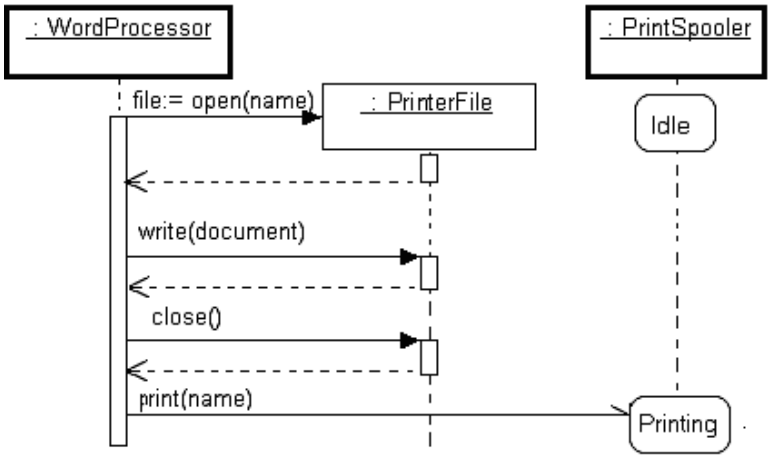


Hình 9.18: Ký hiệu các nhánh đến cùng một lifeline

Chi tiết của tương tác có thể bị che dấu trong biểu đồ tuần tự. Một thông điệp được gởi đến một đối tượng, kết quả là các thông điệp được gởi đến những đối tượng xa hơn, việc hiển thị tất cả các chi tiết chỉ làm cho biểu đồ thêm chật chội hoặc bừa bộn mà thôi. Chi tiết của cách đối tượng thực hiện phản hồi của nó cho các thông điệp có thể được biểu diễn trong một biểu đồ riêng. Biểu đồ trong hình 9.12 có thể có chi tiết cách hoạt động của *Spooler*, xem hình 9.19. Chi tiết hoạt động của *Spooler* được trình bày trong hình 9.15.



Hình 9.19: Biểu đồ tuần tự với các chi tiết được loại bỏ

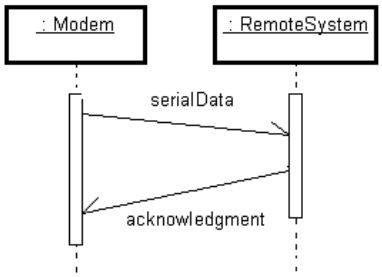


Hình 9.20: Biểu đồ tuần tự với các trạng thái được bổ sung

Theo Rumbaugh (1999), trạng thái của đối tượng có thể được đặt trên một *đường sinh tồn* để cho biết trạng thái hiện tại của nó. Trạng thái sẽ

được giải thích chi tiết hơn trong chương 11. Hình 9.20 trình bày ví dụ về hai trạng thái được đặt trên một *đường sinh tồn*. *Spooler* ở trong trạng thái *idle* (rỗi) cho đến khi nó nhận chỉ thị in, lúc này nó chuyển sang trạng thái *printing*. Tuy nhiên, ký hiệu này không có trong đặc tả UML trên các biểu đồ tuần tự.

Việc gởi các thông điệp thường được giả định là xảy ra ngay lập tức. Tuy nhiên, trong một tình huống mà ở đó thông điệp được gởi thông qua một liên kết truyền thông và một lượng thời gian đáng kể trôi qua giữa quá trình gửi và quá trình nhận thông điệp, thì mũi tên có thể được biểu diễn xiên xuống như hình 9.21.



Hình 9.21: Các thông điệp mất nhiều thời gian để đến đích

9.4.2 Thông điệp

Cú pháp thông điệp của biểu đồ tuần tự về cơ bản giống với cú pháp thông điệp của biểu đồ cộng tác. Chi tiết về cú pháp đã được trình bày trong chương 8. Những khác biệt giữa việc dùng các thông điệp trong biểu đồ tuần tự so với biểu đồ cộng tác sẽ được trình bày dưới đây.

Mức chi tiết về thông điệp được hiển thị kèm theo mũi tên có thể khác nhau. Ở mức đơn giản nhất chỉ có tên của thông điệp. Trong các biểu đồ tuần tự, số thứ tự ít khi được thêm vào vì việc sắp đặt các thông điệp đã cho biết thứ tự của chúng.

Một cách tổng quát, cú pháp của một thông điệp có dạng như sau:

predecessor guard-condition sequence-expression return-value := message-name argument-list

Tất cả những thành phần này có thể bỏ đi, nhưng *message-name* thường được hiển thị. *Return-value*, *message-name* và *argurment-list* được gọi là *chữ ký* (signature) của thông điệp.

- **Message-name** (tên thông điệp): là biến cố gởi đến đối tượng nhận. Đây có thể là tên của một biến cố hoặc của một thao tác trong lớp của đối tượng nhận, ví dụ *setFirstArranged*.

Thuật ngữ *signal* (tín hiệu) được dùng để ám chỉ đến các biến cố được gởi không đồng bộ từ đối tượng này đến một đối tượng khác. Một *signal* có thể có các thuộc tính. Ví dụ, tín hiệu *MouseDown* có thể có các thuộc tính như *button, xyCoordinate* và *timestamp*.

- **Argument-list** (danh sách đối số): Thao tác và biến cố được định nghĩa duy nhất nếu danh sách đối số được đặc tả đầy đủ. Nó là một danh sách chứa tên các đối số được ngăn cách với nhau bằng dấy phẩy, ví dụ như *SetArranged*(*dateArranged*)**,** *MouseDown*(*button,xyCoordinate,timestamp*).
- **Return-value** (trị trả về): Biến cố là không đồng bộ và không có giá trị trả về. Các thao tác đồng bộ có thể có một giá trị trả về được gởi trở lại cho đối tượng gởi thông điệp, ví dụ *journeyName := getJourneyName* (*i*).
- **Sequence-expression** (biểu thức tuần tự): Biểu thức tuần tự định nghĩa thứ tự xảy ra của các tương tác. Nó là một danh sách các *sequence-term* (số hạng tuần tự) cách nhau bởi dấu chấm và kết thúc bởi dấu hai chấm.

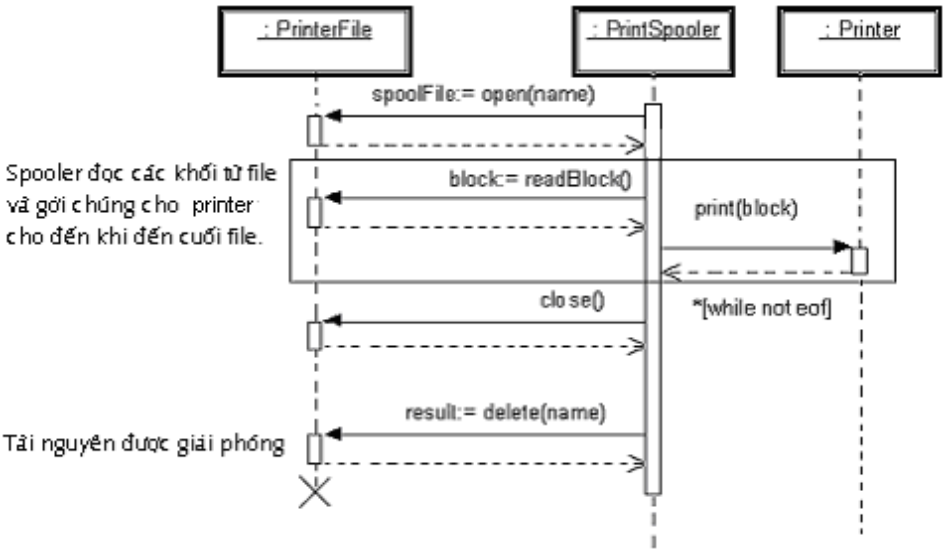
  Mỗi *sequence-term* gồm một số nguyên hoặc tên với *recurrence* (số lần thực hiện) tuỳ chọn. Số hiếm khi được dùng trong biểu đồ tuần tự (xem lại chương 8 để biết thêm chi tiết), tên thì thỉnh thoảng được dùng trong kết hợp với các *ràng buộc*. (Xem phần 9.4.3). *Recurrence* được dùng để chỉ rõ các thông điệp được gởi phụ thuộc vào một vài điều kiện hoặc có tính chất lặp, ví dụ [*dateArranged = null*] hoặc *[*i* : *0..journeys.length*].
- **Predecessor** (tiền bối): *Predecessor* của một thông điệp là một danh sách các *sequence-number* (số thứ tự) cách nhau bởi dấu phẩy được đặt sau một dấu gạch chéo. Predecessor được sử dụng trong biểu đồ tuần tự ít hơn so với biểu đồ cộng tác, bởi vì *sequence-number* hiếm khi được sử dụng, nhưng có thể được sử dụng để biểu diễn sự đồng bộ của các biến cố.
- **Guard-condition** (điều kiện bảo vệ): *Guard-condition* là một điều kiện phải thoả trước khi thông điệp được gởi đi. Mục *ký hiệu UML* trên biểu đồ tuần tự không định nghĩa *guard-condition*, nhưng nó được định nghĩa trong ngữ cảnh của *biểu đồ trạng thái*, xem chương 11. Một *guard-condition* là một biểu thức *boolean* thường được viết bằng ngôn *ngữ ràng buộc đối tượng OCL* (Object Constraint Language), điều kiện này phải thoả để một chuyển tiếp xảy ra, ví dụ [*percentageComplete = 100*]. Trong biểu đồ tuần tự, nếu không có số thứ tự thì không thể phân biệt giữa *guard-*

condition và *condition-clause* của một *recurrence* (trừ khi cả hai đều có mặt, trong trường hợp này *guard-condition* đến trước).

9.4.3 Chú thích

Biểu đồ tuần tự được chú thích theo ba cách *giải thích*, *ràng buộc* và *thời gian*

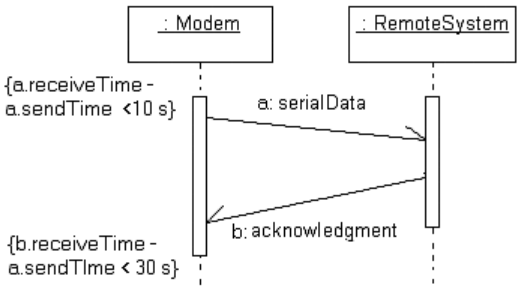
Giải thích (comment): Thường được thêm vào cột riêng bên trái như trong hình 9.22.



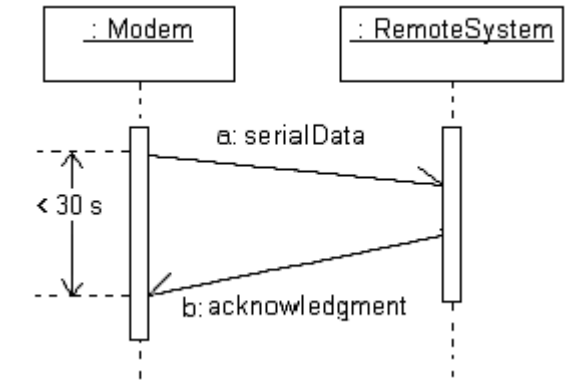
Hình 9.22: Biểu đồ tuần tự với các giải thích

Ràng buộc: Thường là các ràng buộc thời gian. Chúng được dùng để tính thời gian của một thông điệp hoặc khoảng thời gian chuyển đổi giữa các thông điệp. Có nhiều thao tác chuẩn như *sendTime* và *executionStartTime*. Hình 9.23 là một ví dụ minh họa, có sử dụng tên thông điệp để phân biệt.

Khoảng thời gian: Thời gian hoạt động hoặc thời gian tồn tại giữa các thông điệp có thể được biểu diễn bằng các ký hiệu đánh dấu dùng trong xây dựng (construction mark - giống như trong các bản thiết kế kỹ thuật), mặc dù nếu đường thẳng biểu diễn thông điệp được đặt nằm ngang thì thời gian này áp dụng cho thời gian gởi hay nhận là không rõ ràng. Hình 9.24 là một ví dụ.



Hình 9.23: Biểu đồ tuần tự với ràng buộc thời gian



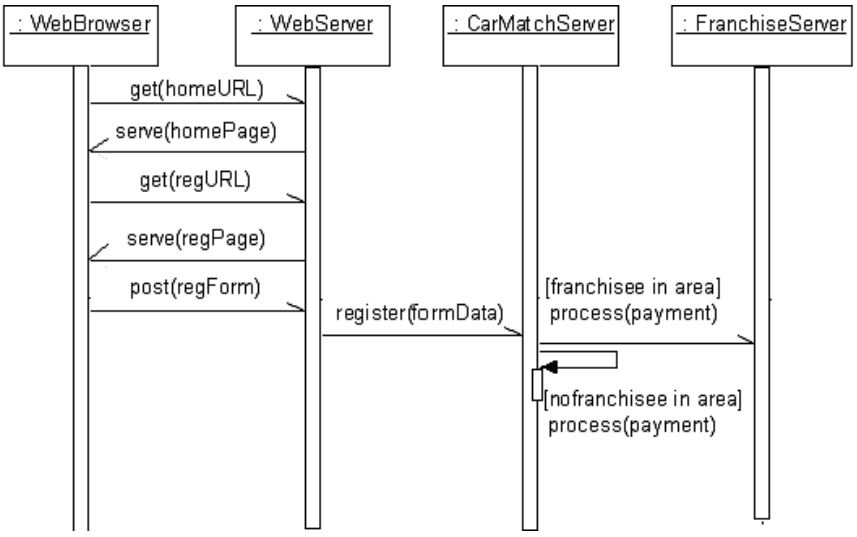
Hình 9.24: Biểu đồ tuần tự với khoảng thời gian

9.5 Cách tạo ra các biểu đồ tuần tự

Biểu đồ tuần tự được dùng để lập mô hình tương tác ở mức cao giữa người dùng và hệ thống, giữa hệ thống này với hệ thống khác, hoặc giữa các hệ thống con với nhau (còn được gọi là *biểu đồ tuần tự hệ thống*). Trong các biểu đồ tuần tự hệ thống, các đối tượng tham gia được vẽ thành các đối tượng hoạt động, như trong hình 9.25. So sánh biểu đồ hoạt động này với biểu đồ hoạt động trong hình 10.1, ta thấy rằng mục đích phục vụ là giống nhau, nhưng biểu đồ trong hình 10.1 tập trung vào hoạt động được thực hiện hơn là tập trung vào các thông điệp được gởi giữa các đối tượng tham gia.

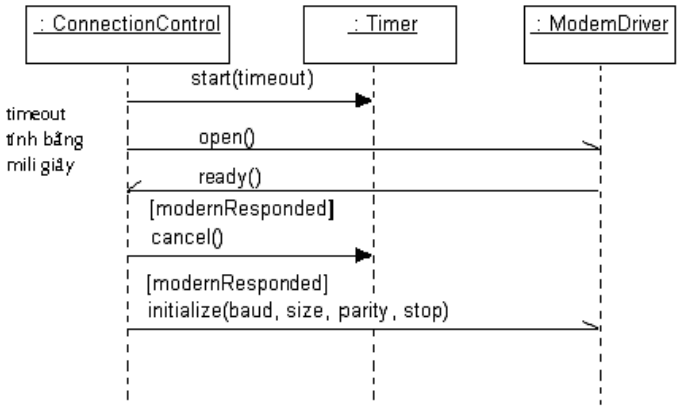
Các biểu đồ tuần tự cũng được dùng để lập mô hình tương tác xảy ra trong một cộng tác nhằm thực hiện một use case hoặc một thao tác. Các ví dụ chúng ta dùng từ trước đến nay thuộc vào loại này.

Biểu đồ tuần tự ở hình thức thứ hai có thể được vẽ như *biểu đồ thể hiện* (instance diagram) hoặc *biểu đồ tổng quát* (generic diagram). Một biểu đồ thể hiện biểu diễn một thể hiện đặc biệt của tương tác, tương đương với một *scenario* trong use case.



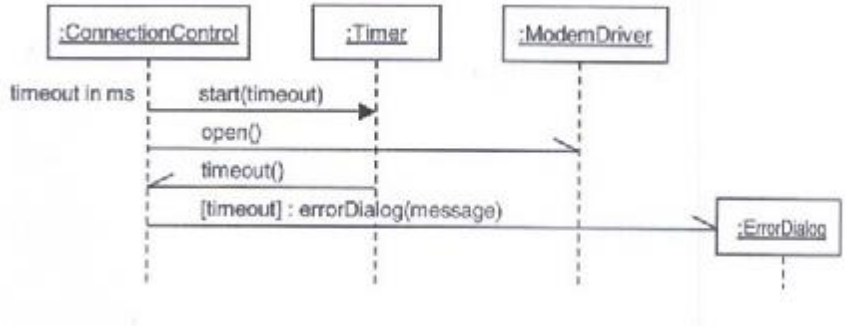
Hình 9.25: Biểu đồ tuần tự hệ thống ở mức cao

Có thể có vài *biểu đồ tuần tự thể hiện*, mỗi biểu đồ là một nhánh tương tác. *Biểu đồ tuần tự tổng quát* biểu diễn sự kết hợp các nhánh này. Điều này tương tự với sự khác nhau giữa biểu đồ cộng tác ở mức thể hiện với biểu đồ cộng tác ở mức đặc tả.



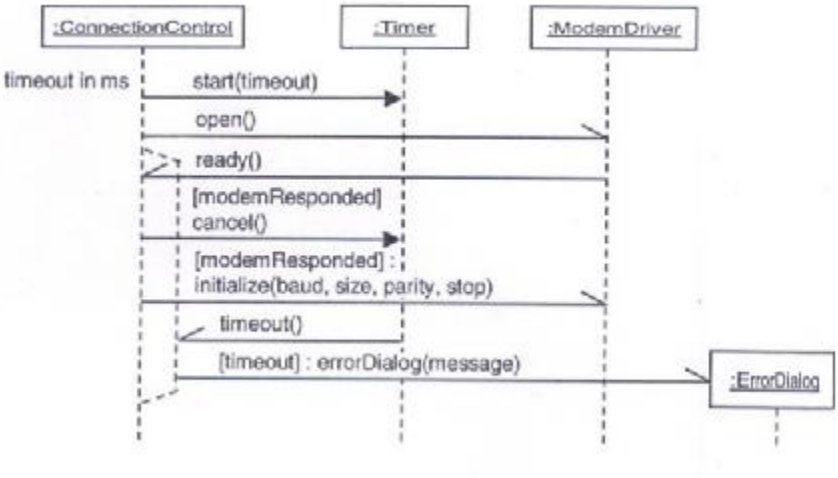
Hình 9.26: Biểu đồ tuần tự thể hiện *cho trường hợp thứ nhất*

Hình 9.26 và 9.27 trình bày hai biểu đồ thể hiện của một ứng dụng, trong ứng dụng này đối tượng *ConnectionControl* phải mở một kết nối với modem. Nếu modem không phản hồi trong một khoảng thời gian thì hoạt động này *hết thời hạn* (time out) và một thông điệp lỗi được hiển thị. Thể hiện đầu tiên biểu diễn một sự kết nối thành công và thể hiện thứ hai biểu diễn biến cố *time out* xảy ra.



Hình 9.27: Biểu đồ tuần tự mức thể hiện cho trường hợp thứ hai

Hình 9.28 minh họa 2 thể hiện được kết hợp trong một biểu đồ tổng quát.



Hình 9.28: Biểu đồ tuần tự tổng quát cho cả hai trường hợp

Các bước để vẽ một biểu đồ tuần tự tương tự như các bước vẽ một biểu đồ cộng tác:

- Quyết định ngữ cảnh của tương tác: hệ thống, hệ thống con hoặc thao tác.

- Xác định các thành phần có cấu trúc (lớp hoặc đối tượng) cần thiết để thực hiện chức năng của use case hoặc thao tác.
- Xem xét các *scenario* thay thế.
- Vẽ các biểu đồ thể hiện:
  - Đặt các đối tượng từ trái sang phải.
  - Bắt đầu bằng thông điệp khởi đầu tương tác, đặt các thông điệp theo chiều từ trên xuống dưới. Biểu diễn các đặc tính của các thông điệp cần thiết để giải thích ngữ nghĩa của tương tác.
  - Thêm *tiêu điểm kiểm soát* nếu cần thiết để trực quan hoá các hành động lồng nhau hoặc thời điểm hoạt động đang diễn ra.
  - Thêm các ràng buộc thời gian nếu cần.
  - Thêm các giải thích vào biểu đồ nếu cần, ví dụ điều kiện đầu và điều kiện cuối.
- Nếu cần hãy vẽ một biểu đồ tổng quát để tóm tắt tất cả các trường hợp trong các biểu đồ thể hiện.

9.5.1 Quyết định ngữ cảnh

Biểu đồ tuần tự có thể mô hình các tương tác ở mức hệ thống, hệ thống con, use case hoặc thao tác. Tuỳ vào giai đoạn phát triển của dự án và tuỳ vào tác vụ đang được thực hiện sẽ xác định mức nào được lập mô hình.

Ví dụ 9.1

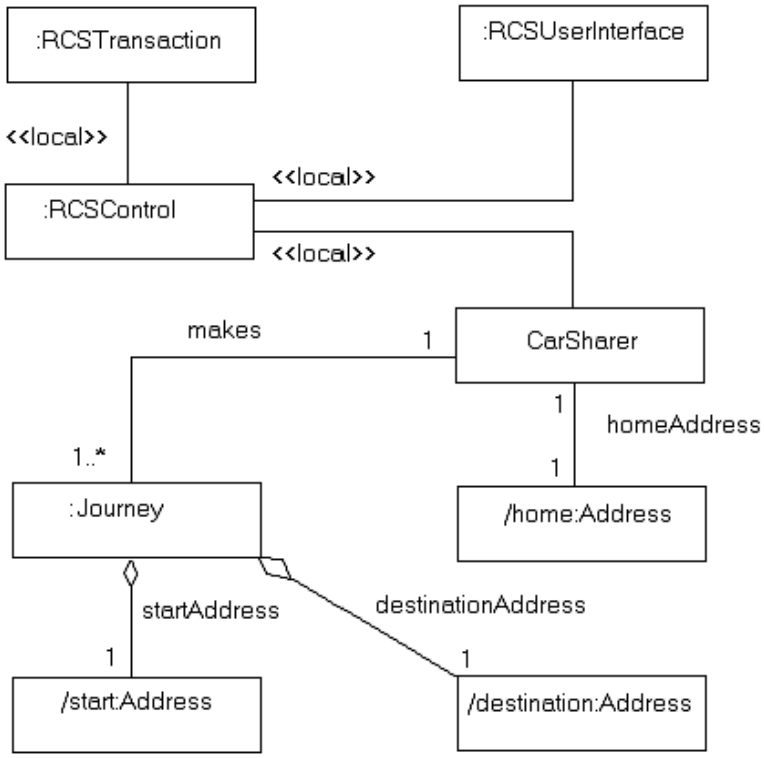
Trong ví dụ này, chúng ta đang lập mô hình cho thao tác khởi tạo một thể hiện mới của lớp *Journey* trong hệ thống *CarMatch*.

9.5.2 Xác định các thành phần cấu trúc

Bước này được giải thích chi tiết trong phần 8.6.2. Một tương tác xảy ra trong ngữ cảnh của một cộng tác, các lớp và các đối tượng tham gia vào cộng tác này nên được nhận biết trước đó. Nếu chúng ta xem xét sự cộng tác cho use case *Regsiter car sharer*, chúng ta có thể thấy cái gì sẽ xảy ra khi một *Journey* được tạo.

Ví dụ 9.2

Khi một *Journey* được tạo, nó sẽ tạo ra hai thể hiện *Address*, một là *địa chỉ xuất phát* (start address) và một là *địa chỉ đích* (destination address), xem hình 9.29.



Hình 9.29: Cộng tác cho use case Register car sharer

9.5.3 Xem xét các *scenario* thay thế

Chúng ta nên kiểm tra các cộng tác để xem có use case nào khác mà ở đó *Journey* được tạo hay không. Trong trường hợp này thì không. Chúng ta cũng xem xét các nhánh khác có thể có của thao tác ứng với các thông tin đầu vào khác nhau.

Ví dụ 9.3

Đối với use case *Register car sharer*, có hai *scenario* khác nhau. Thứ nhất, các địa chỉ đã được định vị trên bản đồ địa lý trên server, không cần định vị lại. Thứ hai, chúng chưa được định vị và điều này phải được thực hiện khi các đối tượng *Address* mới được tạo ra.

9.5.4 Vẽ các biểu đồ thể hiện

Bây giờ chúng ta bắt đầu quá trình vẽ các biểu đồ thể hiện cho hai scenario đã được phát triển. Chúng ta sẽ giải thích chi tiết trường hợp đầu tiên.

9.5.4.1 Bố trí các đối tượng

Đặt các thông điệp từ trái sang phải, bắt đầu từ đối tượng nhận thông điệp làm phát sinh tương tác này. Nếu lập mô hình một use case, có thể có các đối tượng *interface*, các đối tượng này nên được đặt ở bên trái.

Ví dụ 9.4

Trong trường hợp này, không có các đối tượng *interface*. Ở đây một thể hiện của *Journey* nhận thông điệp này. Nó tạo hai thể hiện *Address*. Các đối tượng này được trình bày trong Hình 9.30.

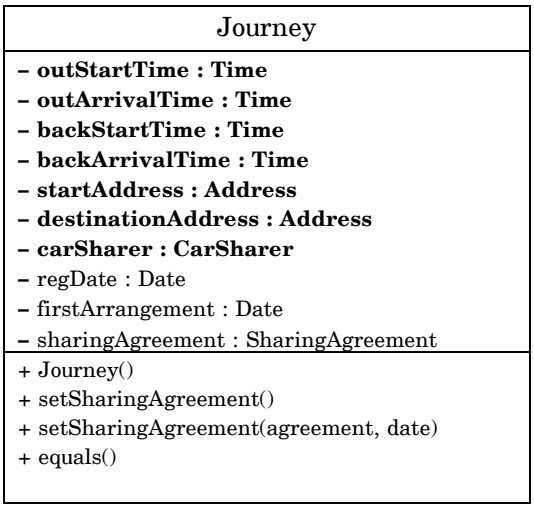
Hình 9.30: Các đối tượng liên quan trong khởi tạo Journey

9.5.4.2 Bố trí các thông điệp

Các thông điệp được sắp xếp từ trên xuống. Chúng ta muốn thêm bao nhiêu chi tiết trong thông điệp cũng được. Nên trình bày đủ để khi nhìn vào biểu đồ có thể hiểu được biểu đồ và hiểu được cách nó làm việc.

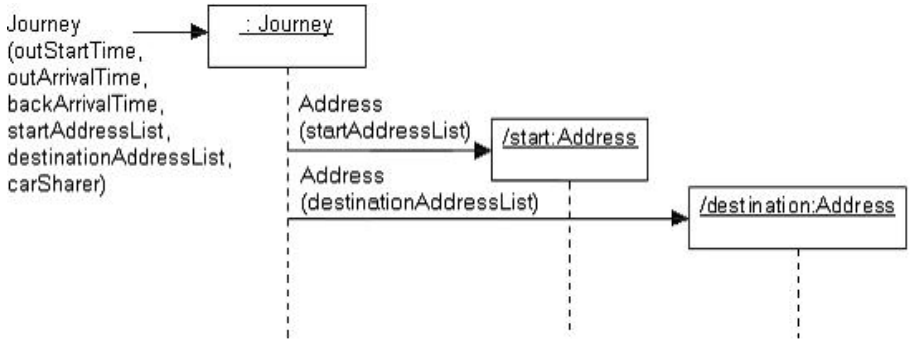
Ví dụ 9.5

Thể hiện *Journey* được tạo ra sẽ nhận thông điệp. Đến lượt, nó tạo ra hai thể hiện *Address*. Mỗi thể hiện được tạo ra bởi hàm khởi tạo. Chúng ta trình bày tất cả các thông điệp này theo khuôn dạng <<*create*>>, do đó sẽ không chuyển tải được nhiều thông tin về cách làm việc của nó.



Hình 9.31: Lớp Journey (với các tham số khởi tạo được in đậm)

Journey được truyền một số tham số khi nó được tạo. Nếu nhìn vào định nghĩa lớp, chúng ta có thể thấy được các giá trị nào được thiết lập khi một *Journey* được tạo. Điều này được trình bày trong hình 9.31, trong đó các thuộc tính bắt buộc được in đậm.



Hình 9.32: Các thông điệp cho các thao tác

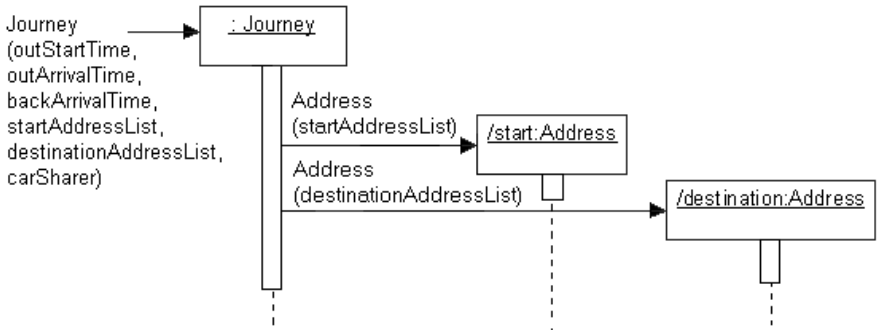
Các thuộc tính *Address* phải được tạo. Dữ liệu được truyền vào để tạo một thể hiện mới *Address* là một danh sách các giá trị. Hình 9.32 biểu diễn các thông điệp cùng với các tham số. Chúng là các thao tác đồng bộ.

9.5.4.3 Thêm focus of control

Vùng *focus of control* có thể được thêm vào nếu cần.

Ví dụ 9.6

Các vùng *focus of control* không giúp cho biểu đồ dễ đọc hơn trong *scenario* này, nhưng sẽ hữu ích cho các *scenario* khác, vì vậy thêm chúng vào (hình 9.33).



Hình 9.33: thêm Focus of control *vào*

9.5.4.4 Thêm các ràng buộc về thời gian

Ràng buộc thời gian có thể được thêm vào biểu đồ (xem bài tập 9.9).

9.5.4.5 Thêm các chú thích

Mọi chú thích cần thiết đều có thể được thêm vào biểu đồ (bài tập 9.9).

9.5.4.6 Lặp lại các bước cho mỗi scenario

Các bước như vậy có thể được lặp lại cho tất cả các *scenario* khác.

Ví dụ 9.7

Trong trường hợp này, chúng ta có một *scenario* khác, ở đây các địa chỉ chưa được định vị, xem hình 9.34.

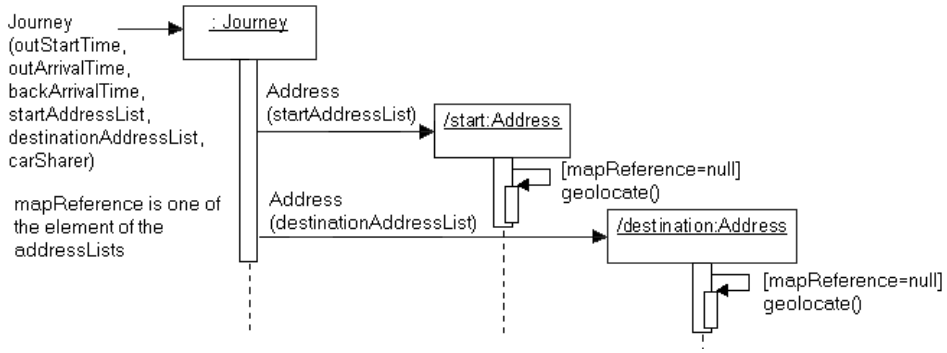
Chúng ta vừa thêm một chú thích để giải thích nguồn gốc của *mapReference*. Điều này cũng có thể được mô tả bằng một *ghi chú* (note).

9.5.5 Vẽ biểu đồ tổng quát

Bây giờ chúng ta có thể phối hợp các *scenario* khác nhau vào một biểu đồ.

Ví dụ 9.8

Trong ví dụ này, không có thêm các thông điệp hoặc các hoạt động bổ sung. Biểu đồ hình 9.34 là một biểu đồ tổng quát và không cần phải vẽ một biểu đồ riêng.



Hình 9.34: Một scenario *thay thế*

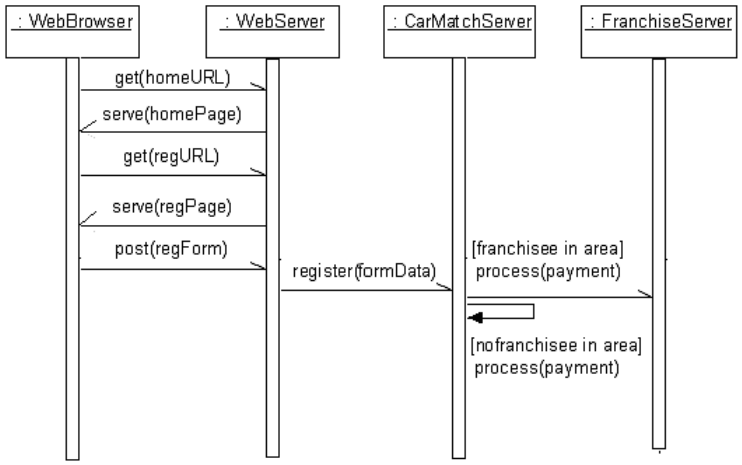
9.6 Lập mô hình nghiệp vụ với biểu đồ tuần tự

Trong *profile* về việc lập mô hình nghiệp vụ, có 5 lớp *stereotype* được định nghĩa cho các đối tượng nghiệp vụ. Đó là *Actor*, *Worker*, *Case*

Worker, *Internal Worker* và *Entity* (mục 8.7). Các biểu tượng *stereotype* có thể được sử dụng trong biểu đồ tuần tự để mô hình sự cộng tác giữa *worker* và *entity* trong việc thực hiện các use case.

Các biểu đồ tuần tự của hệ thống ở mức cao có thể được tạo ra để mô hình tương tác bên trong các use case nghiệp vụ. Hình 9.35 là một ví dụ. Các biểu đồ như thế thường được vẽ không cần đến vùng *focus of control* trên *lifeline*.

Đây là loại biểu đồ được tạo ra ở giai đoạn đầu của qui trình phát triển hệ thống, như là một phần của việc mô hình hóa nghiệp vụ, được tạo ra trước khi phân tích chi tiết được thực hiện.



Hình 9.35: Biểu đồ tuần tự hệ thống ở mức cao

9.7 Quan hệ với các biểu đồ khác

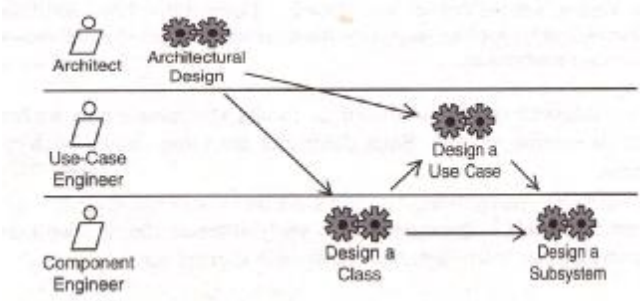
Biểu đồ tuần tự được dùng để mô hình việc hiện thực use case hoặc thao tác. Ở trường hợp thứ hai, mỗi thao tác được mô hình phải tồn tại trong một biểu đồ lớp. Tên thông điệp phải là biến cố hoặc thao tác của lớp nhận. Nếu các trạng thái được tham chiếu đến trong các điều kiện bảo vệ hoặc được hiển thị trên một *lifeline* của đối tượng, thì chúng phải là các trạng thái hợp lệ của lớp liên quan và xuất hiện trong biểu đồ trạng thái (chương 11 trình bày biểu đồ trạng thái).

Các biểu đồ tuần tự và cộng tác (chương 8) dùng để mô hình các khía cạnh như nhau: các đối tượng cộng tác với nhau và các thông điệp được trao giữa chúng để đạt được một mục tiêu nào đó. Biểu đồ tuần tự và biểu đồ cộng tác có thể chuyển đổi qua lại cho nhau. Một số công cụ CASE có thể thực hiện được điều này một cách tự động. Tuy nhiên có một số

thông tin bị mất khi chuyển đổi: biểu đồ tuần tự không hiển thị các liên kết giữa những đối tượng hoặc các kết hợp giữa những vai trò lớp; biểu đồ cộng tác không hiển thị thông tin ràng buộc thời gian.

9.8 Biểu đồ tuần tự trong UP

Trong UP, biểu đồ tuần tự được dùng đầu tiên trong *dòng thiết kế* , xem hình 9.36.



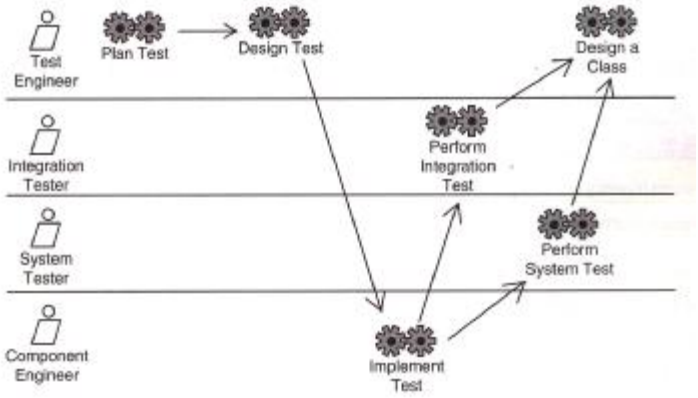
Hình 9.36: Design workflow

Các biểu đồ cộng tác được tạo ra như là một phần của *mô hình phân tích* (Analysis Model) và được tinh chế trong *thiết kế* (Design). Trong UP, ở giai đoạn thiết kế chúng ta nên dùng biểu đồ tuần tự để lập mô hình tương tác hơn là dùng biểu đồ cộng tác.

Hoạt động *thiết kế một use case* (Design a Use Case) tương tự với hoạt động *phân tích một use case*, ngoại trừ các lớp thiết kế được sử dụng và có nhiều bước hơn, cũng như chi tiết hơn khi một dự án tiến gần đến giai đoạn cài đặt. Các bước chính là: *xác định các lớp thiết kế tham gia* (Indentifying the Participating Design Classes), *mô tả các tương tác của đối tượng thiết kế* (Describing Design Object Interactions), *xác định các giao diện và các hệ thống con tham gia* (Identifying the Participating Subsystems and Interfaces) và *mô tả các tương tác của hệ thống con* (Describing Subsystem Interactions). Các lớp thiết kế có thể khác so với các lớp phân tích. Ví dụ, các lớp *collection* có thể được thêm vào trong thiết kế để quản lý các tương tác liên quan đến một tập các đối tượng. Các tương tác trên biểu đồ cộng tác và biểu đồ tuần tự sẽ phải được sửa lại để chứa thêm những thể hiện của những lớp này và những tương tác của chúng. Chúng ta cũng có thể xem xét một use case dưới dạng các tương tác giữa những hệ thống con thiết kế hơn là những đối tượng thiết kế. Điều này cung cấp một phân rã phân cấp của các tương tác, chi tiết của tương tác trong một hệ thống con có thể ẩn và được mô tả trong một biểu đồ riêng ở mức thấp hơn. Để hỗ trợ hướng tiếp cận này, các giao

diện của hệ thống con phải được xác định, làm rõ các thao tác hoặc biến cố được mỗi hệ thống con xử lý. Bước thứ năm là *nắm bắt các yêu cầu cài đặt* (Capturing Inplementation Requirments), ở đó các yêu cầu được xử lý trong cài đặt được lập sưu liệu.

Biểu đồ tuần tự cũng được dùng trong *dòng kiểm tra* (Test Workflow), hình 9.37.



Hình 9.37: Test workflow

Biểu đồ tuần tự được dùng trong hoạt động *kiểm tra thiết kế* (Design Test). Hoạt động này gồm các bước: *thiết kế các trường hợp kiểm tra tích hợp* (Designing Integration Test Cases), *thiết kế các trường hợp kiểm tra hệ thống* (Designing System Test Cases), *thiết kế các trường hợp kiểm tra hồi qui* (Designing Regression Test Cases) và *xác định và lập cấu trúc các thủ tục kiểm tra* (Identifying and Structuring Test Procedures). Biểu đồ tuần tự được dùng ở bước đầu tiên. Kiểm tra tích hợp liên quan đến việc bảo đảm rằng những thành phần khác nhau của hệ thống làm việc với nhau một cách chính xác. Hầu hết các trường hợp kiểm tra đều có thể được bắt nguồn từ việc hiện thực use case được tạo ra trong thiết kế, bởi vì chúng mô tả cách các đối tượng tương tác với nhau.

Một số công cụ kiểm tra tự động có thể dò tìm sự tương tác thông qua một *trường hợp kiểm tra* (test case) và đưa ra kết quả dưới dạng của một biểu đồ tuần tự để so sánh với các biểu đồ tuần tự được tạo trong thiết kế.

Câu hỏi ôn tập

9.1 Tương tác là gì?

9.2 Cách biểu đồ tuần tự mô hình một tương tác khác với cách biểu đồ cộng tác mô hình một tương tác như thế nào?

9.3 Giải thích cách biểu đồ tuần tự biểu diễn thời gian.

9.4 Các mục đích chính của việc sử dụng biểu đồ tuần tự là gì?

9.5 Sự khác nhau giữa một biểu đồ tuần tự tổng quát và một biểu đồ tuần tự thể hiện là gì?

9.6 Ký hiệu *lifeline* của một đối tượng là gì?

9.7 Ký hiệu vùng *focus of control* là gì?

9.8 Làm thế nào chỉ ra dãy hành động theo thủ tục của một đối tượng đang thực hiện một hoạt động hoặc tính toán nào đó.

9.9 Ký hiệu một thông điệp trong biểu đồ tuần tự là gì?

9.10 Trả điều khiển được mô tả trong biểu đồ tuần tự như thế nào?

9.11 Những tên đối tượng nào dưới đây là hợp lệ?

a returnJourney:Journey

b /returnJourney:Journey

c j/returnJourney:Journey

d *j*

9.12 Hãy cho biết bốn kiểu thông điệp được biểu diễn bởi các loại mũi tên khác nhau trong biểu đồ tuần tự?

9.13 Việc tạo một đối tượng được đặc tả như thế nào trong một biểu đồ tuần tự?

9.14 Việc huỷ một đối tượng được đặc tả như thế nào trong một biểu đồ tuần tự?

9.15 Tên được sử dụng như thế nào trong các biểu thức tuần tự?

9.16 *Recurrence* là gì?

9.17 Đối tượng hoạt động là gì?

9.18 Ký hiệu một đối tượng hoạt động là gì?

9.19 Trình bày hai cách biểu diễn phép lặp trong biểu đồ tuần tự.

9.20 Mục đích thường dùng của ràng buộc trong biểu đồ tuần tự là gì?

9.21 Trình bày các bước chính để tạo ra một biểu đồ tuần tự.

9.22 Hai *dòng công việc* UP thường sử dụng biểu đồ tuần tự là gì?

9.23 Hãy trình bày 5 bước trong hoạt động *thiết kế một use case*.

9.24 Trình bày 4 bước trong hoạt động *thiết kế kiểm tra*.

Bài tập có lời giải

9.1 *CarMatch* hoàn trả phí thành viên nếu không thể thực hiện được việc kết hợp. Dưới đây là mô tả của quá trình.

> Nếu không thể kết hợp các lộ trình của một thành viên với thành viên khác trong vòng ba tháng, thì thành viên đó có quyền yêu cầu hoàn lại phí thành viên. Chi tiết của thành viên vẫn được giữ trong hệ thống. Người này được đề nghị để lại các bản ghi lộ trình cho mục đích thống kê. Nếu được giữ, các record được bật cờ là *defunct* (không hiệu lực), bằng không chúng bị xóa thật sự. Hệ thống con *Accounts* được yêu cầu trả lại phí hoàn trả này (bằng ngân phiếu hoặc bằng thẻ tín dụng tuỳ thuộc vào cách thành viên trả lệ phí ở lần đầu tiên).

Tác vụ đầu tiên là quyết định ngữ cảnh của biểu đồ tuần tự.

Lời giải:

Đây là một use case *Refund membership fee* (hoàn lại phí thành viên). Biểu đồ tuần tự sẽ được tạo ra trong ngữ cảnh cộng tác thực hiện use case này.

9.2 Các phần tử cấu trúc của cộng tác này là gì?

Lời giải:

Cộng tác này liên quan đến các đối tượng của lớp *CarSharer* và *Journey*; cũng liên quan đến hệ thống con *Accounts*. Chúng ta xem nó là một hệ thống con trong biểu đồ tuần tự này và không mô hình chi tiết những gì xảy ra bên trong. Khi một *Journey* bị xóa, các đối tượng được nhúng bên trong nó (như *Addresses* và *MapReferences*) cũng bị huỷ theo. Chúng ta mô hình điều này bằng một biểu đồ tuần tự riêng cho thao tác huỷ *Journey* giống với cách chúng ta đã mô hình hàm khởi tạo trong phần 9.5.4. Cộng tác cũng sẽ gọi một lớp *interface* và một lớp *control*.

9.3 Có bao nhiêu scenario cho biểu đồ tuần tự này?

Lời giải:

Có hai scenario, tuỳ vào thông tin nhập từ người sử dụng:

Thứ nhất, thành viên cho phép các chi tiết lộ trình của họ được giữ lại và các lộ trình được đánh dấu huỷ.

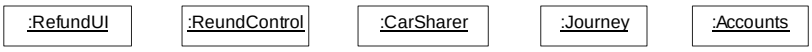
Thứ hai, chi tiết về lộ trình bị xóa.

Có thể có các scenario khác tuỳ thuộc vào cách thành viên trả lệ phí thành viên lần đầu, nhưng chúng được che dấu bên trong hệ thống con *Accounts* đối với mục đích của ví dụ này.

9.4 Bây giờ chúng ta tiến hành các bước vẽ biểu đồ tuần tự cho mỗi scenario. Trước hết chúng ta sẽ xem xét trường hợp các lộ trình được giữ lại. Bước đầu tiên là bố trí các đối tượng liên quan.

Lời giải:

Hình 9.38 trình bày các đối tượng bao hàm trong biểu đồ tuần tự này. Lưu ý các lớp *interface* và *control* được đặt ở bên trái của biểu đồ, sau đó đến đối tượng *CarSharer* và *Journey*, và cuối cùng là hệ thống con *Accounts*.



Hình 9.38: Các đối tượng trong use case Refund membership fee

Bước thứ hai là sắp xếp các thông điệp. Lúc này *focus of control* có thể được thêm vào.

Lưu ý trong một số công cụ CASE, bạn không thể tuỳ chọn thêm *focus of control* hay không. Nếu bạn đang vẽ các biểu đồ trong một gói có mục đích tổng quát (và sẽ không vẽ lại), thì hãy vẽ *focus of control* từ đầu nếu bạn có kế hoạch dùng chúng.

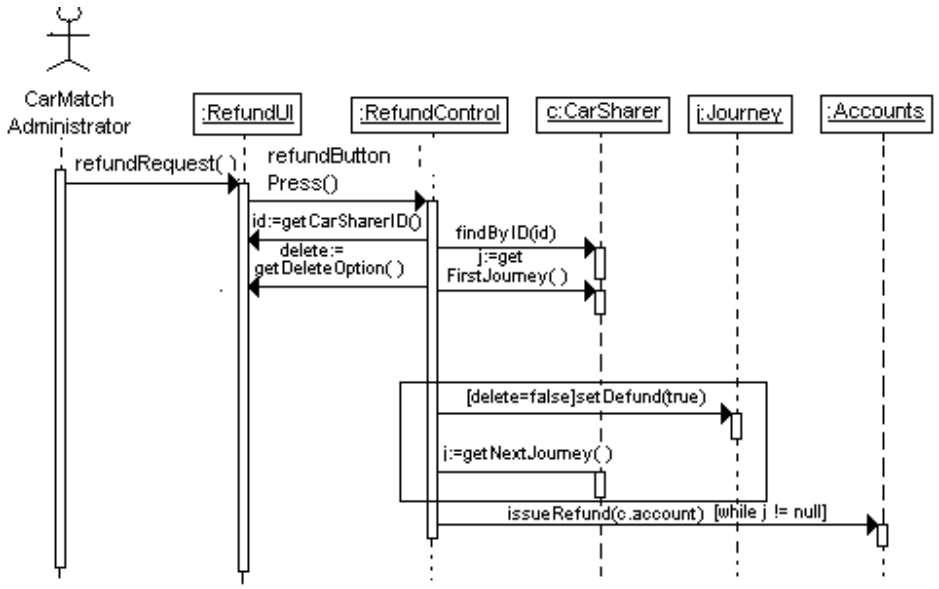
Tương tác được phát sinh bởi một biến cố từ tác nhân *CarMatch Administrator*. Có thể là một cú nhấp chuột lên nút *Refund* trên màn hình. Giả sử rằng *CarSharer* được chọn từ một danh sách và một *check box* có thể được chọn để xác định chi tiết các lộ trình của anh ta hoặc cô ta nên được giữ lại hay xóa đi. Giao diện có thể tương tự như trong hình 9.39.

Hình 9.39: Giao diện của use case Refund membership fee

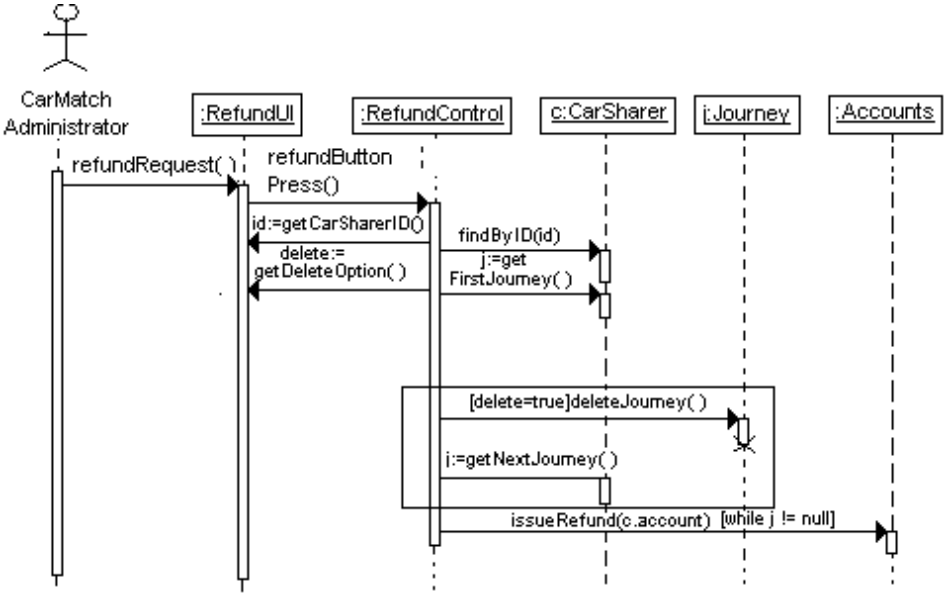
Khi người dùng nhấp chuột lên nút *Refund*, một biến cố được gởi đến đối tượng điều khiển. Dữ liệu từ đối tượng giao diện sẽ được lấy: trước tiên là *id* của *CarSharer*, *id* này phải tồn tại trong cơ sở dữ liệu; thứ hai là giá trị *boolean* cho đối tượng điều khiển biết là có nên xóa lộ trình hay không. *CarSharer* trong thiết kế này chứa một tập hợp các đối tượng *Journey*. Phải có ít nhất một, chúng ta sẽ lấy đối tượng đầu tiên và gán nó là *defunct*, và lặp lại cho tất cả các *Journey* còn lại trong tập hợp. Mỗi *CarSharer* có một liên kết đến đối tượng *Account* trong gói *Accounts*, và một thông điệp được gởi đến hệ thống con *Accounts*, yêu cầu hệ thống này hoàn trả cho *Account* hợp lệ này. Hình 9.40 biểu diễn biểu đồ tuần tự kết quả.

Cuối cùng chúng ta thêm vào các ràng buộc về thời gian và chú thích. Trong trường hợp này, không có gì để thêm.

Bây giờ chúng ta lặp lại các bước cho biến cố thứ hai. Kết quả như hình 9.41.



Hình 9.40: Các thông điệp cho tương tác trong biến cố thứ nhất

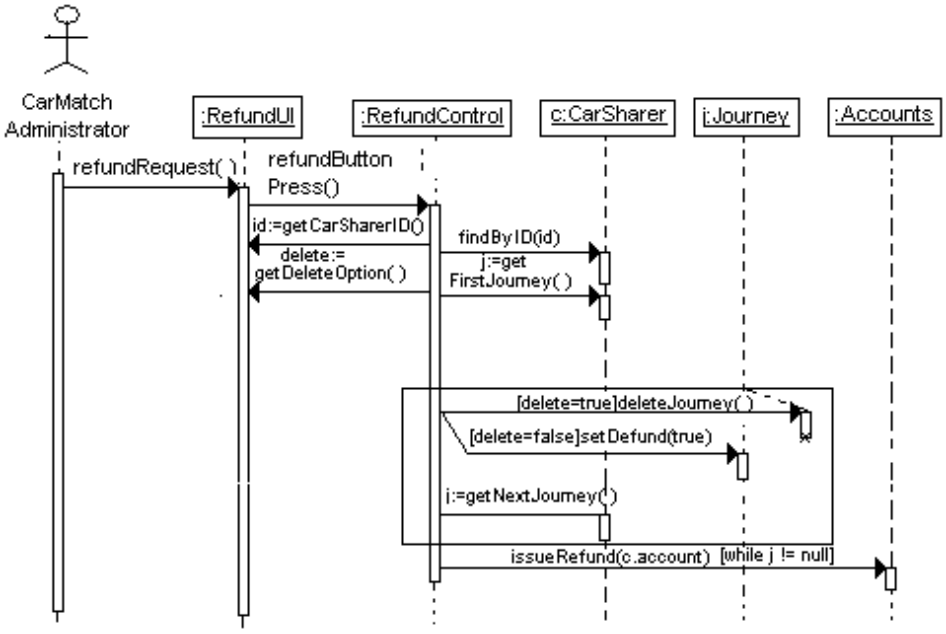


Hình 9.41: Các thông điệp cho tương tác trong trường hợp thứ hai

9.5 Bước cuối cùng là phối hợp các *scenario* vào một biểu đồ tuần tự tổng quát.

Lời giải:

Xem hình 9.42.



Hình 9.42: Biểu đồ tuần tự tổng quát

9.6 Vẽ biểu đồ tuần tự mô hình tương tác xảy ra khi một thành viên *CarMatch* lái xe qua *đèn tín hiệu tính cước* nằm trên lề đường, dùng một hệ thống tiếp sóng radio để truyền thông với đèn tín hiệu. Quá trình tương tác được mô tả như sau:

Đèn tín hiệu ở lề đường phát một tín hiệu theo chu kỳ cho mọi hệ thống tiếp sóng đi qua. Bất cứ khi nào một hệ thống tiếp sóng phản hồi, đèn tín hiệu và hệ thống tiếp sóng phải trao đổi dữ liệu trong vòng 0.25 giây. Hệ thống tiếp sóng sẽ phát *đặc điểm nhận diện* (identification) của nó đến đèn tín hiệu. Đèn tín hiệu sẽ phát ra thông tin cước phí của đoạn đường đó cho hệ thống tiếp sóng - để hiển thị thông tin trên màn hình LCD cho tài xế. Đèn tín hiệu cũng giao tiếp với cơ sở dữ liệu trung tâm để thẩm định *đặc điểm nhận diện* này. Phản hồi phải đến từ cơ sở dữ liệu trong vòng 1 giây. Nếu *đặc điểm nhận diện* không hợp lệ, đèn tín hiệu sẽ gởi một yêu cầu đến *camera* bên lề đường để chụp ảnh phương tiện khi nó đi qua.

Tác vụ đầu tiên là dựa vào ngữ cảnh của biểu đồ tuần tự.

Lời giải:

Đây là use case *Vehicle passes beacon* (xử lý phương tiện đi ngang qua đèn tín hiệu). Biểu đồ tuần tự chúng ta tạo ra ở đây là một biểu đồ ở mức cao dùng để mô hình use case này hơn là một biểu đồ chi tiết cho tất cả các đối tượng liên quan. Biểu đồ này sẽ được mô hình hóa dưới dạng hệ thống con.

9.7 Các phần tử cấu trúc của cộng tác này là gì?

Lời giải:

Cộng tác này liên quan đến 4 hệ thống con: *Transponder*, *Beacon*, *Database* và *Camera*. Ngoài ra còn có *LCDScreen* kết hợp với hệ thống tiếp sóng.

9.8 Các scenario nào cho biểu đồ tuần tự này?

Lời giải:

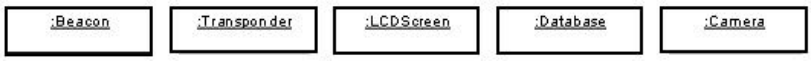
Có hai scenario, tuỳ thuộc vào kết quả trả về của cơ sở dữ liệu trung tâm.

Thứ nhất *đặc điểm nhận diện* hợp lệ; và thứ hai *đặc điểm nhận diện* không hợp lệ và một thông điệp được gởi đến camera.

9.9 Bây giờ chúng ta thực hiện từng bước vẽ các biểu đồ thể hiện cho mỗi scenario. Chúng ta sẽ xét scenario có *đặc điểm nhận diện* hợp lệ trước. Bước đầu tiên là bố trí các đối tượng có liên quan nhau.

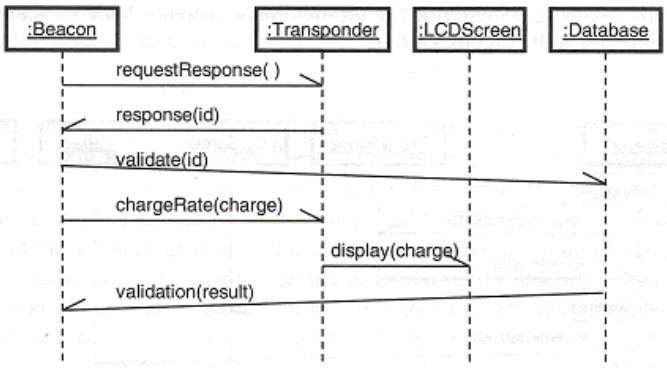
Lời giải:

Hình 9.43 biểu diễn các đối tượng được có liên quan nhau trong biểu đồ tuần tự này.



Hình 9.43: Các đối tượng trong use case Vehicle passes beacon

Bước thứ hai sắp xếp các thông điệp. Truyền thông ở đây là không đồng bộ, và không cần phải vẽ các vùng *focus of control*. Hình 9.44 trình bày biểu đồ tuần tự kết quả.



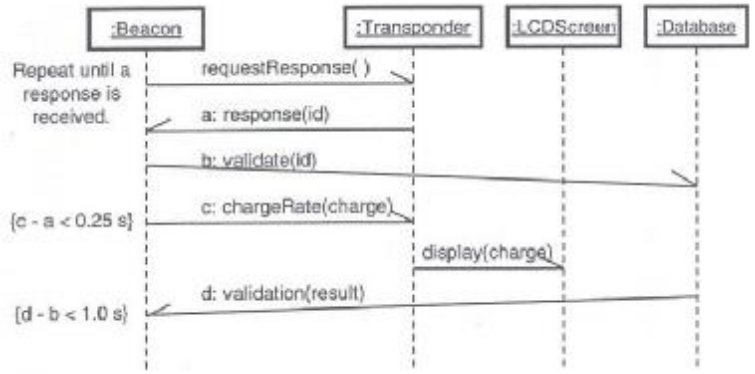
Hình 9.44: Biểu đồ tuần tự của Vehicle passes beacon, *scenario thứ nhất*

Chúng ta giả sử rằng việc gởi các tín hiệu cục bộ xảy ra ngay lập tức, trong khi đó có một khoảng thời gian chờ khi truyền thông với cơ sở dữ liệu từ xa.

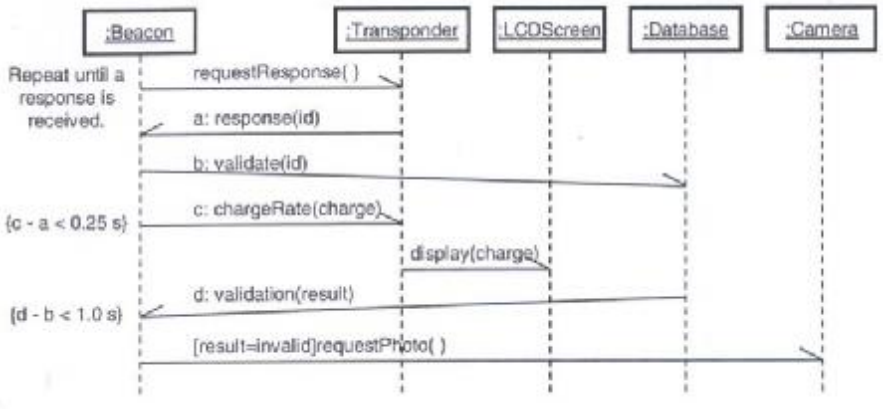
Có các ràng buộc thời gian trong tương tác này, chúng ta cũng nên thêm vào một chú thích, như hình 9.45.

Thủ tục này được lặp lại cho scenario thứ hai tạo ra biểu đồ như trong hình 9.46.

Biểu đồ này cũng là một biểu đồ tổng quát vì nó bao hàm trường hợp [result = valid] và không có thông điệp nào được gởi đến camera.



Hình 9.45: Scenario thứ nhất của Vehicle passes beacon *với các ràng buộc*

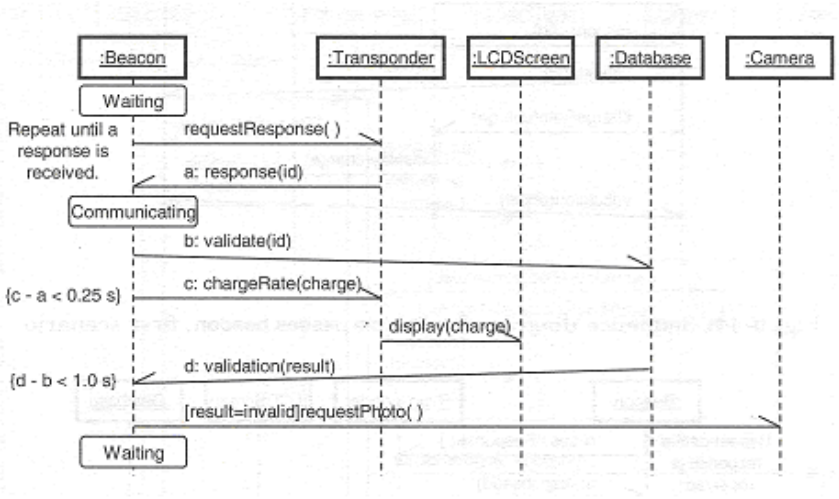


Hình 9.46: Biểu đồ tuần tự của Vehicle passes beacon, *scenario thứ hai*

9.10 Biểu đồ hình 9.46 có thể được mở rộng để mô tả các trạng thái của *Beacon*. Ở mức đơn giản nhất, có thể có hai trạng thái: *Waiting* và *Communicating* (đợi và truyền). Hãy thêm hai trạng thái này vào.

Lời giải:

Beacon ở trạng thái *Waiting* cho đến khi nó nhận phản hồi từ *Transponder* (hệ thống tiếp sóng). Khi ấy nó chuyển trạng thái sang *Communicating*. Sau khi hoàn tất tương tác, *Beacon* sẽ chuyển lại trạng thái *Waiting*. Điều này được mô tả trong hình 9.47.



Hình 9.47: Biểu đồ tuần tự với các trạng thái được thêm vào

Bài tập bổ sung

9.11 Bố trí các đối tượng trên biểu đồ tuần tự *Process payment* có các đối tượng thuộc các lớp *CarSharer*, *Account* và *Transaction* tham gia vào. Thêm vào biểu đồ các lớp *control* và *interface* thích hợp.

9.12 Bố trí các đối tượng của các lớp *PPUserInterface*, *PPControl*, *CarSharer*, *Account*, *Transaction*. Thêm các thông điệp sau đây vào biểu đồ.

1. *transactionReady*()
2. *itemList* := *getTransactionDetails*()
3. *account* := *getCarSharerAccount*()
4. *transaction* := *Transaction*(*amount*, *type*, *date*, *account*)
5. *balance* := *postTransaction*(*transaction*)
6. *displayBalance* (*balance*)

Biến *ItemList* chứa các mục dữ liệu *carsharerID*, *amount*, *type* và *date*.

9.13 Một trong các use case bạn tìm thấy trong chương 3 nên có tên chẳng hạn là *Record an individual's for help*. Trong trường hợp bạn không có một use case như thế, hãy xem đoạn mô tả dưới đây.

Các cá nhân có thể yêu cầu trợ giúp từ *VolBank*, kể cả những tình nguyện viên. Trước tiên, người quản trị *VolBank* nhập vào chi tiết của cá nhân. Nếu đây là một tình nguyện viên đang tồn

tại, thì nhập tên của người này và chi tiết sẽ được hiển thị. Nếu có nhiều người cùng tên thì một danh sách sẽ được hiển thị, danh sách này có phần đầu địa chỉ của từng người, dựa vào đây người quản trị có thể chọn được người thích hợp. Nếu tình nguyện viên này chưa có trong danh sách, thì cần nhập vào tên, địa chỉ, số điện thoại. Thông tin mã vùng và mã bưu điện rất quan trọng. Chi tiết yêu cầu trợ giúp cần được nhập vào: văn bản tóm tắt và mã mô tả, ví dụ như *DEC* cho *decorating*, *GAR* cho *garden work* hoặc *PET* cho *pet care*. Ngày bắt đầu và ngày kết thúc cũng cần được nhập vào.

Nếu bạn tạo ra một giải pháp cho các vấn đề này trong chương 8, bạn sẽ có một cộng tác là ngữ cảnh của tương tác ở đây. Nếu không bạn hãy trả lời các câu hỏi sau:

1. Những lớp bao hàm trong use case này là gì?
2. Các hệ thống con nào (nếu có) có trong use case này?
3. Những lớp bổ sung nào sẽ được yêu cầu thêm vào cộng tác này?
4. Hãy vẽ một biểu đồ lớp mô tả sự kết hợp giữa các lớp, kể cả các kết hợp stereotype nếu cần.
5. Những vai trò mà các lớp này đảm nhiệm trong cộng tác này là gì?

9.14 Những scenario nào có thể xảy ra?

9.15 Hãy vẽ một biểu đồ tuần tự thể hiện cho mỗi scenario trên. Trong mỗi biểu đồ, hãy thực hiện các bước sau:

1. Bố trí các đối tượng.
2. Bố trí các thông điệp.
3. Thêm các ràng buộc và chú thích.

9.16 Hãy vẽ một biểu đồ tuần tự tổng quát phối hợp các scenario trên.

9.17 Khi một tình nguyện viên đăng ký với *VolBank* trên *web server*, người ấy sẽ thực hiện tương tác sau:

Vào trang chủ *VolBank*. Tìm trang đăng ký *VolBank*. Điền thông tin đầy đủ vào *form* và *submit* nó. *Server* sẽ kiểm tra tính hợp lệ của dữ liệu. Nếu có lỗi, *form* sẽ được hiển thị lại, trường có dữ liệu nhập không hợp lệ sẽ được làm nổi bật lên. Nếu không có lỗi, dữ liệu được *submit* vào trong cơ sở dữ liệu.

Hãy vẽ các biểu đồ tuần tự thể hiện và tổng quát cho tương tác này.

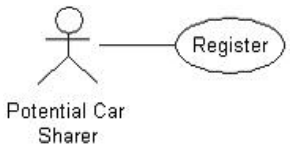
Chương 10
BIỂU ĐỒ HOẠT ĐỘNG

10.1 Giới thiệu

Biểu đồ hoạt động (activity diagram) là một phương tiện mô tả các *dòng công việc* (workflow) và được dùng theo nhiều cách khác nhau. Như một công cụ phân tích, nó mô tả các *dòng nghiệp vụ* (business flow) với nhiều mức độ chi tiết, mô tả các *dòng* phức tạp bên trong use case hoặc giữa các use case. Ở mức thiết kế, biểu đồ hoạt động được dùng để mô tả chi tiết bên trong một thao tác; trong trường hợp này, chúng rất linh động. Ngoài ra trước khi xác định use case, nó còn được dùng để xác định các yêu cầu nghiệp vụ ở mức cao, một phương tiện mô tả use case và các hành vi phức tạp bên trong đối tượng. Các biểu đồ hoạt động bổ sung cho các biểu đồ tương tác, và có quan hệ mật thiết với biểu đồ trạng thái.

10.2 Biểu đồ hoạt động là gì?

Biểu đồ hoạt động gồm *hoạt động* (activity), *trạng thái* (state) và *chuyển tiếp* (transition) giữa các hoạt động, các trạng thái. Sau đây, chúng ta sẽ khảo sát một ví dụ đơn giản. Chúng ta có một use case nghiệp vụ được định nghĩa cho phép thành viên mới đăng ký với *CarMatch*, hình 10.1.

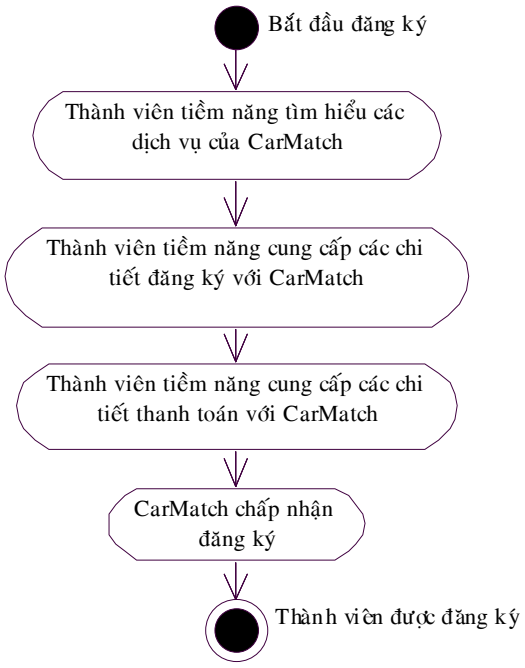


Hình 10.1 Use case nghiệp vụ cho một car sharer *tiềm năng đăng ký với* CarMatch

Trước khi định nghĩa các use case hệ thống, chúng ta phải hiểu rõ qui trình nghiệp vụ của hệ thống. Một cách làm là phát triển một biểu đồ hoạt động (hình 10.2).

Biểu đồ này mô tả qui trình chính đăng ký thành viên mới với *CarMatch*. Theo đó, trước tiên thành viên mới tìm hiểu các dịch vụ, sau đó đồng ý đăng ký, thanh toán lệ phí và cuối cùng được *CarMatch* chấp nhận. Bạn có thể thấy đây là một lưu đồ đơn giản mô tả các hoạt động nghiệp vụ theo thứ tự mà chúng cần được tuân theo. Trong chương này

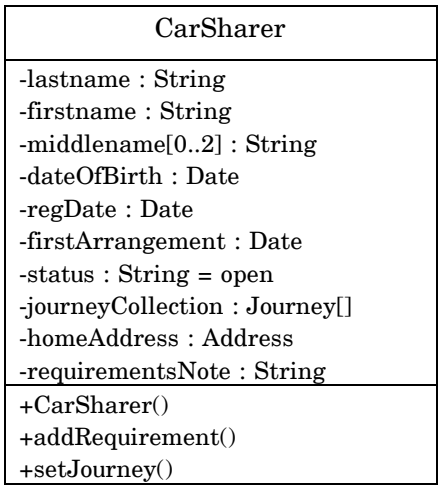
chúng ta sẽ khảo sát cách dùng các biểu đồ trạng thái để mô tả chi tiết các dòng nghiệp vụ. Chúng ta sẽ giải thích cách thêm các điều kiện, cách mô tả các dòng nghiệp vụ song song, cách áp dụng các điều kiện, và cách mô tả đầy đủ các hoạt động và các trạng thái.



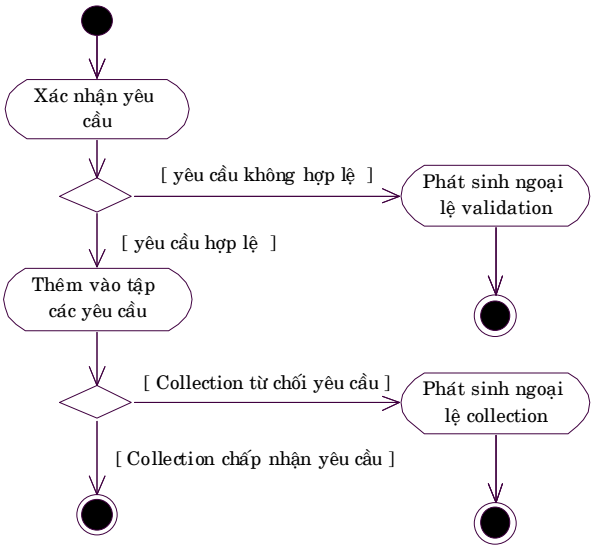
Hình 10.2: Biểu đồ hoạt động mô tả các dòng nghiệp vụ của việc đăng ký

Ở khía cạnh đặc biệt khác, chúng ta muốn mô tả các *dòng* phức tạp bên trong thao tác. Trong hình 10.3, đối tượng *CarSharer* có một thao tác *addRequirement*(). Khi thiết kế, ta có thể nhận ra đây là một thao tác phức tạp, nên muốn dùng biểu đồ hoạt động để mô tả các *dòng* trong thao tác này. *Đường dẫn chính* được xem xét trước tiên, như trong hình 10.4, cho thấy một yêu cầu mới được thêm vào như thế nào.

Trong thực hành, không phải tất cả các thao tác đều được mô tả theo cách này, chỉ hầu hết các thao tác phức tạp mà thôi. Mỗi hoạt động sẽ ánh xạ tới một hoặc một vài chỉ thị đơn giản trong một ngôn ngữ lập trình, và bạn được phép đặt các chỉ thị chương trình bên trong biểu đồ hoạt động.



Hình 10.3: Định nghĩa lớp cho đối tượng CarSharer



Hình 10.4: Biểu đồ hoạt động mô tả đường dẫn chính của thao tác addRequirement

10.3 Mục đích của kỹ thuật

Các biểu đồ hoạt động có thể được dùng trong suốt một dự án, từ khâu phân tích nghiệp vụ đến thiết kế chương trình. Chúng có thể được đính kèm với nhiều loại đối tượng, chẳng hạn như các use case nghiệp vụ, use

case hệ thống, biểu đồ use case, các hoạt động và các thao tác. Chúng là các biểu đồ *dòng* tổng quát được dùng để:

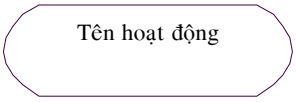
- Mô hình các dòng nghiệp vụ.
- Xác định các use case, qua việc khảo sát các dòng nghiệp vụ.
- Xác định các điều kiện đầu và cuối cho các use case.
- Mô hình *dòng công việc* giữa các use case.
- Mô hình *dòng công việc* bên trong các use case.
- Mô hình các *dòng* phức tạp bên trong thao tác trên đối tượng.
- Mô hình chi tiết các hoạt động phức tạp trong một mô hình hoạt động ở mức cao.

10.4 Ký hiệu

Các ký hiệu ở đây rất phong phú. Không phải tất cả chúng được sử dụng ở bất kỳ thời điểm nào. Một số ký hiệu, như biểu tượng điều khiển chẳng hạn, có tác động trực quan hơn là cung cấp các đặc tả. Trong giai đoạn đầu của qui trình phân tích, các hoạt động trong dòng nghiệp vụ không cần thiết phải mô tả bằng các ký hiệu đầy đủ mà tốt hơn nên mô tả chúng bằng văn bản trong những tài liệu tách rời.

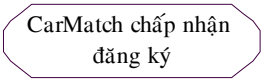
10.4.1 Hoạt động (*activity*) và hành động (*action*)

Một hoạt động là một đơn vị công việc cần được thực hiện. Trong thực tế, nó có thể lớn hay nhỏ, xảy ra trong một khoảng thời gian ngắn hay dài. Một hoạt động chẳng hạn như đòi nợ có thể mất vài tuần lễ. Một hoạt động máy tính, chẳng hạn như thay đổi một thuộc tính của khách hàng gần như được thi hành ngay lập tức. Ký hiệu của hoạt động là một hình chữ nhật bo tròn ở hai đầu, bên trong là tên mô tả hoạt động (hình 10.5).



Hình 10.5: Ký hiệu hoạt động

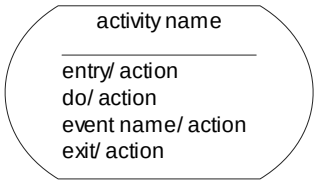
Ở mức thiết kế, tên mô tả có thể là các chỉ thị chương trình (hình 10.6).



Hình 10.6: Hoạt động với tên mô tả

Các hoạt động thực hiện công việc. Công việc được sưu liệu như là các *hành động* trong *hoạt động* này, hoặc được sưu liệu trong một tài liệu văn bản riêng. Có bốn cách để gây ra một *hành động*:

- *On Entry*: hành động được thực hiện ngay khi hoạt động bắt đầu.
- *Do*: hành động được thực hiện trong suốt thời gian sống của hoạt động.
- *On Event*: hành động được thực hiện để đáp ứng một biến cố.
- *On Exit*: hành động được thực hiện trước khi hoạt động hoàn tất.



Hình 10.7: Hoạt động với các hành động

Hành động được mô tả bằng một câu tiếng Anh đơn giản hoặc bằng một chỉ thị của ngôn ngữ lập trình. Với hành động do biến cố gây ra có thể được viết như sau:

^*target.event*(*arguments*)

trong đó *target* là đối tượng đáp ứng biến cố, *event* là tên biến cố và *argument* chứa các thông tin được chuyển tới cùng với biến cố.

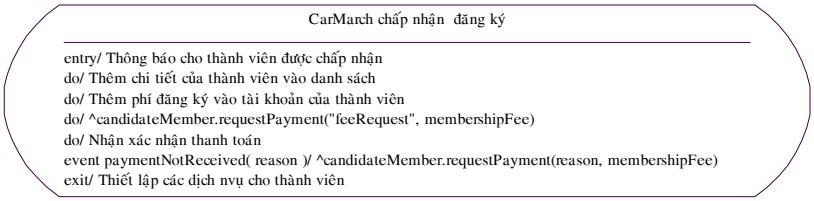
Một biểu đồ hoạt động có thể được dùng để sưu liệu cho các quyết định nghiệp vụ.

Ví dụ 10.1

Một khi *CarMatch* chấp nhận việc đăng ký, thành viên đó được thông báo (*notify*) ngay lập tức. Các chi tiết về người đăng ký được thêm (*add member*) vào danh sách thành viên và một phí đăng ký được tính (*add registration fee*) vào tài khoản của người đó. Yêu cầu thanh toán (*requestPayment*) được phát sinh cho thành viên này (^*candidateMember*), với phí thành viên (*membershipFee*) là một trong các đối số, và yêu cầu thanh toán phí (‘*feeRequest*’) là lý do. Xác nhận thanh toán (*receive confirmation of payment*) được yêu cầu trước khi dịch vụ hiệu lực. Có thể có vấn đề xảy ra với thanh toán, chẳng hạn như các chi tiết về thẻ tín dụng sai, điều này sẽ được thông báo cho hoạt động bằng biến cố không nhận được thanh toán (*paymentNotReceived*) với lý do (*reason*) và yêu cầu thanh toán khác được phát sinh với lý do vừa

nhận được. Khi toàn bộ công việc này được thực hiện xong, thành viên này mới có thể dùng được dịch vụ (*enable services*). (xem toàn bộ các hành động được minh họa trong hình 10.8)

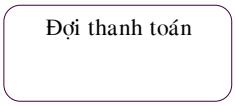
Tóm lại, chúng ta có thể mô tả chi tiết hoạt động trên như trong hình 10.8. Lưu ý rằng chúng ta không xác định đầy đủ thứ tự của các hành động. Một hoạt động phức tạp tốt nhất nên được mô tả bằng một biểu đồ hoạt động, trong đó các hành động của hoạt động này xuất hiện như các hoạt động mới theo thứ tự được chỉ định. Điều này có thể được thực hiện bằng cách dùng các biểu đồ hoạt động con trên cùng một biểu đồ (xem hình 10.25) hoặc bằng cách tạo ra một biểu đồ hoạt động riêng. Một hoạt động cũng có thể được sưu liệu bởi một mô tả dạng văn bản và được lưu trữ trong một tập tin.



Hình 10.8: Một hoạt động với nhiều hành động

10.4.2 Trạng thái (*state*)

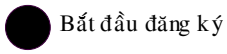
Một trạng thái trong biểu đồ hoạt động là một điểm chờ một biến cố xảy ra trước khi hoạt động được tiếp tục. Nói chung hoạt động và trạng thái gần như tương đương nhau, trạng thái cũng có thể thực hiện các hành động. Tuy nhiên, hoạt động phải thực hiện các hành động, còn trạng thái ngầm chỉ việc chờ đợi. Ký hiệu của trạng thái là một hình chữ nhật có các góc được bo tròn như sau:



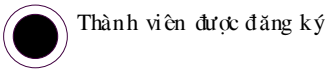
Hình 10.9: Ký hiệu trạng thái

Có hai trạng thái đặc biệt. Trạng thái bắt đầu là điểm vào của dòng. Trong một biểu đồ, chỉ có một trạng thái bắt đầu. Nó được biểu diễn bằng một chấm đen có thể đính kèm nhãn như trong hình 10.10. Trạng thái bắt đầu cũng có thể có các hành động mặc dù hầu hết các trạng thái bắt đầu được dùng như một dấu hiệu cho biết rằng một dòng được bắt

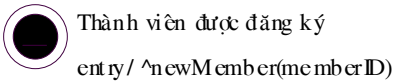
đầu. Trạng thái kết thúc được biểu diễn bằng một chấm đen với một vòng tròn bao bên ngoài như trong hình 10.11. Các trạng thái kết thúc cũng có thể có các hành động như trong hình 10.12. Bạn có thể dùng chúng để phát sinh các biến cố bắt đầu các tiến trình khác, ví dụ trong hình 10.12 trạng thái kết thúc phát tín hiệu cho biết một thành viên mới đã được chấp nhận để qui trình khác có thể thực hiện việc kết hợp.



Hình 10.10: Trạng thái bắt đầu



Hình 10.11: Trạng thái kết thúc.

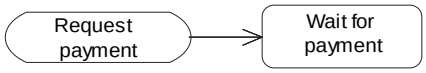


Hình 10.12: Trạng thái kết thúc với một hành động.

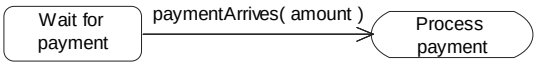
Trong một dòng có thể có nhiều trạng thái kết thúc, chúng cho biết có các tiến trình khác theo sau. Ví dụ, qui trình xử lý hóa đơn có thể kết thúc với việc thanh toán thành công, dẫn đến việc khách hàng được phép thực hiện các việc mua sắm sau này, hoặc kết thúc với việc thanh toán thất bại, dẫn đến việc khách hàng được ghi nhận như là một con nợ xấu.

10.4.3 Sự chuyển tiếp (*Transition*)

Một *transition* là sự di chuyển từ một hoạt động này sang một hoạt động khác, thay đổi từ trạng thái này sang trạng thái khác hoặc chuyển tiếp từ một trạng thái sang một hoạt động hoặc ngược lại. *Transition* thường xảy ra khi tất cả các hành động của một hoạt động đã hoàn tất hoặc khi một biến cố gây ra sự kết thúc trạng thái hoặc hoạt động. *Transition* được biểu diễn bằng một mũi tên như trong hình 10.13.



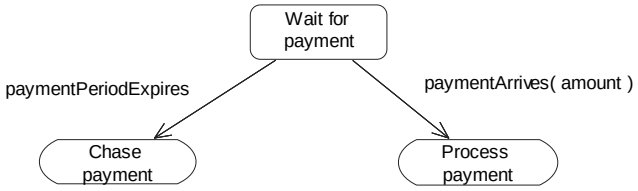
Hình 10.13: Một chuyển tiếp không cần kích hoạt giữa hoạt động và trạng thái



Hình 10.14: Một chuyển tiếp có kích hoạt từ một trạng thái

Trong ví dụ này, khi hoạt động *yêu cầu thanh toán* (request payment) được thực hiện xong là bắt đầu trạng thái *chờ thanh toán* (wait for payment). Đây là một ví dụ về một *chuyển tiếp không có biến cố* vì không có biến cố bên ngoài nào làm việc *chuyển tiếp* xảy ra ngoại trừ hoạt động *yêu cầu thanh toán* hoàn tất công việc của nó. Các *chuyển tiếp không có biến cố* dùng để kết thúc hoạt động. Còn khi thoát ra khỏi một trạng thái, cần phải có một biến cố. Ví dụ, khi đang ở trạng thái *chờ thanh toán*, biến cố *thanh toán* (paymentArrives) sẽ kết thúc trạng thái này, kèm theo lượng thanh toán được mô tả bằng đối số amount (hình 10.14).

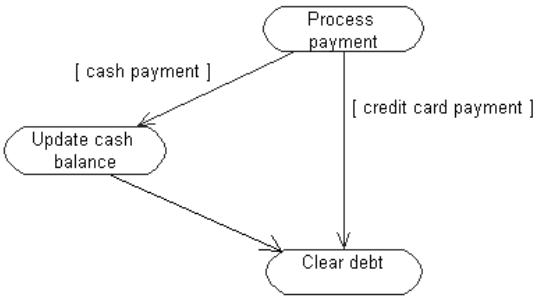
Có thể có nhiều hơn một *transition* từ một hoạt động hoặc trạng thái. Điều này xảy ra khi có nhiều biến cố khác nhau làm kết thúc hoạt động hoặc trạng thái. Ví dụ, nếu việc trả tiền không được thực hiện trong một khoảng thời gian nhất định, sẽ bắt đầu hoạt động *Chase payment* thay vì *Process payment*, xem hình 10.15



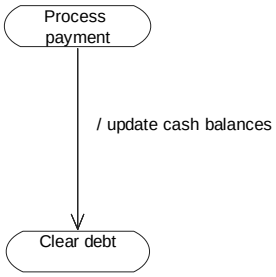
Hình 10.15: Nhiều chuyển tiếp từ một trạng thái, được gây ra bởi các biến cố khác nhau

Các điều kiện có thể được thiết lập trên *transition* được đặt trong cặp dấu ngoặc vuông và được gọi là các *điều kiện bảo vệ* (guard-condition). Việc chuyển tiếp chỉ xảy ra nếu điều kiện được thoả. Ví dụ, giả sử trong quá trình thanh toán, có một tuỳ chọn là *thanh toán bằng tiền mặt* và điều này phát sinh thêm yêu cầu *cân đối tiền mặt* trước khi *xóa nợ*. Dòng này được mô tả trong hình 10.16.

Một *transition* có thể có một hoặc nhiều hành động kết hợp với nó. Một hành động được bắt đầu bằng một dấu gạch chéo ('/'). Trong môi trường mà tiền mặt là phương tiện thanh toán duy nhất, hình 10.6 có thể được vẽ lại đơn giản hơn như trong hình 10.17.



Hình 10.16: Nhiều chuyển tiếp không cần kích hoạt dùng điều kiện bảo vệ



Hình 10.17: Dùng một hành động trên một chuyển tiếp

Một hành động trên *transition* có thể gây ra một biến cố. Cú pháp đầy đủ của nhãn của *transition* (transition label) như sau:

event (arguments) [condition] / action ^target.sendEvent (arguments)

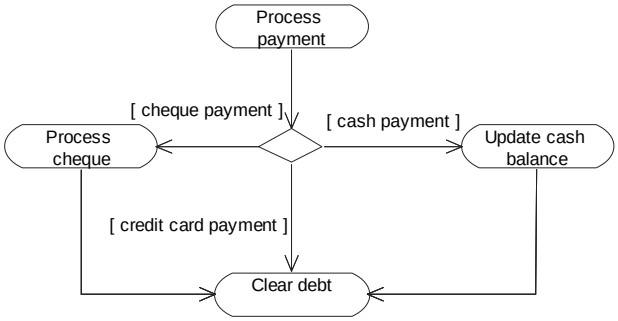
Thứ tự thực hiện của các hành động như sau:

1. Khi hoạt động hoặc trạng thái kết thúc (do biến cố hoặc do hoàn tất tất cả các hành động), thì hành động *on exit* được thực hiện.
2. Sau đó, sự chuyển tiếp xảy ra và các hành động kết hợp được thực hiện.
3. Cuối cùng là các hành động *on entry* của hoạt động hay trạng thái mới.

10.4.4 Điểm quyết định (decision point)

Điểm quyết định là một điểm trên *dòng công việc* mà tại đó việc kết thúc có thể rẽ nhánh theo các hướng khác nhau tuỳ thuộc vào điều kiện. Các quyết định được ký hiệu bằng một hình thoi, một quyết định có thể có một hoặc nhiều đầu vào và hai hoặc nhiều đầu ra. Trong hình 10.18, có ba chọn lựa sau khi kết thúc hoạt động *Process payment*, tuỳ thuộc vào việc thanh toán được trả bằng *ngân phiếu* (cheque), *tiền mặt* (cash) hay

thẻ tín dụng (credit card). Lưu ý UML không chỉ định cú pháp của điều kiện, các điều kiện có thể có cấu trúc bất kỳ.



Hình 10.18: Rẽ nhánh được mô tả bằng điểm quyết định

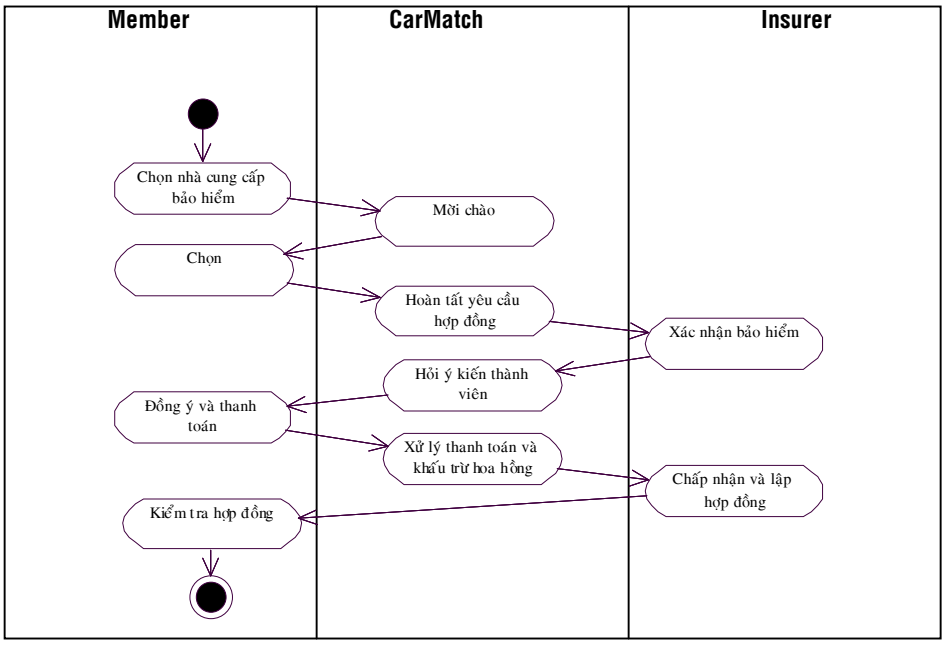
Cách này mô tả rõ ràng các quyết định và các nhánh rẽ, nhưng có thể làm biểu đồ phức tạp. Tuy nhiên, để lần theo *dòng* được dễ hơn, đôi khi ta nên kết thúc hoạt động hoặc trạng thái bằng nhiều cách, thông qua nhiều điều kiện bảo vệ hoặc nhiều biến cố, xem mô tả trong mục 10.4.3.

10.4.5 Swimlane (đường phân dòng nghiệp vụ)

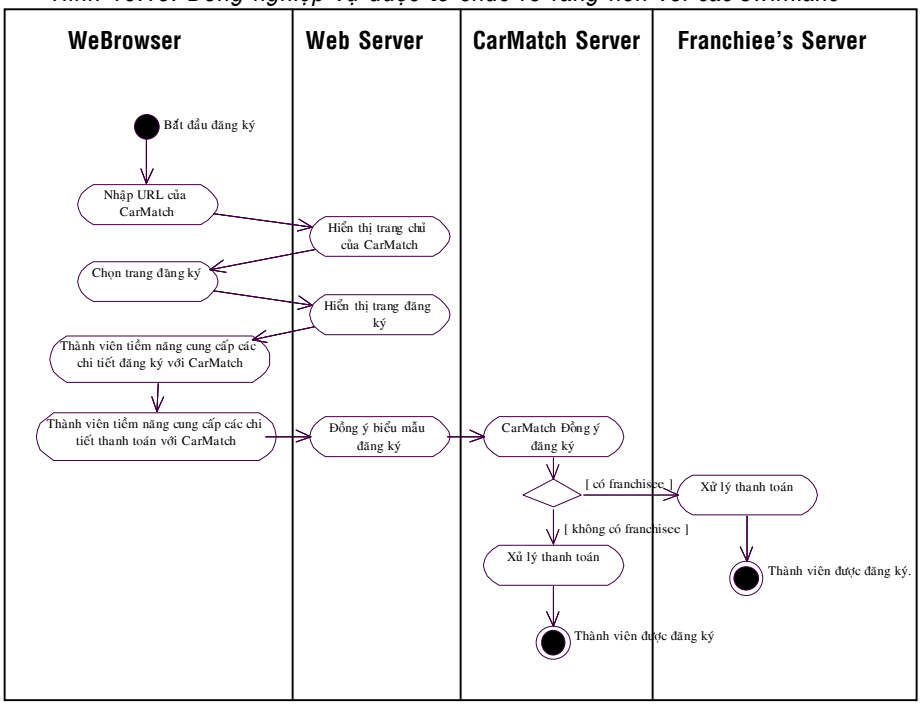
Swimlane là một ký hiệu rất hữu ích cho biết một hoạt động xảy ra ở đâu trong một hệ thống phức tạp. Các *swimlane* là các cột trên một biểu đồ hoạt động và các hoạt động trong biểu đồ được nhóm lại trong các *swimlane*. Ví dụ, *CarMatch* thay mặt cho công ty bảo hiểm đưa ra các hợp đồng bảo hiểm cho các thành viên. Để mua hợp đồng, các hoạt động phải được thực hiện bởi khách hàng, bởi *CarMatch* và bởi công ty bảo hiểm. Qui trình này được sắp xếp theo hai chiều như trong hình 10.19.

Lưu ý rằng một số hoạt động được thực hiện bởi khách hàng. Để xây dựng một hệ thống thông tin thoả mãn các yêu cầu hỗ trợ các hoạt động này, cần đến các use case với tác nhân bên trong để xử lý các kết quả từ hoạt động của khách hàng.

Swimlane cũng được dùng để xác định ở mức kỹ thuật các vùng có các hoạt động được thực hiện. Một cách dùng phổ biến là xác định các hoạt động sẽ được thực hiện ở đâu trong môi trường Internet. *CarMatch* có một web-server chính ở mỗi quốc gia. Các thành viên và các tổ chức quan tâm có thể truy cập đến các web-server này. Các server sẽ liên kết đến server *CarMatch* trung tâm, server này sẽ lấy đăng ký ban đầu sau đó ủy quyền xử lý cuối cùng của việc đăng ký cho một chi nhánh địa phương, nếu có. Biểu đồ này được vẽ như trong hình 10.20.



Hình 10.19: Dòng nghiệp vụ được tổ chức rõ ràng hơn với các swimlane

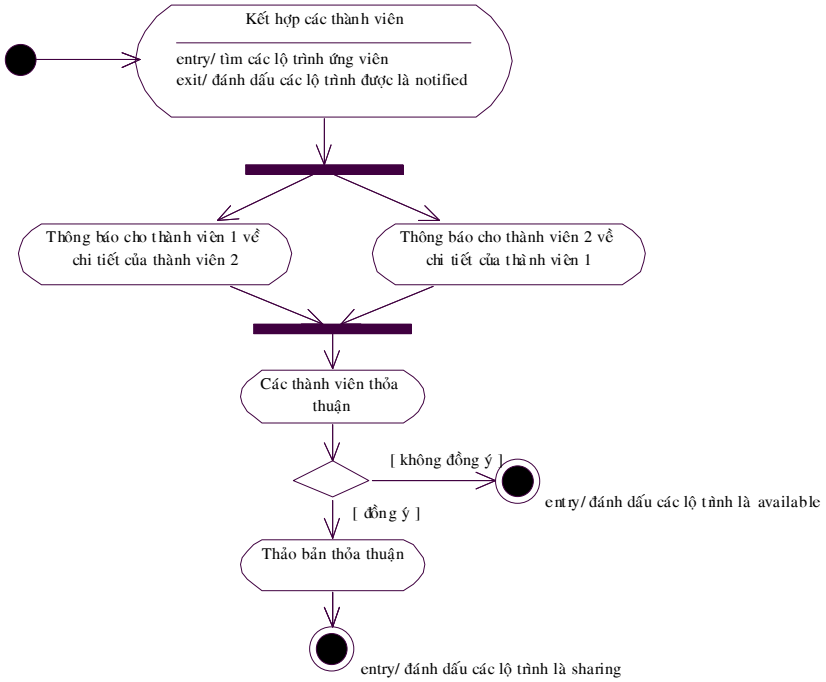


Hình 10.20 Đăng ký thành viên CarMatch qua mạng

10.4.6 Fork và Join (phân nhánh và kết hợp)

Đôi khi có một số hoạt động cần được thi hành song song. Một *transition* có thể được chia thành nhiều nhánh và ngược lại các nhánh có thể được kết hợp lại thành một *transition* bằng cách dùng một *thanh đồng bộ hóa* (synchronization bar). Chỗ tách ra gọi là *fork* và chỗ kết hợp gọi là *join*.

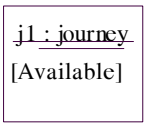
Trong *CarMatch*, có một use case là *Match car sharers*. Use case này được mô hình thành một qui trình nghiệp vụ. Một khi các lộ trình được kết hợp, ta phải thông báo đến các thành viên cùng lúc. Sau đó các thành viên sẽ phải thu xếp để cùng nhau thảo một bản thoả thuận với sự giúp đỡ của *CarMatch*. Biểu đồ này sẽ có dạng như trong hình 10.21.



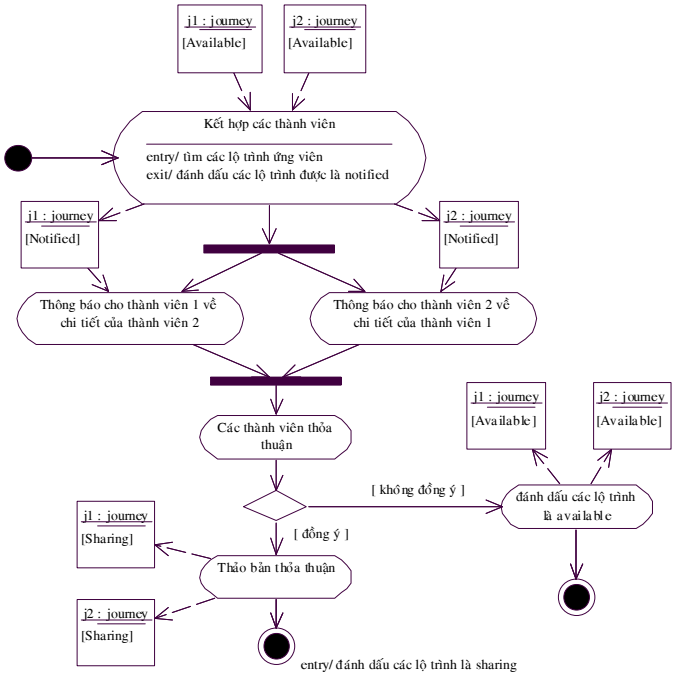
Hình 10.21: Các hoạt động song song được biểu diễn các thanh đồng bộ hóa

Một thanh đồng bộ hóa có thể có một *transition* vào và hai hay nhiều *transition* ra hoặc ngược lại nhiều *transition* vào và một *transition* ra. Điều này là quan trọng khi *dòng* bị chia thành các *dòng* song song rồi được kết hợp lại trên cùng biểu đồ.

10.4.7 Các đối tượng trên biểu đồ hoạt động



Hình 10.22: Một đối tượng trên biểu đồ hoạt động được kết hợp với một trạng thái



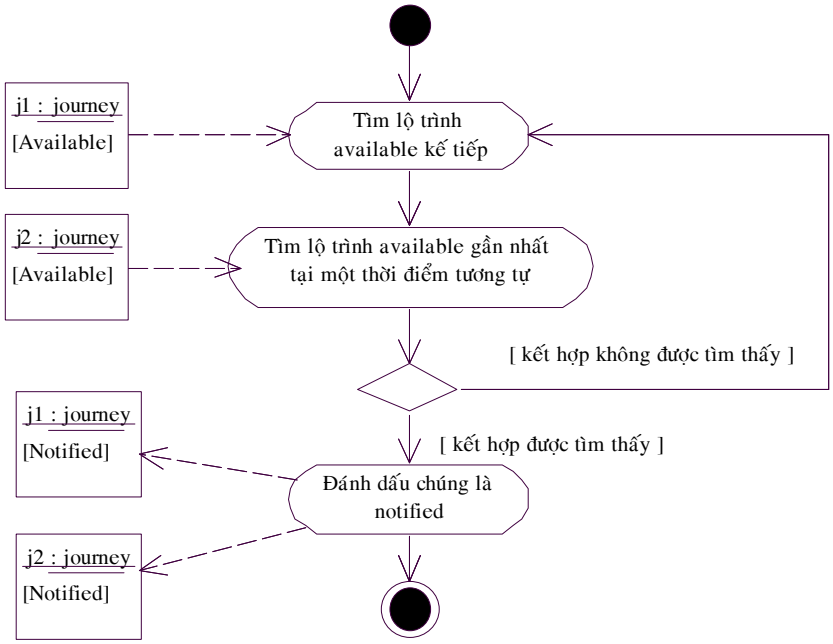
Hình 10.23: Các dòng đối tượng trên use case journey matching

Cho đến bây giờ, các biểu đồ hoạt động có vẻ hơi tách khỏi các đối tượng. Tuy nhiên, trong một cài đặt hướng đối tượng, tất cả các hoạt động đều cần được thực hiện bởi các đối tượng. Đôi khi việc chỉ ra vị trí trên biểu đồ ảnh hưởng đến một đối tượng là rất hữu ích. Bạn có thể thực hiện việc này bằng cách đặt một đối tượng vào biểu đồ và liên kết nó đến một hoạt động bằng mối quan hệ phụ thuộc. Các phụ thuộc như thế được gọi là các *dòng đối tượng* bởi chúng chỉ ra cách đối tượng được dùng. Dòng nghiệp vụ sẽ làm thay đổi trạng thái của đối tượng. Vì thế, mỗi đối tượng sẽ có một trạng thái kết hợp (cách lập mô hình trạng thái sẽ được khảo

sát trong chương 11). Với *Match car sharers* chúng ta muốn thấy tác động của qui trình lên các lộ trình. Chúng ta đặt một thể hiện *j1* của lớp *Journey* với trạng thái *Available* trên biểu đồ như trong hình 10.22.

Chúng ta có thể gắn các đối tượng vào các hoạt động làm thay đổi trạng thái của chúng. Vì thế với biểu đồ hoạt động ở trên chúng ta đưa ra hai lộ trình bắt đầu với trạng thái *Available*. Các thành viên đưa ra các lộ trình này sẽ được thông báo và các chúng sẽ có trạng thái *Notifed* để quá trình không kết hợp chúng nữa cho đến khi xử lý hoàn tất. Cuối cùng, các lộ trình hoặc kết thúc với trạng thái *Sharing* nếu các thành viên đồng ý chia sẻ, hoặc quay về trạng thái *Available* nếu thoả thuận không đạt được. Điều này được trình bày trong hình 10.23.

10.4.8 Các biểu đồ hoạt động lồng nhau

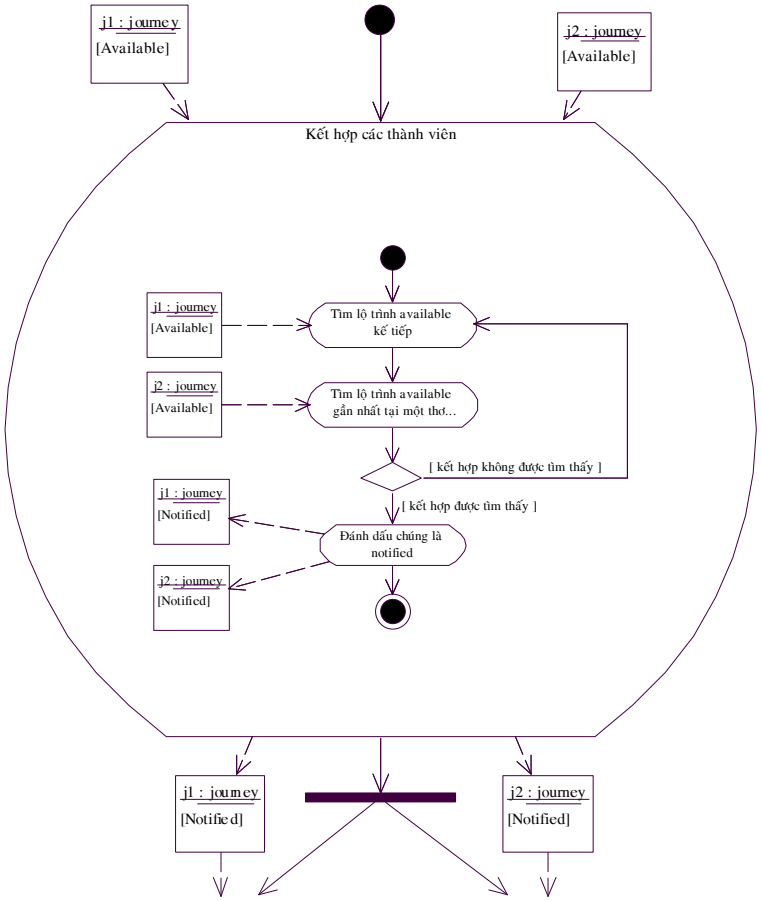


Hình 10.24: Một biểu đồ con

Biểu đồ hoạt động có thể dần phức tạp; các hoạt động trong biểu đồ cũng vậy. Khi ấy một hoạt động có thể được mô hình bởi một biểu đồ con. Hoạt động *Match Sharers* ở trên có thể có một biểu đồ hoạt động riêng. Trong hình 10.24, mô tả cách tìm một kết hợp, bắt đầu với lộ trình *available*, sau đó tìm một lộ trình khác phù hợp với lộ trình này. Nếu

không thấy lộ trình thứ hai thì lộ trình đầu sẽ bị bỏ qua và tới lượt tìm kiếm khác, cứ như thế cho đến khi tìm thấy một cặp.

Hình 10.25 minh họa các dòng lồng nhau trong cùng một biểu đồ.



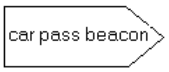
Hình 10.25: Một biểu đồ con được lồng bên trong một hoạt động

10.4.9 Các biểu tượng điều khiển (control icon)

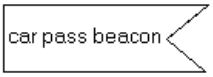
Biến cố xuất hiện hoặc hoạt động kết thúc sẽ phát sinh *transition*, ngược lại *transition* cũng có thể phát sinh biến cố. Chúng ta đã có các ký hiệu mô tả điều này một cách đầy đủ, nhưng UML cung cấp thêm hai biểu

tượng để mô tả việc xử lý các biến cố được rõ ràng hơn. Chúng là các *biểu tượng điều khiển*.

Đầu tiên là biểu tượng gởi tín hiệu (ví dụ như biến cố), hình 10.26, với mũi nhọn hướng ra biểu diễn việc phát sinh một biến cố. Trong hình này, nó cho biết một xe hơi đã qua một mốc tín hiệu trên một lược đồ tính cước phí đi đường. Biểu tượng thứ hai biểu thị việc nhận tín hiệu có dạng mũi nhọn hướng vào, hình 10.27, cho biết việc nhận một biến cố.

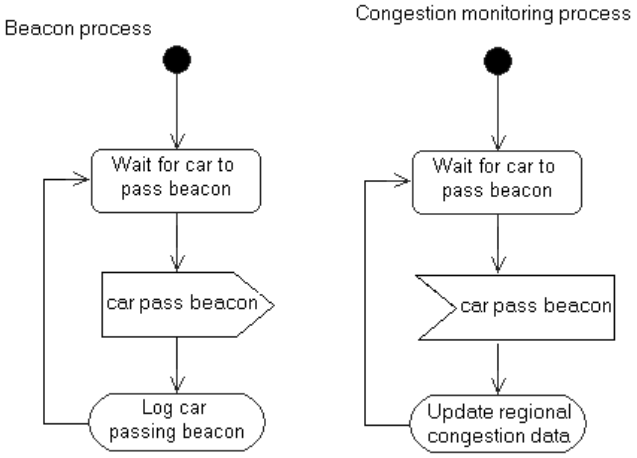


Hình 10.26: Gửi tín hiệu khi xe đi qua mốc tín hiệu trong lược đồ road-pricing



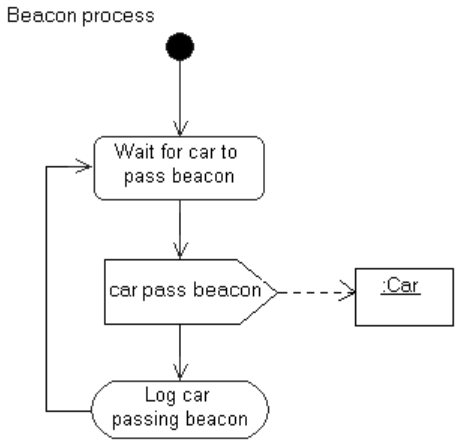
Hình 10.27: Nhận tín hiệu khi xe đi qua mốc tín hiệu trong lược đồ road-pricing

Biến cố có thể được truyền giữa các tiến trình. Chúng ta có một tiến trình trong một *mốc tín hiệu* để ghi nhận xe đi qua và một tiến trình trong hệ thống theo dõi tắc đường, nó lấy tín hiệu từ *mốc tín hiệu* và cập nhật số xe trong một vùng (hình 10.28). Các tiến trình này xảy ra trong các hệ thống máy tính khác nhau, được liên kết qua một hệ thống liên lạc. Lưu ý trong trường hợp này, nhiều *mốc tín hiệu* có thể gửi cùng một kiểu biến cố tác động đến cùng một vùng.



Hình 10.28: Các qui trình trong mốc tín hiệu *và hệ thống theo dõi sự ùn tắc giao thông để xử lý xe đi ngang qua*

Các phụ thuộc giữa hoạt động và đối tượng xử lý biến cố cũng có thể được thêm vào biểu đồ. Ví dụ, trong *mốc tín hiệu* có thể có một đối tượng có nhiệm vụ đáp ứng tín hiệu xe qua, cập nhật số tiền phải trả. Điều này được biểu thị bằng một liên kết phụ thuộc vào biểu đồ như trong hình 10.29.



Hình 10.29: Một phụ thuộc cho biết rằng một thể hiện của lớp Car *đáp ứng tín hiệu khi một xe hơi qua* mốc tín hiệu

10.5 Cách tạo biểu đồ hoạt động

10.5.1 Biểu đồ hoạt động cho mô hình nghiệp vụ

Biểu đồ hoạt động là một phương tiện mô tả các dòng nghiệp vụ khi phát triển mô hình nghiệp vụ. Các mô hình nghiệp vụ bao gồm rất nhiều qui trình. Các qui trình này có thể được mô tả đầy đủ bằng các biểu đồ hoạt động. Việc phát triển các mô hình nghiệp vụ bao gồm:

- Tìm kiếm các tác nhân nghiệp vụ và các use case nghiệp vụ.
- Xác định kịch bản chính của các use case, dùng các nhánh chính và các nhánh thay thế.
- Kết hợp các kịch bản để tạo ra các dòng nghiệp vụ có thể hiểu được, các dòng nghiệp vụ này được mô tả bằng biểu đồ hoạt động.
- Nếu một hành vi chính của đối tượng được gây ra bởi một dòng nghiệp vụ, thì chúng ta thêm dòng đối tượng vào biểu đồ.
- Nếu thích hợp, ánh xạ các hoạt động vào các vùng nghiệp vụ và ghi lại điều này bằng cách dùng *swimlane*.
- Định nghĩa lại các hoạt động phức tạp theo cách thức tương tự.

Khi mô hình nghiệp vụ đã được mô tả như trên, có thể bắt đầu việc định nghĩa các use case hệ thống.

10.5.1.1 Xác định các use case nghiệp vụ

Giai đoạn chính đầu tiên của một dự án tin học là lập mô hình nghiệp vụ. *Jacobson, Ericsson* xem một nghiệp vụ như một hệ thống và lập mô hình một nghiệp vụ dưới dạng các use case nghiệp vụ. Điều này được hợp nhất chặt chẽ trong UP. Lý do của việc tạo ra điểm bắt đầu để phân tích này là tầm quan trọng trong việc tăng cường sự hiểu biết chung giữa tất cả các thành viên trong dự án, bao gồm chủ đầu tư, khách hàng, người dùng, người quản lý và các thành viên trong nhóm phát triển. Hơn nữa, ngữ cảnh của một hệ thống sẽ quyết định cách hoạt động của nó. Bằng cách xem nghiệp vụ như một hệ thống, trong đó hệ thống máy tính là các hệ thống con được nhúng vào, có thể mô hình hóa hành vi của môi trường mà một hệ thống máy tính hoạt động trong đó. Giai đoạn đầu tiên trong qui trình này là xác định các tác nhân dùng nghiệp vụ, bao gồm cả khách hàng, các nhà đầu tư, ngân hàng, nhà cung cấp và có thể là cả các người điều chỉnh. Mỗi tác nhân sẽ sử dụng một số use case nghiệp vụ.

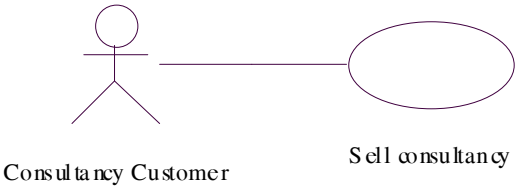
10.5.1.2 Xác định kịch bản chính trong use case nghiệp vụ

Một use case nghiệp vụ thường chứa một dòng nghiệp vụ phức tạp với rất nhiều nhánh. Khi một use case nghiệp vụ đã được xác định, giai đoạn kế tiếp là phân tích kịch bản chính sẽ được thi hành bởi use case nghiệp vụ. Một kịch bản được xem như một dãy các hoạt động nghiệp vụ. Số lượng kịch bản có thể rất lớn.

Việc phân tích nghiệp vụ cần đạt được một ví dụ tiêu biểu của các kịch bản từ đó có thể xác định một bản mô tả toàn diện các dòng trong use case nghiệp vụ. Một cách tiếp cận được khuyên dùng là bắt đầu bằng cách tìm đường dẫn chính. Đây là kịch bản được dùng phổ biến nhất. Có hai lý do để thực hiện điều này, thứ nhất là phần lớn các thể hiện của use case sẽ liên quan đến đường dẫn chính, và use case sẽ không hiệu quả nếu nó không thể xử lý công việc này. Dựa trên kinh nghiệm là 80% hoạt động dùng 20% khả năng của một hệ thống bất kỳ, việc cài đặt đường dẫn chính có thể cung cấp cách thực hiện một hệ thống khả dụng một cách nhanh chóng. Thứ hai, bằng cách xác định đường dẫn chính, một khung làm việc được thiết lập để xác định các kịch bản khác.

Ví dụ 10.2

CarMatch muốn bán các dịch vụ tư vấn khách hàng. Điều này được ghi nhận ở mức cao như một use case nghiệp vụ như trong hình 10.30.



Hình 10.30: use case nghiệp vụ Sell consultancy

Sau đây là qui trình của việc bán các đề án tư vấn cho khách hàng:

1. Liên hệ với khách hàng tương lai, từ *CarMatch* hoặc từ một tổ chức liên kết chuyển đến hoặc từ khách hàng trực tiếp.
2. Điều tra nghiên cứu bước đầu các nhu cầu của khách hàng.
3. Xây dựng một đề xuất dự án.
4. Đề xuất dự án được chấp nhận.
5. Các hợp đồng được thoả thuận.
6. Dự án được thi hành.
7. Lập hóa đơn cho khách hàng và dự án chấm dứt.

Sau khi đã có một đường dẫn chính như trên, bây giờ chúng ta có thể tìm các nhánh dẫn khác bằng cách kiểm tra mỗi bước của đường dẫn chính. Bước thứ 4 giả sử khách hàng không có phản đối gì và chấp nhận đề xuất đó, nhưng trong thực tế có thể có các thay đổi nhỏ hoặc lớn, thậm chí là huỷ bỏ qui trình. Sau đây là ba chọn lựa từ bước thứ 4:

4.1. Khách hàng muốn thay đổi đề xuất nhưng vẫn sẵn sàng đồng ý đề xuất, vì thế yêu cầu xem lại đề xuất.

4.2. Khách hàng cảm thấy đề xuất này không thoả mãn những gì họ cần nhưng họ sẽ xem xét các đề xuất khác.

4.3. Khách hàng không muốn thảo luận về một dự án nào với *CarMatch*, ít ra là tại thời điểm này.

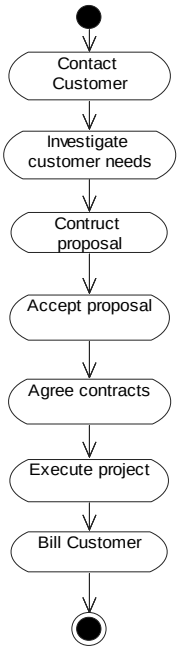
10.5.1.3 Kết hợp các kịch bản để tạo ra một dòng hoàn chỉnh

Bây giờ bạn có thể vẽ các dòng nghiệp vụ cho mỗi kịch bản, bắt đầu với đường dẫn chính và thêm các nhánh khác từng cái một.

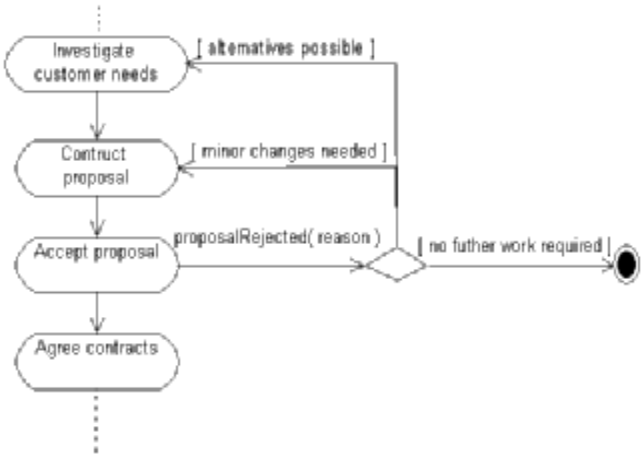
Ví dụ 10.3

Từ đường dẫn chính của use case *Sell consultancy*, chúng ta vẽ dòng nghiệp vụ như hình 10.31, sau đó thêm vào các nhánh khác. Hình 10.32 cho thấy cách dùng một biến cố thoát ra khỏi hoạt động *Accept proposal*

(đồng ý đề xuất) và quyết định có hay không việc thoát khỏi qui trình hoặc thay đổi đề xuất hiện hành hoặc tìm kiếm ý tưởng đề xuất khác; thông qua các điều kiện bảo vệ. Chúng ta tiến hành phân tích theo cách này: đặt câu hỏi cho mỗi bước trong qui trình để tìm các nhánh khác.



Hình 10.31: Dòng nghiệp vụ chính trong use case Sell consultancy



Hình 10.32: Một nhánh khác của use case Sell consultancy

Kết quả của quá trình này là một tập các use case nghiệp vụ được định nghĩa hoàn chỉnh với các dòng nghiệp vụ. Rõ ràng, việc mô tả chi tiết các use case nghiệp vụ là trọng điểm của dự án.

10.5.1.4 Gán các dòng đối tượng

Thêm các đối tượng vào ở những nơi hoạt động làm thay đổi trạng thái của chúng.

Ví dụ 10.4

Trong ví dụ *CarMatch* về việc kết hợp các thành viên, chúng ta thấy các lộ trình được kết hợp đi qua các thay đổi trạng thái có ý nghĩa (hình 10.23). Ví dụ, không thể đưa một hành trình cho thành viên khác trong khi nó đang được chờ thoả thuận. Chỉ những nơi chủ yếu như thế này là thích hợp để thêm các đối tượng vào nhằm cho biết các thay đổi trạng thái chủ yếu. Trong ví dụ *Sell consultancy* ở đây, không thích hợp xem xét các dòng đối tượng .

10.5.1.5 Ấn định hoạt động vào vùng nghiệp vụ dùng swimlane

Khi tổ chức nghiệp vụ được mô tả trước và dòng nghiệp vụ đi qua một số vùng nghiệp vụ, sẽ rất hữu ích khi vẽ các dòng nghiệp vụ bằng cách dùng các *swimlane* để cho biết vùng nghiệp vụ nào thực hiện hoạt động nào (xem hình 10.19).

10.5.1.6 Tinh chế các hoạt động

Các hoạt động ở mức trên cùng được xác định theo cách này thường đòi hỏi chi tiết hơn. Qui trình tinh chế có thể được áp dụng, tìm các đường dẫn chính và các nhánh phụ, sau đó kết hợp chúng lại để tạo ra một dòng nghiệp vụ hoàn chỉnh. Quá trình này có thể tiếp tục cho đến khi đủ chi tiết để mô tả một mô hình nghiệp vụ hoàn chỉnh. Trong thực tế, chỉ cần không quá ba mức phân rã hoạt động và đôi khi chỉ một là đủ, tuỳ vào mức độ phức tạp của qui trình mà con số này có thể lớn hơn.

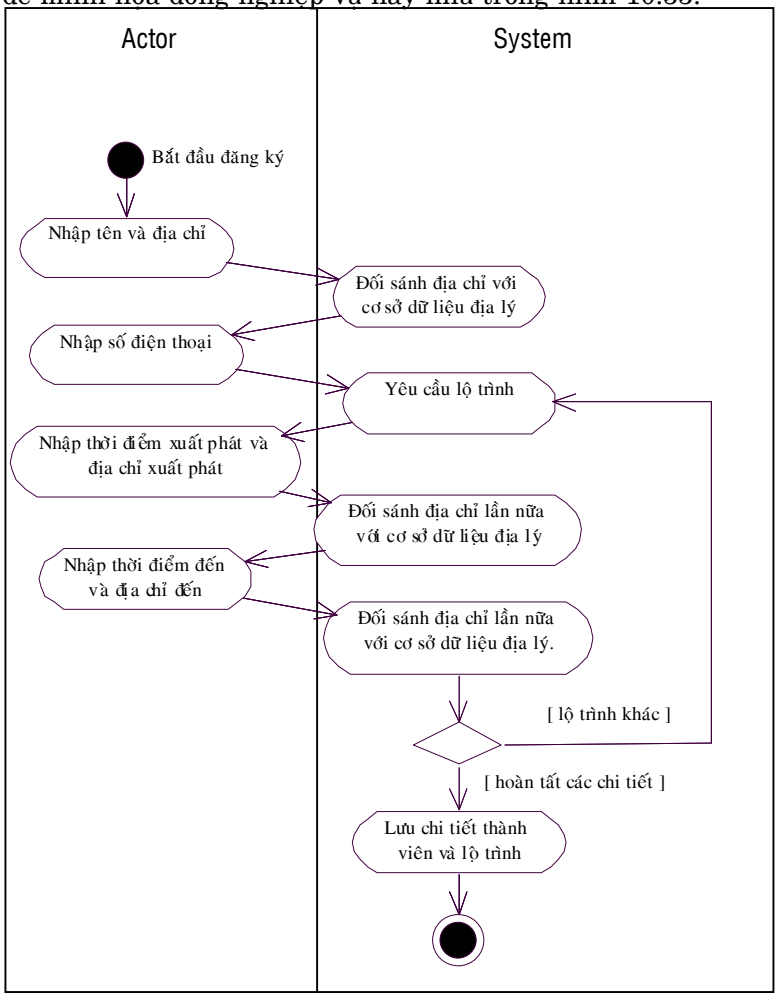
10.5.2 Cách tạo các biểu đồ hoạt động để lập mô hình use case.

Việc tạo biểu đồ hoạt động cho các mô hình use case hệ thống tương tự như đối với việc tạo các dòng nghiệp vụ.

- Xác định kịch bản chính của các use case hệ thống, dùng các đường dẫn chính và phụ.
- Kết hợp các kịch bản để tạo ra các dòng nghiệp vụ hoàn chỉnh, mô tả các dòng nghiệp vụ này bằng biểu đồ hoạt động.

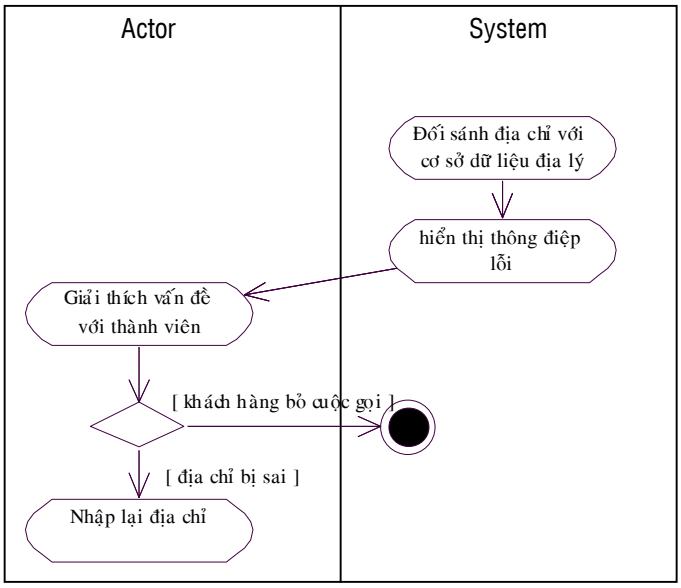
- Khi hoạt động của đối tượng bị gây ra bởi một dòng nghiệp vụ, thêm dòng đối tượng vào biểu đồ.
- Tinh chế các hoạt động phức tạp theo cách tương tự.

Nhớ rằng biểu đồ hoạt động chỉ là một cách ghi lại dòng nghiệp vụ trong một hệ thống. Biểu đồ tuần tự và biểu đồ cộng tác sẽ liên kết các dòng nghiệp vụ này thành mô hình đối tượng cơ bản. Điển hình, bạn dùng các biểu đồ hoạt động được xây dựng sớm để mô tả các use case ở những nơi mà các dòng phức tạp cần được mô tả đầy đủ, trước khi mô tả chi tiết mô hình đối tượng. Mô tả chi tiết use case trong hình 3.11 minh họa đường dẫn chính của việc đăng ký thành viên. Chúng ta sẽ vẽ một biểu đồ hoạt động để minh họa dòng nghiệp vụ này như trong hình 10.33.



Hình 10.33: Dòng nghiệp vụ chính trong use case Register car sharer

Bây giờ, bắt đầu từ đường dẫn chính, chúng ta xét các nhánh khác, chẳng hạn xem điều gì sẽ xảy ra nếu không tìm thấy địa chỉ trong cơ sở dữ liệu địa lý. Xem hình 10.34.



Hình 10.34: Một nhánh khác trên use case Register car sharer

Tuy nhiên, như đã nói ở trên, có thể biểu diễn dòng nghiệp vụ này như một dòng con của hoạt động đối sánh địa chỉ với cơ sở dữ liệu địa lý bằng cách gắn một biểu đồ hoạt động vào hoạt động *Match address against geographical database*.

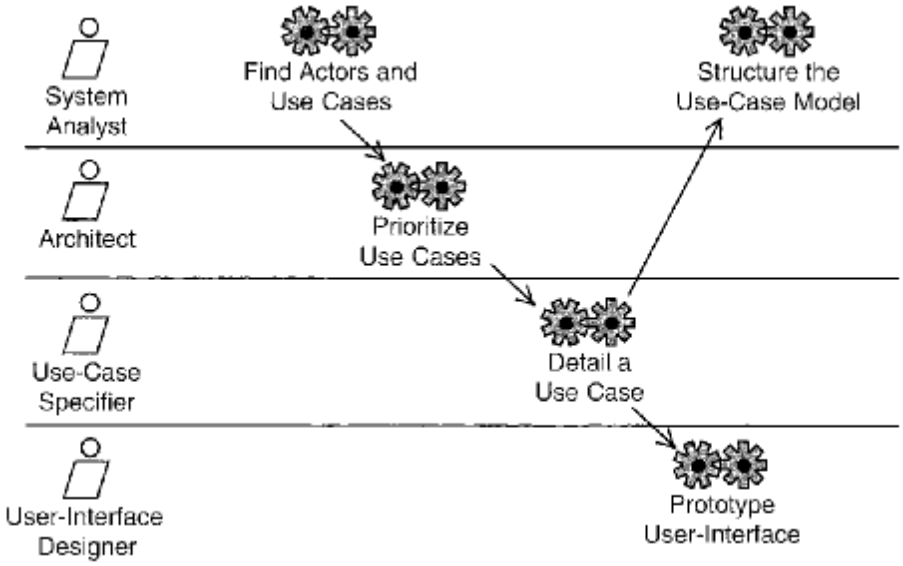
10.6 Quan hệ với các biểu đồ khác

Biểu đồ hoạt động mô tả các dòng nghiệp vụ. Chúng cho phép mô tả các dòng tuần tự và song song. Cùng với biểu đồ tuần tự và song song, biểu đồ hoạt động cung cấp một phương tiện để mô tả hoạt động của hệ thống. Trong một CASE tool (công cụ hỗ trợ phát triển phần mềm, chẳng hạn như Rational Rose), chúng được liên kết như là biểu đồ con của các use case nghiệp vụ, lớp và thao tác.

10.7 Biểu đồ hoạt động trong UP

UP là một tập các dòng nghiệp vụ để xây dựng các hệ thống phần mềm. Chúng là các biểu đồ hoạt động. Dòng *phát triển các yêu cầu* được cho trong hình 10.35.

Bạn có thể thấy rằng đây là một dòng hoạt động mức cao và rất đơn giản. Hầu hết các hoạt động ở đây đều có thời gian thi hành dài và bên trong chúng có rất nhiều thành phần song song. UP định nghĩa năm dòng nghiệp vụ cốt lõi là: mở rộng các yêu cầu, phân tích, thiết kế, cài đặt và kiểm tra. Các dòng nghiệp vụ của UP cung cấp một ví dụ về việc dùng mô hình dòng nghiệp vụ mô tả một hoạt động nghiệp vụ phức tạp ở mức cao. UP là một cách tiếp cận hướng use case, và các use case là các nhóm dòng nghiệp vụ. Trong mô hình nghiệp vụ, việc định nghĩa các use case có thể được hỗ trợ bằng các biểu đồ hoạt động để lập mô hình đầy đủ các dòng nghiệp vụ mà một use case nghiệp vụ biểu diễn. Cũng như thế, các use case hệ thống được mô tả chi tiết bằng cách dùng biểu đồ hoạt động để mô tả các dòng nghiệp vụ phức tạp cần được cài đặt trong một hệ thống máy tính; trong trường hợp này, việc phân tích có thể quyết định dùng biểu đồ tuần tự và biểu đồ cộng tác để mô tả các dòng nghiệp vụ và liên kết chúng tới mô hình đối tượng bên dưới.



Hình 10.35: Requirements workflow

Trong UP, các biểu đồ hoạt động đặc biệt được dùng trong hoạt động *Detail a Use Case* (chi tiết hóa một use case). Trong hoạt động này, có một bước là *Formalizing the Use-Case Descriptions* (hình thức hóa các mô tả chi tiết trong một use case), và các biểu đồ hoạt động có thể được dùng để làm bản mô tả hành vi của các use case nếu một bản mô tả bằng văn bản không phù hợp.

Câu hỏi ôn tập

10.1 Loại dòng (flow) nào có thể được mô tả bằng biểu đồ hoạt động?

10.2 Khi nào bạn nên xem xét đến việc dùng một biểu đồ hoạt động?

10.3 Tại sao bạn vẽ biểu đồ hoạt động?

10.4 Một hoạt động là gì?

10.5 Một hành động là gì?

10.6 Một hoạt động mất bao nhiêu thời gian để hoàn thành?

10.7 Khi nào các hành động xảy ra trong một hoạt động?

10.8 Chuyển tiếp là gì?

10.9 Điều gì gây ra một chuyển tiếp?

10.10 Khi nào một chuyển tiếp không cần kích hoạt xảy ra?

10.11 Trạng thái trong biểu đồ hoạt động là gì? Nó khác với hoạt động như thế nào?

10.12 Cú pháp của các biến cố và hành động xảy ra trên một chuyển tiếp?

10.13 Thế nào là một điểm quyết định?

10.14 Thế nào là một bảo vệ?

10.15 Có bao nhiêu chuyển tiếp có thể đi vào một điểm quyết định?

10.16 Có bao nhiêu chuyển tiếp có thể đi ra khỏi một điểm quyết định?

10.17 Swimlane là gì?

10.18 Tại sao bạn dùng swimlane?

10.19 Thanh đồng bộ hóa là gì?

10.20 Nêu quy tắc về số các chuyển tiếp vào và ra thanh đồng bộ hóa.

10.21 Dòng đối tượng là gì?

10.22 Bạn có thể giữ những thông tin nào về một đối tượng trên dòng đối tượng?

10.23 Tại sao bạn lồng một biểu đồ hoạt động bên trong một hoạt động?

10.24 Biểu đồ hoạt động có thể gắn với phần tử UML nào?

10.25 Các biểu tượng điều khiển là gì? Tại sao phải dùng chúng?

Bài tập có lời giải

10.1 Sau đây là một mô tả một qui trình nghiệp vụ trong CarMatch:

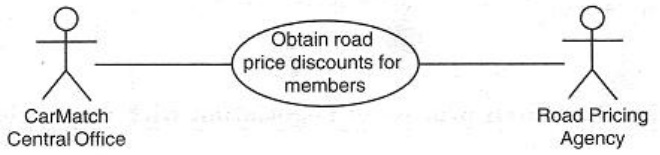
> *CarMatch* dự định rằng các thành viên có thể thu được một khoảng chiết khấu đối với các lược đồ tính cước phí giao thông.

CarMatch thương lượng riêng với từng đơn vị tính phí. Mỗi lược đồ có một hệ thống máy tính riêng. *CarMatch* muốn tiếp quản việc lập hoá đơn cho các thành viên của họ và cố thương lượng việc này trước tiên, nhưng đôi khi không thể. Cuối cùng họ thoả thuận về cơ chế chuyển đổi thông tin. Cần một ít thời gian để thoả thuận các tiêu chuẩn. Phần còn lại của cuộc thương lượng sẽ tạm treo. Nếu cuộc thương lượng thành công, một hợp đồng được phát thảo. Đôi khi phạm vi hợp đồng sẽ phát sinh việc thương lượng lại. Bất kỳ lúc nào, các bên có thể treo hoặc huỷ các cuộc thương lượng.

Hãy phát thảo một dòng công việc cho qui trình này.

Lời giải:

Chúng ta xác định use case nghiệp vụ là *Obtain road-price discounts for members*. Nó bao gồm văn phòng trung tâm của *CarMatch* và chi nhánh định giá giao thông. Chúng ta biểu diễn use case nghiệp vụ này trong hình 10.36.



Hình 10.36: Use case nghiệp vụ để thu xếp với các chi nhánh nhằm thiết lập các chiết khấu cho thành viên

Chúng ta đưa ra một đường dẫn chính qua các bước:

1. Đề xuất với chi nhánh để *CarMatch* lập hoá đơn cho thành viên
2. Thu xếp chiết khấu với chi nhánh
3. Đồng ý cơ chế chuyển đổi dữ liệu.
4. Phác thảo hợp đồng.

và các nhánh thay thế

3.1 Cuộc thương lượng có thể bị dừng lại khi thoả thuận cơ chế chuyển đổi dữ liệu, có thể bắt đầu lại sau này hoặc bị huỷ bỏ.

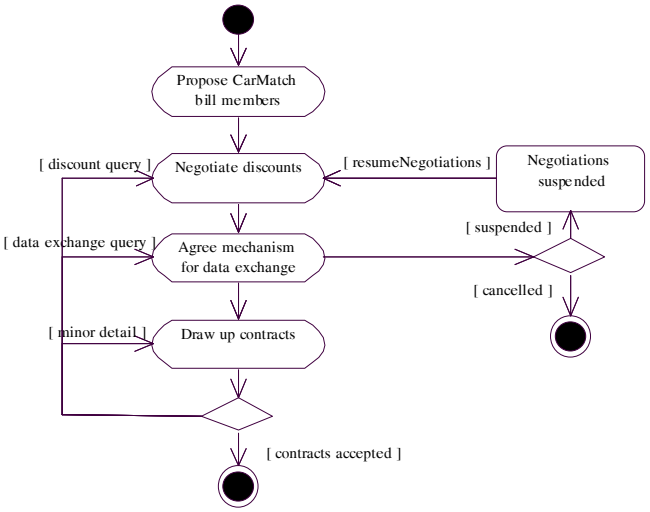
4.1 Có các điều chỉnh nhỏ về hợp đồng cho nên quay về bước 4.

4.2 Có vấn đề về chuyển đổi dữ liệu nên quay về bước 3.

4.3 Có vấn đề về chiết khấu, nên quay về bước 2.

Chúng ta giả định là sau khi dừng cuộc thương lượng, việc thương lượng lại luôn bắt đầu tại điểm thoả thuận chiết khấu. Một giả định khác nằm ở các điều chỉnh hợp đồng, qui trình sẽ quay lại điểm thương lượng nơi phát sinh bàn cãi. Trong thực hành, không có vấn đề gì, bởi các hoạt động khác không có hành động ngoại trừ một số thay đổi sớm trong qui trình. Có quá nhiều điểm quyết định làm rối biểu đồ.

Chúng ta nên xác định nhiều *scenario*, rồi trộn các *scenario* này để cung cấp một dòng công việc phối hợp như trong hình 10.37.



Hình 10.37: Qui trình thương lượng với các chi nhánh

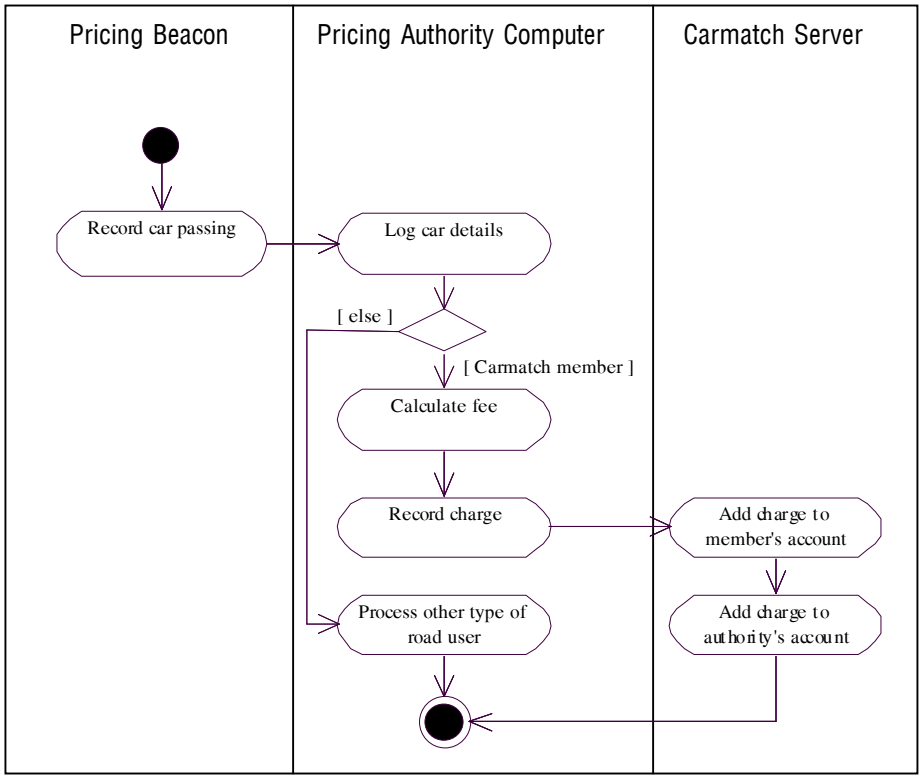
Đây là một qui trình nghiệp vụ ở mức cao, cho nên không có điểm nào xác định đối tượng. Về sau, sẽ có một kết nối thích hợp đối tượng contract đến dòng. Chúng ta có thể muốn phân rã các hoạt động trên thành các hoạt động con. Cũng vậy, ở mức này, chưa thích hợp dùng swimlane khi chưa gán các hoạt động cho các khu vực nghiệp vụ.

10.2 Giả sử *CarMatch* đã thu xếp một giao dịch với cơ quan tính phí giao thông. Bất kỳ khi nào một thành viên đi qua một điểm tính phí, đèn hiệu sẽ ghi nhận biến cố và truyền đến cơ quan này. Cơ quan sẽ tính phí và truyền đến hệ thống *CarMatch*. Để hiểu các kết nối giữa các hệ thống máy tính liên quan, ta cần một biểu đồ hoạt động.

Lời giải:

Sau khi nói chuyện với các bộ phận, ta xác định dãy biến cố sau đây (kịch bản chính).

1. Thành viên đi ngang qua điểm tính phí. Đèn tín hiệu phát hiện xe của thành viên, ghi lại các chi tiết và chuyển đến máy tính của cơ quan tính phí.
2. Máy tính này xác định đây là một thành viên của *CarMatch* và tính phí (đã được chiết khấu).
3. Máy nạp phí này vào tài khoản của *CarMatch*.
4. Máy thông báo kết quả cho server của *CarMatch*.
5. *CarMatch* nạp phí này vào tài khoản của thành viên.
6. *CarMatch* nạp phí này vào tổng chưa thanh toán cho cơ quan tính phí.



Hình 10.38: Quá trình nạp phí giao thông qua CarMatch

Có 3 khu vực kỹ thuật: đèn tín hiệu, máy tính của cơ quan tính phí và server của *CarMatch*. Chúng ta dùng các swimlane cho ba khu vực này và đưa ra một dòng được mô tả trong hình 10.38. Chú ý đây là một quá trình được thực hiện trong một khoảng thời gian dài. Việc chuyển từ đèn tín hiệu đến máy tính của cơ quan được thực hiện theo thời gian thực hoặc theo lô một cách đều đặn. Việc chuyển đến server của hầu như chắc chắn được thực hiện theo lô.

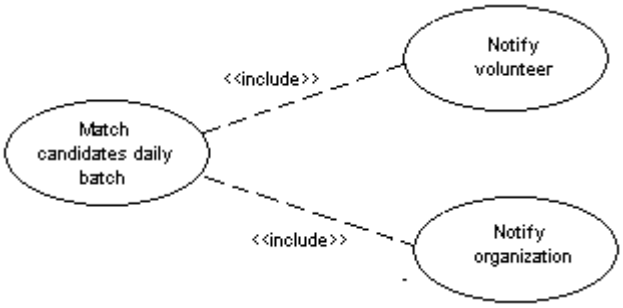
Giai đoạn sau sẽ xác định các use case hệ thống từ dòng nghiệp vụ này, các use case hệ thống này có thể có các biểu đồ hoạt động hoặc các biểu đồ tuần tự riêng mô tả các dòng bên trong chúng.

Bài tập bổ sung

10.3 Xét case study *Volbank* trong chương 1, phác thảo quá trình kết hợp tình nguyện viên với tổ chức tình nguyện trong khu vực của người này và đưa ra một bản thoả thuận. Giả sử một tình nguyện viên không thể kết hợp với hơn một tổ chức tại cùng một thời điểm, nhưng một tổ chức tình nguyện có thể dùng nhiều tình nguyện viên, lên đến một giới hạn đã định.

10.4 Sau khi phân tích qui trình nghiệp vụ trong bài tập 10.3, có một quá trình xử lý theo lô, được thực hiện hàng ngày, nhằm kết hợp

tình các nguyện viên với các tổ chức tình nguyện. Một tổ chức có tối đa 3 ứng viên được phỏng vấn cùng lúc. Quá trình này được phác thảo như một use case, *Match candidates daily batch*, bao gồm hai use case thông báo, *Notify member* và *Notify organization*, xem hình 10.39.



Hình 10.39: Use case hệ thống cho Match volunteers with organizations

Các đường dẫn chính ngang qua use case này là:

1. Lấy tình nguyện viên kế tiếp
2. Kết hợp với tổ chức.
3. Đánh dấu tình nguyên viên được thông báo tham dự phỏng vấn
4. Tăng số cuộc phỏng vấn đã được sắp xếp
5. Thông báo cho tình nguyện viên
6. Thông báo cho tổ chức
7. Nếu có tình nguyện viên chưa xét, quay về bước 1

Có 3 lựa chọn được xác định là

2.1 Ứng viên sẵn sàng cho cuộc phỏng vấn.

2.2 Ứng viên được hỏi riêng không được xét bởi tổ chức.

2.3 Ứng viên không kết hợp được.

Hãy đưa ra một biểu đồ hoạt động làm tài liệu cho các dòng thông qua use case này.

10.5 Với case study *Volbank*, hãy đưa ra một dòng công việc bao gồm việc đăng ký của một tình nguyện viên trên web và việc chuyển dữ liệu đến server của Volbank.

10.6 Với case study Volbank, xét use case Notify member trong bài tập 10.4. Hãy đưa ra một biểu đồ hoạt động mô tả việc gởi thư với ngày, địa chỉ tình nguyện viên, chi tiết tổ chức và mô tả hành động mà tổ chức này tìm kiếm.

Chương 11
BIỂU ĐỒ TRẠNG THÁI

11.1 Giới thiệu

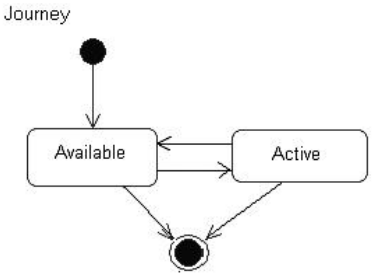
Biểu đồ trạng thái (statechart diagram) là phương tiện mô tả hành vi của các phần tử của mô hình động, nó quan hệ mật thiết với biểu đồ hoạt động. Trong khi biểu đồ hoạt động mô tả *dòng công việc*, thì biểu đồ trạng thái mô tả sự thay đổi *trạng thái* của các thể hiện. Ví dụ, máy điện thoại có thể có các trạng thái: đang gác máy, đang quay số, máy bận và bị ngắt kết nối. Chúng ta có thể dùng một biểu đồ trạng thái để liên kết các trạng thái với nhau và xác định các dòng hợp lệ trong hệ thống. Khi một phần tử đang ở một trạng thái nào đó, công việc có thể được tiến hành hoặc không. Các trạng thái là cách nhìn logic của một thực thể.

Biểu đồ trạng thái thường được dùng để mô tả hành vi của các lớp, tuy nhiên chúng cũng được dùng để mô tả hành vi của các phần tử khác, như use case, actor, hệ thống con, và thao tác. Các biểu đồ trạng thái còn được dùng trong qui trình phân tích để mô tả hành vi phức tạp của các phần tử trong môi trường hệ thống, chẳng hạn như hành vi của một khách hàng. Chúng cũng được dùng ở mức thiết kế để mô tả hành vi của các lớp *control*, các lớp *boundary* (ví dụ màn hình), hoặc các lớp *entity* phức tạp (ví dụ tài khoản). Biểu đồ trạng thái là một trong các biểu đồ UML được dùng để mô hình hóa các dòng (các biểu đồ khác là biểu đồ hoạt động, biểu đồ cộng tác và biểu đồ tuần tự). Biểu đồ trạng thái mô hình các hành vi từ góc độ một thực thể, chẳng hạn như một lớp, còn biểu đồ hoạt động và biểu đồ cộng tác có thể lập mô hình hành vi của nhiều thực thể.

Hình 11.1 minh hoạ một biểu đồ trạng thái đơn giản của lớp *Journey* trong hệ thống *CarMatch*. Khi lộ trình được tạo, trạng thái của nó là *Available* (sẵn sàng) Khi tìm thấy một thành viên và đạt được thoả thuận, trạng thái của lộ trình trở thành *Active* (hoạt động). Sau đó thoả thuận kết thúc, có thể do một thành viên rút ra, lộ trình trở về trạng thái *Available*. Ngoài ra, lộ trình cũng có thể bị kết thúc bất kỳ lúc nào.

Chương này sẽ giới thiệu các ký hiệu về trạng thái, chuyển tiếp và biến cố; liên kết chúng với các thành phần UML khác. Vài khái niệm là tương

tự với biểu đồ hoạt động. Thuật ngữ *máy trạng thái* (state machine) cũng được sử dụng cho biểu đồ trạng thái.



Hình 11.1: Biểu đồ trạng thái đơn giản mô tả lớp Journey

11.1.1 Trạng thái (*state*)

Các hệ thống và các thực thể bên trong, chẳng hạn như các đối tượng, có thể được theo dõi khi chuyển từ trạng thái này sang trạng thái khác. Các biến cố bên ngoài có thể phát sinh một hoạt động làm thay đổi trạng thái của hệ thống và một số thành phần của nó. Trong một hệ thống nghiệp vụ ngân hàng, tiền có thể được rút khỏi một tài khoản và tài khoản này chuyển từ trạng thái có tiền sang trạng thái hết tiền. Tại một thời điểm, một phần tử có thể có nhiều trạng thái. Các trạng thái cũng có thể được lồng vào nhau.

Việc xem một hệ thống là một tập các trạng thái và các chuyển tiếp giữa các trạng thái là một cách hữu dụng để mô tả các hành vi phức tạp. Đây là một cách để mô tả các đường đi hợp lệ trong một hệ thống. Việc hiểu các chuyển tiếp hợp lý từ trạng thái này sang trạng thái khác là một bộ phần chính trong qui trình phân tích và thiết kế.

11.1.2 Biến cố (*event*)

Các hệ thống được điều khiển bởi các *biến cố*, trong và ngoài. Chúng được hệ thống đáp ứng rồi phát sinh tiếp các biến cố khác. Một biến cố ngoài thường phát sinh một loạt các biến cố trong và đôi khi cũng phát sinh các biến cố ngoài. Ví dụ, một khách hàng đang gửi tiền bằng ngân phiếu vào tài khoản tiết kiệm của mình sẽ phát sinh một *biến cố trong* đặt một khoản gửi vào tài khoản của mình, và phát sinh một *biến cố ngoài* yêu cầu chuyển khoản từ nơi phát hành ngân phiếu.

Trong UML, thông tin kèm theo biến cố được biểu diễn như các đối số. Thông tin này sẽ luân chuyển trong hệ thống qua các biến cố. Các đối số này có thể được dùng một phần trong việc quyết định trạng thái kế tiếp khi xảy ra một biến cố. (Ở đây chúng ta cần phân biệt giữa tham số và

đối số. *Chữ ký* của một thao tác hoặc một biến cố bao gồm các tham số, tham số định nghĩa tên và kiểu của các giá trị truyền cho thao tác hoặc biến cố; còn đối số là các giá trị thực sự liên kết với một thể hiện cụ thể của thao tác hoặc biến cố đó).

Các biến cố phải được đối tượng nhận ra dưới dạng các thao tác (một số ngôn ngữ lập trình chứa cả biến cố, và các thao tác được cài đặt như là các trình xử lý biến cố), qui trình thiết kế sẽ khảo sát tất cả các biến cố trong biểu đồ trạng thái và xem xét cách các đối tượng sẽ hỗ trợ chúng như thế nào.

11.2 Mục đích của kỹ thuật

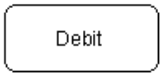
Vai trò chính của biểu đồ trạng thái là mô tả các thực thể phức tạp có các trạng thái có ý nghĩa, và các chuyển tiếp phức tạp giữa các trạng thái. Chúng đặc biệt hữu ích để mô tả:

- Các thực thể nghiệp vụ phức tạp, như *khách hàng* và *tài khoản*.
- Hành vi của các hệ thống con.
- Tương tác trong các lớp *boundary* trong suốt qui trình định nghĩa giao diện của hệ thống, chẳng hạn như màn hình.
- Việc thực hiện các use case.
- Các đối tượng phức tạp hiện thực các thực thể nghiệp vụ hoặc các thực thể thiết kế phức tạp.

11.3 Ký hiệu

Tên của thực thể đang được mô hình (lớp, use case hoặc hệ thống con) được hiển thị như một nhãn trên biểu đồ, bình thường được đặt ở góc trên bên trái của biểu đồ, như trong hình 11.1.

11.3.1 Trạng thái (state)



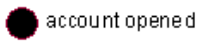
Hình 11.2: Trạng thái Debit của Account

Một trạng thái trong biểu đồ trạng thái là một điểm trong chu trình sống của một phần tử thoả mãn một số điều kiện, đang thực hiện một số hành động hoặc đang chờ một số biến cố. Ký hiệu trạng thái là một hình chữ nhật với bốn góc được bo tròn, bên trong là tên của trạng thái. Hình 11.2 là trạng thái *Debit* của lớp *Account*. Một trạng thái có thể kéo dài

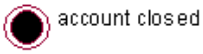
trong một khoảng thời gian mặc dù trong thực hành có một số trạng thái tồn tại rất ngắn.

Các trạng thái có thể là *đơn* hoặc *phức*. Một trạng thái *đơn* sẽ không bị phân rã trong tương lai, với đối tượng chúng được biểu diễn bởi các giá trị của các thuộc tính. Một trạng thái *phức* sẽ bị phân rã bằng các biểu đồ trạng thái lồng nhau hoặc nhúng các trạng thái khác vào trong cùng một biểu đồ duy nhất.

Có hai trạng thái đặc biệt. Thứ nhất là trạng thái bắt đầu, điểm vào dòng. Trên một biểu đồ, chỉ được phép có một trạng thái bắt đầu, được biểu diễn bằng một chấm đen và có thể có nhãn như trong hình 11.3. Thứ hai là trạng thái kết thúc, trong một biểu đồ có thể có nhiều trạng thái kết thúc. Trạng thái này được biểu diễn bằng một chấm đen với một vòng tròn bao bên ngoài như trong hình 11.4.



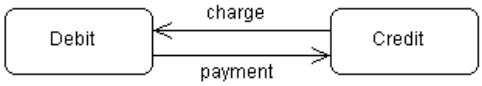
Hình 11.3: Trạng thái bắt đầu



Hình 11.4: Trạng thái kết thúc

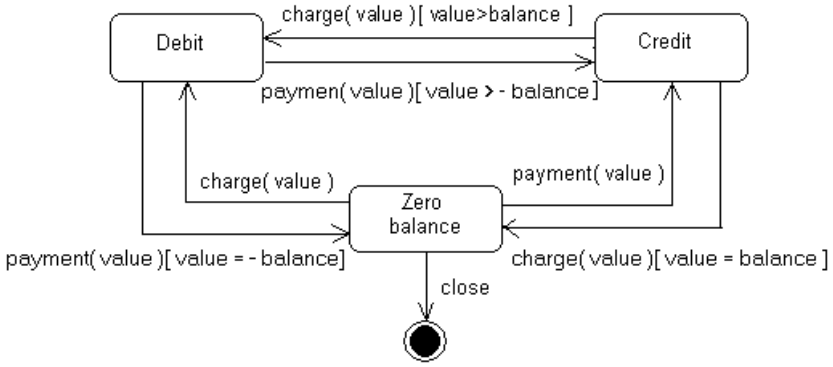
11.3.2 Các chuyển tiếp (*transition*)

Một *chuyển tiếp* là sự di chuyển giữa các trạng thái. Các chuyển tiếp được biểu diễn bằng các mũi tên giữa các cặp trạng thái như trong hình 11.5. Biểu đồ này minh hoạ một thể hiện của *Account* có thể thay đổi giữa các trạng thái *Credit* và *Debit*. Trong một biểu đồ trạng thái, các chuyển tiếp luôn xuất hiện để đáp ứng một biến cố nào đó. (Trong biểu đồ hoạt động, các chuyển tiếp ra khỏi hoạt động xảy ra khi hoạt động hoàn tất, đây là chuyển tiếp không cần kích hoạt). Tên của biến cố gây ra việc chuyển tiếp được ghi bên cạnh mũi tên.



Hình 11.5: Các chuyển tiếp giữa trạng thái Credit *và* Debit *của* Account

Một trạng thái có thể có một hoặc nhiều chuyển tiếp đi ra, xảy ra khi có nhiều biến cố kết thúc trạng thái hoặc khi có nhiều *điều kiện bảo vệ*. Trong hình 11.6, có ba biến cố kết thúc trạng thái *Zero balance* là *charge*, *payment* và *close*. Các điều kiện bảo vệ được thêm vào chuyển tiếp nằm trong cặp móc vuông. Chuyển tiếp chỉ xảy ra nếu điều kiện thoả mãn. Trong hình 11.6, việc đổi trạng thái từ *Debit* sang *Zero balance* hay *Credit* là kết quả của việc thanh toán phụ thuộc vào giá trị thanh toán (value) và số dư (balance) của tài khoản



Hình 11.6: Nhiều chuyển tiếp từ các trạng thái của lớp Account

11.3.3 Hành động (action)

Một trạng thái có thể phát sinh các *hành động* theo năm cách:

- *On Entry*: hành động được phát sinh ngay khi trạng thái bắt đầu.
- *Do*: hành động được phát sinh trong suốt thời gian tồn tại của trạng thái.
- *On Event*: hành động được phát sinh để đáp ứng một biến cố.
- *On Exit*: hành động được phát sinh trước khi trạng thái kết thúc.
- *Include*: gọi một *máy trạng thái con*, được biểu diễn bởi một biểu đồ trạng thái khác.

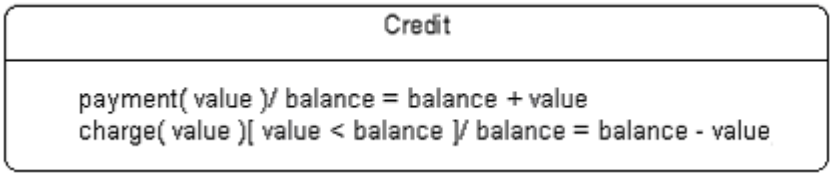
Hành động được biểu diễn trong biểu đồ như trong hình 11.7. Tên trạng thái được đặt ở ngăn trên cùng, tách biệt với danh sách hành động bên dưới bằng một đường thẳng nằm ngang. Cú pháp cho hành động như sau:

action-label / *action*

Trong đó, *action-label* là *entry, do, exit, include* hoặc tên *biến cố*. Trong trường hợp là tên biến cố, cú pháp của hành động là:

event-name (parameters) [guard-condition] / *action*

Trong đó *parameters* là danh sách tham số cách nhau bởi dấu phẩy, *guard-condition* là điều kiện, điều kiện này phải thoả để biến cố phát sinh hành động.



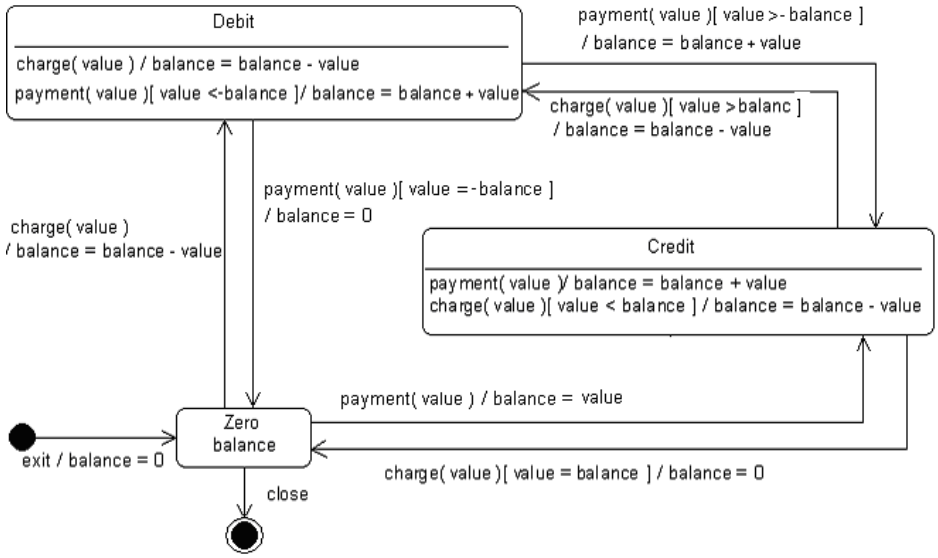
Hình 11.7: Các hành động trong trạng thái Credit *của lớp* Account

Hành động có thể được mô tả bằng một phát biểu bằng tiếng Anh đơn giản, hoặc đối với thiết kế khi tiết hơn thì dùng các lệnh của một ngôn ngữ lập trình. Một dạng khác của hành động là phát sinh biến cố, điều này được biểu diễn dưới dạng:

^target.event (parameters)

Trong đó *target* là đối tượng nhận biến cố, *event* là tên biến cố và *parameters* là các thông tin đi cùng biến cố.

Các chuyển tiếp có thể gây ra hành động. Cú pháp tương tự và được viết ngay cạnh mũi tên. Nếu chỉ có một chuyển tiếp ra khỏi trạng thái, chắc chắn đó là hành động kết thúc trạng thái, nhưng nếu có nhiều cách kết thúc trạng thái với các hành động khác nhau, chúng phải được gắn với các chuyển tiếp thích hợp.

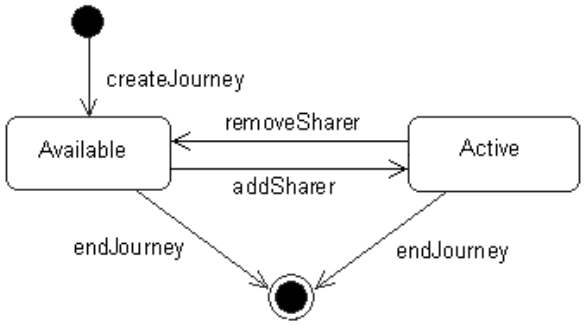


Hình 11.8: Mô tả đầy đủ các trạng thái của lớp Account

Trong hình 11.8, chúng ta thấy một mô tả đầy đủ các trạng thái của lớp *Account*. Khi một tài khoản được mở, hành động kết thúc *trạng thái bắt đầu* sẽ thiết lập số dư tài khoản là 0 và chuyển tài khoản đến trạng thái *Zero balance*. Một thanh toán sẽ chuyển tài khoản sang trạng thái *Credit* và một khoản chi sẽ chuyển tài khoản sang trạng thái *Debit*. Số dư dương vẫn để tài khoản ở trạng thái *Credit* cho đến khi một khoản chi làm nó hoặc quay về trạng thái *Zero balance* hoặc chuyển sang trạng thái *Debit*. Số dư âm vẫn để tài khoản ở trạng thái *Debit* cho đến khi một thanh toán làm nó hoặc quay về trạng thái *Zero balance* hoặc chuyển sang trạng thái *Credit*. Các biến cố giống nhau có thể giữ nguyên trạng thái hoặc chuyển sang trạng thái khác phụ thuộc vào điều kiện bảo vệ (ví dụ biến cố thanh toán trên trạng thái *Debit*)

11.3.4 Các trạng thái phức hợp

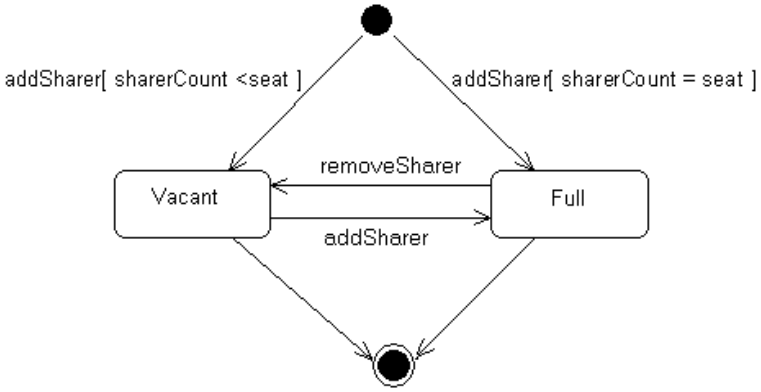
Một *trạng thái phức hợp,* hoặc *trạng thái phức* cho ngắn gọn*,* có thể được phân rã thành các *trạng thái con*. Các trạng thái con có thể được vẽ bên trong hoặc trong một biểu đồ trạng thái riêng. Xem biểu đồ trạng thái của lớp *Journey* trong hình 11.9.



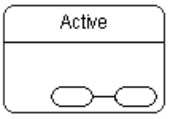
Hình 11.9: Biểu đồ trạng thái của lớp Journey

Khi một lộ trình được tạo, vì có một thành viên muốn chia sẻ lộ trình này, nên nó có trạng thái *Available*. Cuối cùng, khi có một thành viên khác đồng ý chia sẻ lộ trình này, nó chuyển sang trạng thái *Active*. Một trong hai thành viên có thể huỷ bỏ giữa chừng, và lộ trình trở về trạng thái *Available* vì chỉ có một thành viên đăng ký lộ trình này. Khi lộ trình ở trạng thái *Active*, các thành viên khác có thể gia nhập vào lộ trình. Trạng thái *Active* có thể có hai trạng thái con, là *Vacant* (khuyết) cho phép thành viên khác gia nhập vào lộ trình, và *Full* (đầy) cho biết rằng lộ trình này đã đầy không thể nhận thêm người nào nữa. Chúng ta có thể vẽ một biểu đồ trạng thái để biểu diễn hành vi này như trong

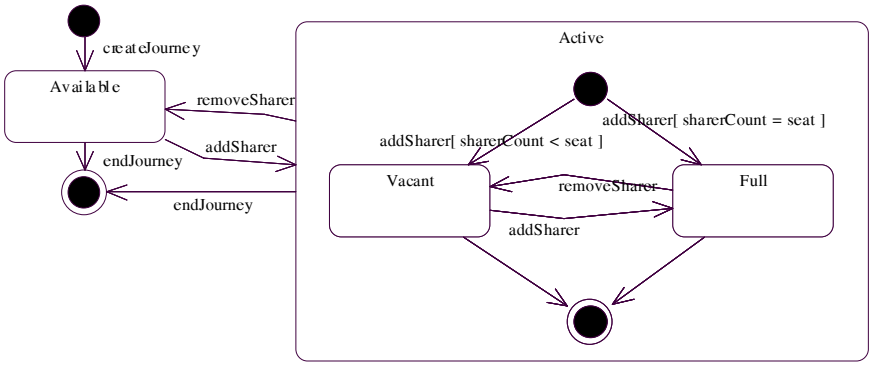
hình 11.10. Bây giờ chúng ta có hai chọn lựa để thay đổi biểu đồ hình 11.9 bằng cách cập nhật trạng thái *Active* như trong hình 11.11 hoặc thêm các trạng thái con vào biểu đồ như trong hình 11.12.



Hình 10.10: Phân rã trạng thái Active



Hình 11.11: Ký hiệu biểu diễn một dòng con của một trạng thái



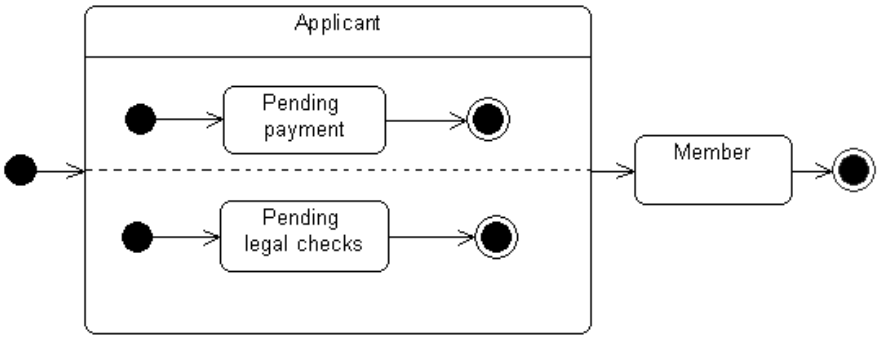
Hình 11.12: Dòng con được vẽ bên trong một trạng thái phức trong biểu đồ trạng thái của lớp Journey

Trong ví dụ này, một thể hiện của lớp *Journey* trong trạng thái *Active* phải ở trong trạng thái *Vacant* hoặc *Full*. Lưu ý sự khác nhau giữa điều này với ký hiệu của các *trạng thái con đồng hành* trong mục sau.

11.3.5 Các trạng thái con đồng hành (concurrent substate)

Một trạng thái *phức* có thể gồm nhiều *trạng thái con đồng hành*, xem hình 11.13. Khi một người đăng ký làm thành viên, người đó được xem là một ứng viên và chưa thể tham gia vào các thoả thuận dùng chung xe. Qui trình đăng ký liên quan đến qui trình thanh toán và các kiểm tra pháp lý để bảo đảm ứng viên không phạm luật hoặc không bị cấm thực hiện các hoạt động này. Điều này có thể mất thời gian vì thế trạng thái *Applicant* của một thể hiện của *CarSharer* phải tiếp tục cho đến khi ra khỏi hai trạng thái con này. Các trạng thái con là độc lập và có thể kết thúc ở các thời điểm khác nhau, điều này được minh họa trên biểu đồ bằng cách chia các dòng đồng hành bằng các đường thẳng không liền nét bên trong trạng thái chứa chúng.

Ngược với ví dụ trước, bạn thấy ở đây thể hiện của lớp *CarSharer* ở trong cả hai trạng thái *Pending payment* và *Pending legal checks*.

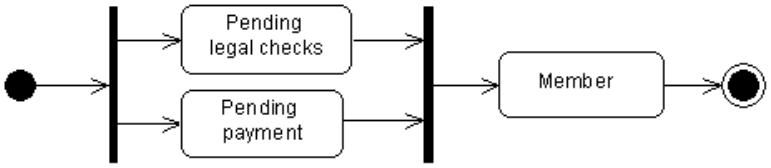


Hình 11.13: Các trạng thái con đồng hành của trạng thái Applicant

11.3.6 Các thanh đồng bộ hóa

Khi có các trạng thái đồng hành và muốn biểu diễn chúng, một *chuyển tiếp* có thể rẻ thành nhiều nhánh và ngược lại các nhánh được kết hợp vào một *chuyển tiếp* duy nhất bằng cách dùng thanh đồng bộ hóa. Dùng lại ví dụ trên, thay vì tạo hai trạng thái con đồng hành của trạng thái *Applicant*, chúng ta sẽ bỏ trạng thái *Applicant* sử dụng hai trạng thái con như trong hình 11.14. Bây giờ chúng ta có một biểu đồ mới trong đó trạng thái *Member* đạt tới khi cả hai trạng thái *Pending legal checks* và *Pending payment* cùng kết thúc.

Một thanh đồng bộ có một *chuyển tiếp* vào và hai hay nhiều *chuyển tiếp* ra hoặc ngược lại có nhiều *chuyển tiếp* vào và một *chuyển tiếp* ra. Điều quan trọng là toàn bộ trạng thái của lớp được chia thành các trạng thái song song, sau đó các trạng thái này được kết hợp lại trên cùng biểu đồ.

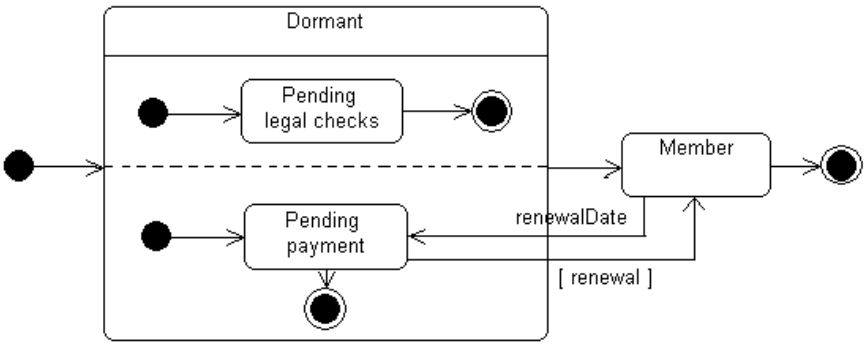


Hình 11.14: Các trạng thái song song được biểu diễn trên biểu đồ trạng thái bằng cách dùng thanh đồng bộ hóa.

11.3.7 Các *chuyển tiếp* đến và đi khỏi trạng thái phức

Một *chuyển tiếp* được vẽ đến đến biên của trạng thái phức sẽ bắt đầu các dòng con của nó. Các hành động *on entry* của trạng thái phức được phát sinh trước, rồi đến các hành động của *trạng thái bắt đầu* của các dòng con được áp dụng thích hợp.

Một chuyển tiếp được vẽ từ biên của một trạng thái phức ảnh hưởng trực tiếp trên tất cả các trạng thái con; nghĩa là tất cả chúng phải kết thúc trước khi thi hành bất kỳ các hành động kết thúc nào của trạng thái *phức*.



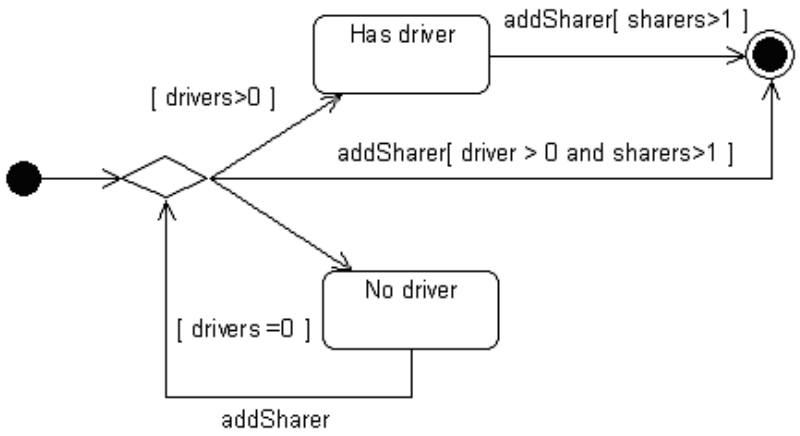
Hình 11.15: Các chuyển tiếp vào và ra các trạng thái con

Các chuyển tiếp có thể được vẽ trực tiếp vào các trạng thái con, hoặc từ một trạng thái con tới các trạng thái khác, xem hình 11.15. Chúng ta dùng lại ví dụ về trạng thái của một ứng viên trước khi có đủ tư cách làm thành viên và gọi trạng thái này là *Dormant* (tiềm tàng). Như trong hình 11.13, khi một ứng viên gia nhập lần đầu, bắt buộc phải có một kiểm tra pháp lý và bắt buộc thanh toán trước khi ứng viên trở thành thành viên. Đến kỳ gia hạn, tài khoản sẽ bị chuyển sang trạng thái *Dormant*, được biểu diễn trong biểu đồ bằng một chuyển tiếp trực tiếp từ trạng thái *Member* sang trạng thái con *Pending payment*. Khi trạng thái *Pending payment* kết thúc, có thể phục hồi lại trạng thái của thành viên

là *Member* mà không xảy ra việc kiểm tra pháp lý, điều này được biểu diễn bằng một chuyển tiếp được bảo vệ, chuyển tiếp này chỉ xảy ra nếu đây là sự gia hạn.

11.3.8 Điểm quyết định

Một điểm quyết định cho phép một chuyển tiếp được chia thành nhiều chuyển tiếp dựa trên một điều kiện hoặc ngược lại. Xem hình 11.16, đây là mô hình hành vi của một *journey* trước khi nó ở trạng thái *Active*. Đây là bản mô tả chi tiết trạng thái *Available* trong hình 11.9. Hình thoi biểu diễn điểm quyết định ở vị trí chuyển tiếp có thể chia thành nhiều chuyển tiếp tuỳ vào điều kiện. Khi bắt đầu, điểm quyết định sẽ được tạo ra để xem có tài xế hay chưa. Nếu chưa, lộ trình có trạng thái *No driver*. Khi thêm thành viên, chuyển sang trạng thái trước điểm quyết định. Nếu người mới là một tài xế thì trạng thái *Available* hoàn tất, ngược lại nó sẽ quay lại trạng thái *No driver*. Nếu người đầu tiên là tài xế, chuyển trạng thái sang *Active*, chỉ cần đợi một người thứ hai, bất kể người này có khả năng lái xe hay không.



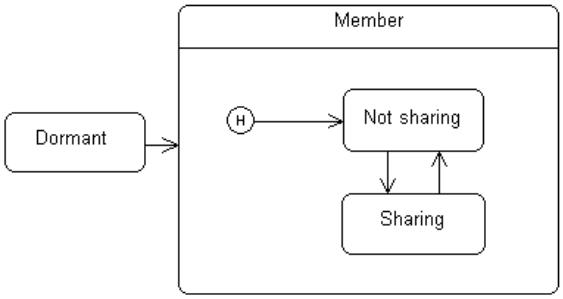
Hình 11.16: Một điểm quyết định trên biểu đồ trạng thái

11.3.9 Các trạng thái lịch sử (history)

Khi bước vào trạng thái *phức*, nói chung sẽ bắt đầu ở *trạng thái vào* hoặc một trạng thái con nào đó được chỉ định bởi chuyển tiếp. Tuy nhiên, đôi khi rất có ích khi vào lại nơi nó đi ra trước đó. Điều này được thực hiện bằng cách thêm vào một *trạng thái history*, được biểu diễn bằng một vòng tròn có ký tự **H** bên trong. Xem hình 11.17, một thành viên có thể ở trong trạng thái *Sharing* hoặc *Not sharing*. Ở điểm vào lần đầu, thành

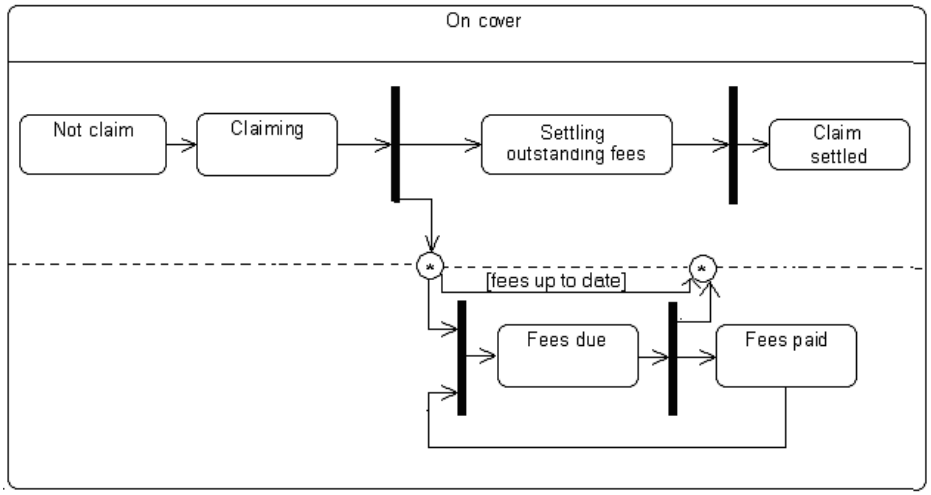
viên ở trạng thái *Not sharing*. Nếu thành viên ở vào trạng thái *Dormant* (tiềm tàng) do trả phí gia hạn trễ, ta cần ngăn chặn thành viên không tham gia một thoả thuận mới nhưng các thoả thuận cũ vẫn được tiếp tục để không ảnh hưởng đến các thành viên khác. Khi thành viên này được kích hoạt lại, trạng thái *Sharing* sẽ được phục hồi như trước đây. Điều này được minh họa bằng trạng thái *history*.

Nếu trạng thái bên trong một trạng thái *phức* lại là trạng thái *phức*, cần phục hồi lại trạng thái bị treo lồng nhau. Nếu một trạng thái *history* chứa từ *H thì đây là một trạng thái *history sâu* và trạng thái con được phục hồi tại điểm treo trước đó. Quá trình này tiếp tục ở mức sâu hơn.



Hình 11.17: Một trạng thái history trong một trạng thái phức

11.3.10 Các trạng thái đồng bộ (synch state)



Hình 11.18: Các trạng thái đồng bộ hóa
trong một trạng thái phức với các dòng con đồng hành

Các dòng song song trong một trạng thái phức đôi khi cần được đồng bộ hoá. Xem hình 11.18, một hợp đồng bảo hiểm có hai dòng đồng hành riêng biệt, một cho việc thanh toán và một cho các yêu cầu. Tuy nhiên yêu cầu chỉ được thiết lập khi tất cả các phí bên ngoài đã được trả. Để mô tả điều này, chúng ta đưa ra hai *trạng thái đồng bộ*, được xác định bởi một vòng tròn với một dấu hoa thị bên trong. Một khi yêu cầu được thực hiện, dòng *claim* bước vào trạng thái *Setting outstading fees*, và vẫn ở đó nếu phí không được cập nhật (chạy song song với trạng thái *Fees due* trong dòng payment). Các trạng thái đồng bộ được xem như là các trạng thái chung của cả hai dòng, chỉ có thể kết thúc khi cả hai dòng đều đi đến chúng.

11.4 Cách tạo biểu đồ trạng thái

Biểu đồ trạng thái mô hình hoá hành vi của các thực thể như lớp, use case hoặc các hệ thống con, và có quan hệ mật thiết với biểu đồ hoạt động. Trong khi biểu đồ hoạt động mô tả các qui trình thì biểu đồ trạng thái lại thích hợp cho việc mô tả lịch sử quá trình sống của các thực thể ở các giai đoạn khác nhau. Biểu đồ trạng thái mô tả một tập các trạng thái và các chuyển tiếp giữa các trạng thái trong một lớp. Mỗi thể hiện có các đường đi riêng qua các trạng thái này, tuỳ vào dãy biến cố ảnh hưởng đến mỗi thể hiện.

Các lớp mà biểu đồ trạng thái lập mô hình có thể là các lớp trong miền phân tích, chẳng hạn như các tác nhân hoặc các đại diện của các phần của hệ thống, hoặc các lớp cần được cài đặt. Biểu đồ trạng thái không cần thiết cho tất cả các thực thể, chúng chỉ cần thiết cho các hành vi phức tạp có thể được biểu diễn như một tập các trạng thái.

Biểu đồ trạng thái được phát triển như sau:

1. Xác định các thực thể có hành vi phức tạp.
2. Định nghĩa trạng thái ban đầu và các trạng thái kết thúc của thực thể.
3. Xác định các biến cố ảnh hưởng đến thực thể.
4. Làm việc từ trạng thái bắt đầu, dò theo tác động của các biến cố và xác định các trạng thái trung gian.
5. Xác định các hành động *on entry* và *on exit* của các trạng thái.
6. Mở rộng các trạng thái bằng cách dùng các trạng thái con khi cần thiết.
7. Nếu thực thể là một lớp, kiểm tra các hành động trong trạng thái có được hỗ trợ bởi các thao tác và các kết hợp của lớp không. Nếu không thì phải mở rộng lớp này.

11.4.1 Xác định các thực thể có hành vi phức tạp

Ở mức phân tích, thực thể là tác nhân nghiệp vụ tương tác với hệ thống và bất kỳ đối tượng thực nào như tài khoản, chính sách và hợp đồng. Các thực thể này có thể thay đổi trạng thái và có mối quan hệ phức tạp giữa các trạng thái. Ở mức thiết kế, một số lớp phức tạp được đưa ra, như các lớp *boundary* biểu diễn màn hình, các lớp *control* quản lý các giao dịch. Các lớp này cũng là các ứng viên cho việc mô hình trạng thái.

Sau khi đã xác định được một thực thể phức tạp, chúng ta cần xác định kiểu biểu đồ thích hợp. Biểu đồ trạng thái thích hợp nhất khi các thực thể có một tập đầy đủ các trạng thái mà chúng sẽ trải qua. Nếu thực thể có qui trình phức tạp, một biểu đồ hoạt động sẽ thích hợp hơn, còn nếu có một sự tương tác phức tạp giữa các đối tượng liên quan, thì biểu đồ tuần tự hoặc biểu đồ cộng tác là thích hợp nhất. Trong một số trường hợp, có thể dùng nhiều biểu đồ khác nhau, nhất là trong các use case phức tạp.

11.4.2 Xác định các trạng thái bắt đầu và kết thúc của thực thể

Câu hỏi đầu tiên cần đặt ra là: "*Một thực thể được tạo ra như thế nào*?". Trong hệ thống *CarMatch*, các ứng viên được đăng ký trước khi họ bắt đầu các thoả thuận dùng chung xe. Câu hỏi kế tiếp là: "*Một đối tượng bị huỷ như thế nào*?". Một thành viên có thể bị mất quyền thành viên do phạm luật hoặc tự bỏ quyền thành viên do di chuyển khỏi vùng, do đau ốm hoặc không hài lòng với dịch vụ.

11.4.3 Xác định các biến cố ảnh hưởng đến thực thể

Các biến cố ảnh hưởng đến một người chia sẻ xe có thể là: đăng ký lộ trình, kết hợp lộ trình với thành viên khác, đồng ý thoả thuận, huỷ bỏ thoả thuận, ngày thanh toán đến, v.v... Nguồn biến cố chính cho các đối tượng là tập các use case. Các use case là nhóm các biến cố cung cấp các chức năng có ý nghĩa cho hệ thống và một use case được thực hiện bởi một nhóm các đối tượng cộng tác với nhau.

11.4.4 Lần theo biến cố và xác định các trạng thái trung gian

Bắt đầu ở trạng thái đầu tiên, đặt câu hỏi *biến cố nào có thể ảnh hưởng đến trạng thái này*. Một ứng viên phải trả phí thành viên và có một kiểm tra pháp lý ngay sau khi đăng ký. Điều này được minh họa trong hình 11.15, trong đó trạng thái *Dormant* (tiềm tàng) có hai trạng thái con thi hành song song tên là *Pending legal checks* (chưa kiểm tra) và *Pending payment* (chưa thanh toán). Tiếp tục theo cách này, kiểm tra xem các biến cố nào làm kết thúc trạng thái *Dormant* để vào trạng thái

Member và biến cố nào làm thoát ra trạng thái *Member* để quay lại trạng thái *Dormant*.

11.4.5 Xác định các hành động vào và ra trạng thái

Khi một thể hiện của *CarSharer* bước vào trạng thái *Pending payment* trong hình 11.15, các hành động là yêu cầu thanh toán, khởi tạo một thể hiện *Account*, ghi nợ phí thành viên vào tài khoản. Một hành động ra (đáp ứng cho biến cố nhận được thanh toán) sẽ ghi có số tiền thanh toán vào tài khoản.

11.4.6 Mở rộng các trạng thái bằng cách dùng các trạng thái con

Một thành viên ở trạng thái *Dormant* vì nhiều lý do, hai trong số các lý do này là đến kỳ thanh toán và cần các kiểm tra pháp lý; hoặc tạm ngưng tư cách thành viên khi đang làm việc ở xa nhà. Mỗi lý do như thế có thể được mô hình hoá thành một trạng thái con của trạng thái *Dormant* trong hình 11.15.

11.4.7 Kiểm tra có tồn tại các thao tác hỗ trợ

Tất cả các hành động phải được cài đặt như các thao tác. Với lớp *CarSharer*, cần các thao tác hỗ trợ *yêu cầu thanh toán*, *nhận tiền thanh toán*, *ghi nhận kết quả của các kiểm tra pháp lý* và một số hành động khác. Các thao tác này có thể cài đặt trực tiếp trên lớp hoặc trên các lớp liên quan. Ví dụ, thao tác xử lý thanh toán được cài đặt trên lớp *Account* chứ không phải trên lớp *CarSharer*.

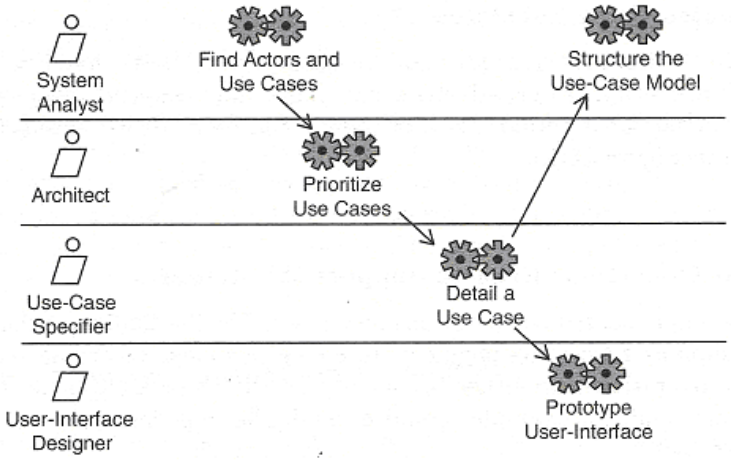
11.5 Quan hệ với các biểu đồ khác

Biểu đồ trạng thái mô tả hành vi, tập trung vào các trạng thái và chuyển tiếp giữa các trạng thái. Chúng cho phép các dòng tuần tự và song song giữa các trạng thái. Bên cạnh biểu đồ hoạt động, biểu đồ cộng tác và biểu đồ tuần tự, biểu đồ trạng thái cung cấp một phương tiện mô tả hành vi động của một hệ thống. Chúng có quan hệ mật thiết với biểu đồ hoạt động. Bên trong một công cụ CASE, biểu đồ trạng thái đươc liên kết như các biểu đồ con của các use case nghiệp vụ, các use case, các lớp và các thao tác.

11.6 Biểu đồ trạng thái trong UP

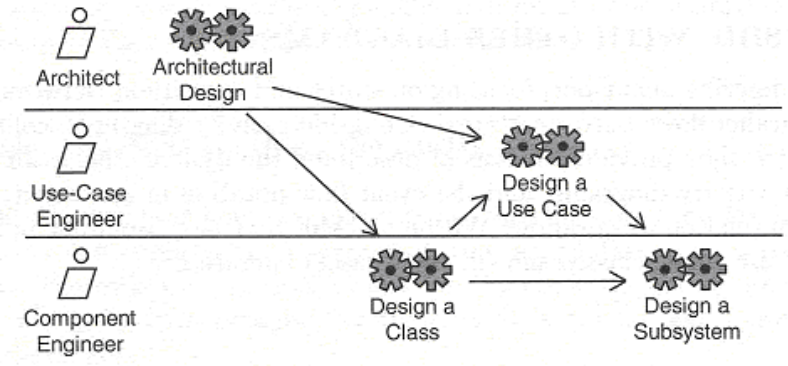
UP là một tiếp cận theo use case, các use case tự nó có các trạng thái phụ thuộc hành vi, các trạng thái này được lập mô hình bằng biểu đồ trạng thái. Trong *dòng công việc xác định yêu cầu* (Requirements Workflow, hình 11.19), các biểu đồ trạng thái được dùng trong hoạt động

chi tiết một use case (Design a use case). Chúng có thể được dùng để định nghĩa hành vi của một use case như một tập các trạng thái và chuyển tiếp giữa các trạng thái tuỳ vào các biến cố được khởi đầu bởi người dùng hoặc từ bên trong hệ thống.



Hình 11.19: Requirements Workflow

Biểu đồ trạng thái được dùng mô tả hành vi của các lớp cộng tác để mô tả một use case. Chúng thường là các lớp với hành vi phức tạp và có ý nghĩa. Việc tạo các biểu đồ trạng thái được thực hiện trong *dòng thiết kế* (Design Workflow, hình 11.20) trong hoạt động *thiết kế một lớp* (Design a class). Tại thời điểm thiết kế các thuộc tính các thao tác và các đặc trưng khác của lớp, biểu đồ trạng thái có thể được dùng để thiết kế và lập sưu liệu các trạng thái phụ thuộc hành vi.



Hình 11.20: Design Workflow

Câu hỏi ôn tập

11.1 Biểu đồ trạng thái dùng để mô hình hoá cái gì?

11.2 Trạng thái là gì?

11.3 Một trạng thái có thể kéo dài bao lâu?

11.4 Các biến cố ảnh hưởng đến trạng thái như thế nào?

11.5 Biểu đồ trạng thái có thể lập mô hình cho các loại thực thể nào?

11.6 Có thể có một chuyển tiếp không cần kích hoạt đi ra khỏi một trạng thái không?

11.7 Năm loại hành động trên một trạng thái là gì?

11.8 Cú pháp của một hành động như thế nào?

11.9 Cú pháp của một nhãn hành động cho một biến cố thế nào?

11.10 Cách biểu diễn một hành động trên trạng thái thế nào?

11.11 Một chuyển tiếp có thể gây ra bao nhiêu hành động?

11.12 Trạng thái phức là gì?

11.13 Mô tả trạng thái phức như thế nào?

11.14 Các trạng thái con đồng hành là gì?

11.15 Thanh đồng bộ hóa là gì?

11.16 Khi một chuyển tiếp hướng trực tiếp đến biên của một trạng thái phức, điều gì sẽ xảy ra?

11.17 Khi một biến cố gây nên một chuyển tiếp từ biên của một trạng thái phức, điều gì sẽ xảy ra?

11.18 Khi một chuyển tiếp đến một trạng thái con của một trạng thái phức, điều gì sẽ xảy ra?

11.19 Điểm quyết định là gì?

11.20 Sự khác nhau giữa các trạng thái *history* chứa H và *H là gì?

11.21 Trạng thái đồng bộ là gì? Cho biết mục đích sử dụng?

Bài tập có lời giải

11.1 Xem lại bài tập 10.1, trong bài tập này thủ tục thiết lập một chiết khấu được mô tả. Hợp đồng (*contract*) là một thực thể nghiệp vụ mà chúng ta sẽ lập mô hình. Hãy vẽ một biểu đồ trạng thái đối cho lớp *Contract*.

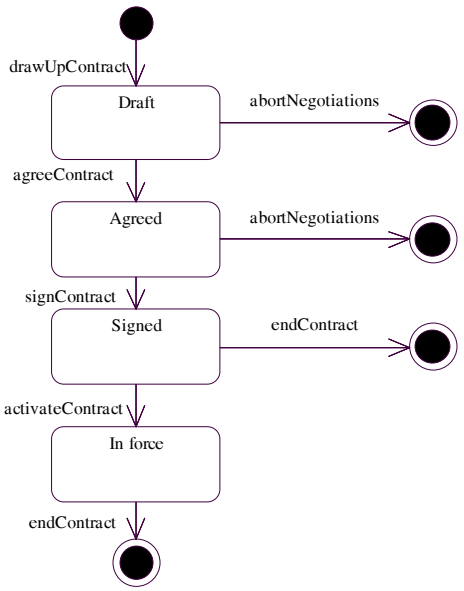
Lời giải:

Giai đoạn đầu tiên là xác định các thực thể cần lập mô hình với biểu đồ trạng thái. Trong việc thương lượng phí giao thông, thực thể chính sẽ là *Contract* và sẽ được

phác thảo, được đồng ý và được ký bởi tất cả các bên tham gia. Tiếp theo chúng ta xem xét các biến cố chính ảnh hưởng đến hợp đồng. Các biến cố này có thể như sau:

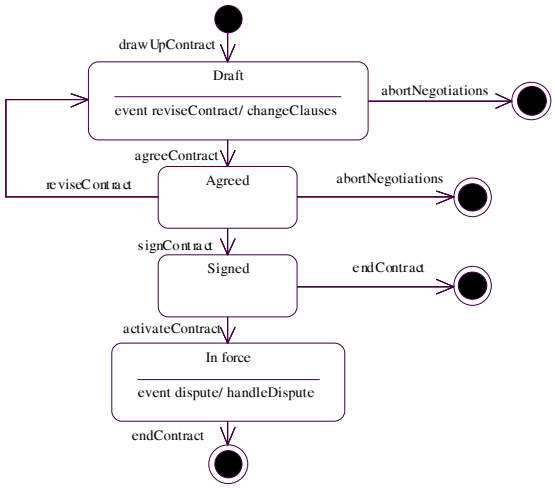
1. Phác thảo hợp đồng: một công việc chi tiết được thực hiện bởi các luật sư, sau khi được phác thảo nó là một bản thảo để các bên xem xét.
2. Xem lại hợp đồng: xảy ra nhiều lần trước khi hợp đồng được ký.
3. Đồng ý hợp đồng: tất cả các bên đều đồng ý hợp đồng trước khi ký kết.
4. Ký hợp đồng: là mốc thời gian khi tất cả các bên đều đồng ý hợp đồng.
5. Bỏ qua thương lượng: trước khi ký kết, tất cả các bên đều rút và hợp đồng kết thúc.
6. Tranh luận: xuất hiện các tranh cãi trong suốt thời hạn hiệu lực của hợp đồng.
7. Kết thúc hợp đồng: có nhiều cách hoặc hợp đồng hết hạn, hoặc một trong các bên rút lui.

Trạng thái bắt đầu khi hợp đồng được phác thảo. Trạng thái kết thúc hoặc khi các thương lượng bị huỷ bỏ, hoặc khi hợp đồng chấm dứt vì bất kỳ lý do nào. Bây giờ chúng ta làm việc từ trạng thái bắt đầu đến trạng thái kết thúc và xem xét các thay đổi. Biểu đồ đầu tiên có lẽ như trong hình 11.21.



Hình 11.21: Biểu đồ trạng thái đầu tiên cho Contract.

Nếu chúng ta xem xét trạng thái *Draft* của *Contract*, chúng ta thấy điều này đáp ứng cho biến cố xem lại hợp đồng; nó không bao gồm một thay đổi trạng thái, chúng ta xem nó như một hành động trong trạng thái. Cũng có thể chấp nhận biến cố này sau thoả thuận nhưng trước lúc ký, xem lại hợp đồng cần ký nếu có trục trặc thì chuyển sang trạng thái *Draft*. Một khi hợp đồng đã được ký, trong thời gian hợp đồng còn hiệu lực có thể có những tranh luận về hợp đồng cần được xử lý. Do đó biểu đồ được mở rộng như trong hình 11.22.



Hình 11.22: Biểu đồ trạng thái cho Contract *với các hành động.*

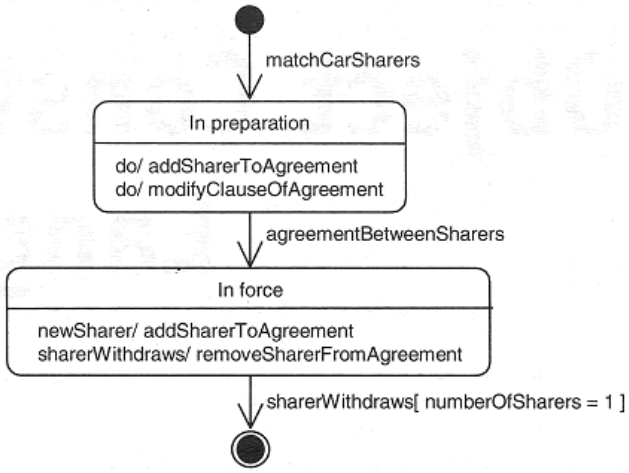
Để cài đặt một hệ thống bảo trì một phiên bản điện tử của hợp đồng, chúng ta nên xem xét các biến cố và hành động vừa xác định bên trên. Các thao tác tạo ra một hợp đồng; phát sinh, cập nhật và huỷ các điều khoản; ghi nhận hợp đồng đã được đồng ý và đã được ký; và ghi nhận lại các tranh luận. Tất cả các thao tác này nên được cài đặt trong lớp *Contract*.

11.2 Trong hệ thống *CarMatch*, một thoả thuận được xây dựng để bảo vệ thành viên đối với vấn đề bảo hiểm và thuế, và để hợp thức hoá quan hệ giữa các thành viên với nhau. Thoả thuận này được phác thảo khi hai (hoặc nhiều) thành viên được kết hợp trên một lộ trình. Lưu ý mỗi thành viên sẽ nhập lộ trình trên hệ thống, và việc kết hợp xảy ra với các lộ trình khác. Việc kết hợp xảy ra khi đã có một thoả thuận và vẫn còn chỗ trống trong xe. Chúng ta đã biết sau khi đăng ký lộ trình có thể đi từ trạng thái *Available* sang trạng thái *Active* khi đạt được một thoả thuận; các lộ trình *Active* có thể hết hoặc còn chỗ. Hãy vẽ biểu đồ trạng thái cho lớp *SharingAggreement*.

Lời giải:

Trạng thái bắt đầu của bản thoả thuận xảy ra khi có hai (hoặc nhiều) thành viên được xác định là cùng một lộ trình. Trạng thái kết thúc khi một người rút lui và chỉ còn lại một. Thoả thuận trước tiên sẽ đi vào trạng thái chuẩn bị (*preparation*), trong suốt trạng thái này những người khác sẽ được thêm vào thoả thuận, và các điều khoản của thoả thuận được điều chỉnh. Một khi thoả thuận được tất cả những người thuê xe chấp nhận, họ tiến hành ký tên vào các bản sao và các bản sao sẽ được mang trở lại *CarMatch*, để sắp xếp và ghi nhận trạng thái của bản thoả thuận vào hệ thống. Trong thời gian thoả thuận còn hiệu lực, các thành viên có thể tham gia hoặc rút lui, và

CarMatch sẽ ghi nhận lại những thay đổi này trong hệ thống. Biểu đồ trạng thái kết quả được biểu diễn trong hình 11.23.



Hình 11.23: Biểu đồ trạng thái cho SharingAgreement

Bài tập bổ sung

11.3 Trong hệ thống *VolBank*, các cơ hội tình nguyện đã được đăng ký và công việc kết hợp sẽ tìm kiếm những người có kỹ năng và thời gian phù hợp với các cơ hội này. Các cơ hội có thể còn trống hoàn toàn, hay còn trống một phần hoặc đã hết chỗ. Đôi khi, các tình nguyện viên có thể từ chối cơ hội hoặc cơ hội có thể cần được trợ giúp thêm. Nếu như cơ hội còn trống hoàn toàn hoặc một phần, thì quá trình phỏng vấn có thể diễn ra. Điều này có liên quan đến việc xác định các tình nguyện viên tiềm năng, mời họ phỏng vấn, xem xét các ứng cử viên và chọn lựa họ. Đối với những vùng nhạy cảm, có thể cần phải xem xét kỹ lưỡng ứng cử viên bằng cách kiểm tra hồ sơ của họ. Hãy vẽ một biểu đồ trạng thái với các trạng thái phức để mô hình lớp *Opportunity*.

11.4 Trong *Volbank*, các tình nguyện viên có thể đăng ký và đi đến các thoả thuận tình nguyện. Trước khi *VolBank* cho phép việc tình nguyện, hệ thống cần kiểm tra và bảo đảm rằng tình nguyện viên không bị ngăn cản vì một lý do nào đó (như bị than phiền về chất lượng kém). Tất nhiên trước khi tình nguyện viên kết hợp với một công việc tình nguyện, người này đã được kiểm tra về kỹ năng cũng như các vấn đề liên quan để đảm bảo có thể đáp ứng tốt yêu cầu của công việc. Tình nguyện viên có thể bị đình chỉ làm thành viên vì một lý do làm việc nào đó, hoặc do di chuyển tạm thời ra

khỏi vùng hoạt động của *VolBank*. Các tình nguyện viên cũng có thể rút lui hoặc đến lúc hợp đồng hết hạn. Họ cũng có thể bị trục xuất vì đạo đức xấu. Hãy vẽ môt biểu đồ trạng thái để mô hình lớp *Volunteer*.

Chương 12
NGÔN NGỮ RÀNG BUỘC ĐỐI TƯỢNG

12.1 Giới thiệu

Ràng buộc là quy tắc cho phép bạn xác định giới hạn trên các thành phần của mô hình. *Warmer* và *Kleppe* (1999) mô tả đó là '*sự hạn chế trên một hoặc nhiều phần của một mô hình hoặc hệ thống hướng đối tượng*'. Các ràng buộc tồn tại trong thế giới thực, như xe hơi không thể chạy quá tốc độ giới hạn, các cử tri phải có tuổi lớn hơn 18. Ràng buộc là các bổ sung quan trọng cho một mô hình nhất là về mặt giới hạn của hệ thống.

Ví dụ, trong hệ thống *CarMatch*, một người không thể có nhiều hơn 10 thoả thuận dùng chung xe. Trong một bản thoả thuận, cần có người lái xe và có chủ xe. Trong hệ thống *VolBank*, một thành viên không thể đề nghị nhiều hơn 5 loại dịch vụ.

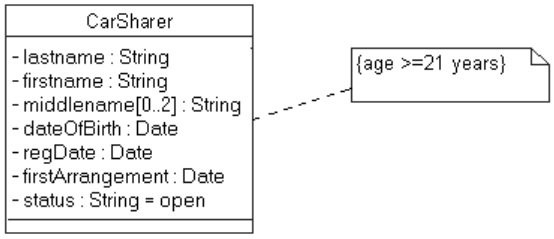
Trong quá trình phân tích một vấn đề, chi tiết của các ràng buộc cần được xét đến. Lúc đầu, các ràng buộc có thể hơi mơ hồ, chẳng hạn như '*một khách hàng không được có một khoản nợ chưa trả nào*'. Tuy nhiên khi phân tích, các ràng buộc sẽ trở nên cụ thể hơn, chẳng hạn như '*trước khi thực hiện một đơn đặt hàng, một khách hàng không thể có các khoản nợ lớn hơn giá trị nợ giới hạn và không có hóa đơn quá hạn thanh toán*'.

Cuối cùng, các ràng buộc cần được mô tả trong một ngôn ngữ có thể chuyển thành các câu lệnh máy tính. UML không đặc tả ngôn ngữ mô tả ràng buộc cũng như cú pháp hoặc ngữ nghĩa của ngôn ngữ cài đặt. Tuy nhiên, các câu lệnh bằng ngôn ngữ tự nhiên có thể trở nên nhập nhằng làm biểu diễn của ngôn ngữ cài đặt không chính xác. Vì thế UML cho phép bao gồm thêm *ngôn ngữ ràng buộc đối tượng OCL* (Object Constraint Language, được đặc tả trong một phụ lục của đặc tả UML, OMG 1999a) để biểu diễn ràng buộc chính xác hơn. Trước đây có rất nhiều loại ngôn ngữ hình thức, chẳng hạn như Z, được dùng để đặc tả các ràng buộc, tuy nhiên các ngôn ngữ này lại nặng nề và khó sử dụng đối với những người không phải là nhà toán học. OCL được thiết kế như một ngôn ngữ hình thức dễ đọc và cung cấp sự đặc tả chính xác mà không quá khó trong xây dựng và giải thích.

Hình 12.1: Một ràng buộc kèm một phụ thuộc trong biểu đồ UML

Trong biểu đồ, một ràng buộc trên một phần tử là một chuỗi ký tự trong cặp dấu ngoặc móc ({}) được đặt gần phần tử này cùng với một quan hệ phụ thuộc. Bạn cũng có thể đặt ràng buộc trong một ghi chú. Ví dụ trong hệ thống *CarMatch*, một lộ trình có hai địa chỉ, một bắt đầu và một kết thúc. Giả sử *CarMatch* chỉ cho đăng ký các lộ trình dài hơn 2 dặm. Có một phụ thuộc giữa *startAddress* và *destinationAddress* được vẽ bằng một mũi tên đứt nét và một ràng buộc trên phụ thuộc này như trong hình 12.1.

Một ràng buộc đơn giản khác là tuổi của thành viên phải lớn hơn 21 vì lý do luật pháp và an toàn, đây là một thuộc tính không thay đổi của mô hình. Ràng buộc này được mô tả bằng một ghi chú đính kèm lớp *CarSharer* chứa ràng buộc như trong hình 12.2. Ràng buộc này áp dụng cho tất cả các thể hiện của lớp này.



Hình 12.2: Ràng buộc trên một lớp
được đính kèm cách dùng một ghi chú trong biểu đồ UML

Một cách tiếp cận khác là giữ các ràng buộc trong một tập tin riêng. Ràng buộc trong hình 12.2 có thể được viết như sau:

context *CarSharer* **inv**:

age >= 21

Theo sau từ khóa *context* (ngữ cảnh) là tên của phần tử áp dụng ràng buộc, và *inv* cho biết ràng buộc này là một *invariant* (bất biến), cú pháp này sẽ được giải thích đầy đủ ở phần sau. Cách này súc tích hơn, mặc dù

vậy các biểu diễn ràng buộc trên biểu đồ như trong hình 12.1 và 12.2 lại thường được dùng hơn.

Chương này sẽ giới thiệu các khái niệm *invariant*, *pre-condition* (điều kiện đầu) và *post-condition* (điều kiện cuối), và giải thích cách biểu diễn chúng như các ràng buộc trong OCL.

12.1.1 Invariant

Invariant là một tính chất luôn luôn đúng trong suốt thời gian sống của một phần tử trong mô hình, chẳng hạn như một đối tượng. Ví dụ, một thành viên phải lớn hơn 21 tuổi, phải có giấy phép lái xe; các lộ trình phải dài hơn 2 dặm.

Lý do để đưa ra *invariant* là để xác định các ràng buộc cơ sở mà hệ thống có thể xử lý. Một hệ thống vẫn có khả năng điều hành khi *invariant* bị vi phạm (lý tưởng là không nên), nhưng người thiết kế sẽ không bảo đảm các kết quả. Một hệ thống máy tính có thể được dùng cho các mục đích không được dự tính nhưng kết quả là không thể đoán trước được.

Khi hệ thống thực hiện một tác vụ, *invariant* là đúng khi bắt đầu và vẫn đúng khi kết thúc tác vụ. Đó là trách nhiệm của nhà thiết kế và người cài đặt để đảm bảo rằng các *invariant* vẫn duy trì cho sự thi hành hợp lệ của hệ thống.

12.1.2 Pre-condition

Một *pre-condition* là một điều đó gì phải đúng trước khi một phần nào đó của hệ thống được thi hành. Điều này được áp dụng cho các thao tác trong suốt qui trình thiết kế và cài đặt, cho các use case trong qui trình phân tích. (Các use case được cài đặt bằng thao tác, vì vậy *pre-condition* được chuyển từ use case sang thao tác). Ví dụ, muốn đăng ký ứng viên phải lớn hơn 21 tuổi, lộ trình phải dài hơn 2 dặm. Vì thế các *pre-condition* được dùng để bảo đảm rằng các *invariant* luôn đúng.

Nhà thiết kế và cài đặt có thể dùng các *pre-condition* để thực hiện các kiểm tra trước khi một thao tác được thi hành. Đây là cách tránh cho hệ thống khỏi rơi vào trạng thái không hợp lệ. Nếu một *pre-condition* bị vi phạm, hành động sẽ phát ra một ngoại lệ và thao tác bị từ chối. Một ví dụ đơn giản là giả sử một người nào đó nhập mã vùng không hợp lệ trong màn hình đăng ký. Thay vì đăng ký, hệ thống sẽ hiển thị một thông báo lỗi lên màn hình và yêu cầu người dùng nhập lại thông tin. Một ví dụ khác là việc bảo đảm có đủ tiền trong tài khoản trước khi rút tiền.

12.1.3 Post-condition

Một *post-condition* là một điều gì đó phải đúng sau khi một phần hệ thống được thi hành, khi ấy việc thi hành là hợp lệ và hệ thống đã thực hiện thành công các hành động của nó. Ví dụ, sau khi đăng ký, các chi tiết về thành viên *phải* được ghi nhận, yêu cầu thanh toán *phải* được gửi đến thành viên này hoặc đến thẻ tín dụng hoặc tài khoản ngân hàng của anh ta, thể hiện *CarSharer phải* sẵn sàng tiếp nhận các lộ trình. Giống như *pre-condition*, *post-condition* được áp dụng cho các thao tác hoặc use case, và vì các use case sẽ được chuyển thành các thao tác trong qui trình phân tích thiết kế, nên cuối cùng các *post-condition* trên use case sẽ được chuyển thành *post-condition* trên các thao tác.

Post-condition là một phần trong định nghĩa những gì mà một thao tác phải làm. Người thiết kế và cài đặt phải nghĩ ra quy tắc bảo đảm các *post-condition* được thoả. Trong một phát triển hình thức (ví dụ như cho các hệ thống tiêu chuẩn an toàn) điều này phải bao gồm việc kiểm chứng chặt chẽ rằng quy tắc sẽ bảo đảm các kết quả trong *post-condition* là đúng nếu *pre-condition* đúng. Có thể kết hợp các kiểm tra khi các thao tác hoàn tất để bảo đảm rằng *post-condition* là đúng.

12.1.4 Thiết kế gọn (design by contract)

Việc thiết kế thành phần hoạt động của hệ thống có thể được thực hiện bằng cách xác định *pre-condition* và *post-condition*. Đây là một thiết kế gọn. Đây không phải là toàn bộ thiết kế mà chỉ là phần thiết kế thuận tiện cho tính cấu trúc của UML. *Pre-condition* và *post-condition* được dùng để xác định làm thế nào các phần của mô hình khớp với nhau và mục đích của chúng là gì. Bước đầu tiên của thiết kế là tập trung vào xác định cấu trúc tổng quan và các thành phần. Bước sau đó là chi tiết hành vi của các thành phần này và cách tương tác giữa chúng với nhau.

Giá trị của OCL ở chỗ nó cung cấp các ghi chú chính xác cho việc định nghĩa các hành vi của các phần trong hệ thống. Cuối cùng bất kỳ hệ thống nào cũng phải được cài đặt bằng một ngôn ngữ máy tính để thi hành một cách rõ ràng và chính xác. OCL cung cấp một bước vững chắc để cài đặt và loại bỏ sự mơ hồ.

12.2 Mục đích của kỹ thuật

OCL được thiết kế để cung cấp một cách thức rõ ràng mô tả các quy tắc về hành vi của các phần tử trong mô hình UML. Các ràng buộc sẽ được ghi nhận lại trên toàn bộ các yêu cầu và các giai đoạn phân tích. Sau đó các ràng buộc được chuyển sang thiết kế và được cài đặt thành các kiểm

tra trên hệ thống đang xây dựng. Mục đích chính của ràng buộc là dùng để:

- Xác định các *pre-condition* và *post-condition* trên các use case và thao tác.
- Mô tả các *invariant* trong thao tác.
- Mô tả các điều kiện bảo vệ trên các *chuyển tiếp*.
- Mô tả các *invariant* cho các lớp và các kiểu trong mô hình lớp.

Mục đích của OCL là:

- Cung cấp một ngôn ngữ rõ ràng để mô tả các ràng buộc.
- Xác định một cách chính xác hành vi của các phần tử trong mô hình.

12.3 Ký hiệu

OCL là một ngôn ngữ mô tả được bảo đảm không có hiệu ứng lề. Nghĩa là các biểu thức không làm thay đổi giá trị của bất kỳ phần tử nào trong mô hình mà chúng chỉ đơn giản là trả về các giá trị. Công việc của nhà phát triển là hiểu các biểu thức OCL và chuyển chúng thành các hành động có ý nghĩa trong một ngôn ngữ lập trình. Vì vậy, OCL không cung cấp các ký hiệu điều khiển. Chúng ta có thể hiểu OCL, về mặt ngôn ngữ lập trình, là các biểu thức được dùng trong các điều kiện để xác định các hành động, chẳng hạn như các điều kiện trong một vòng lặp *while* hay trong một phát biểu *if* chứ không phải là các chỉ thị thực hiện.

OCL là một ngôn ngữ có kiểu, vì thế các câu lệnh OCL phải tuân theo các quy tắc kiểu, chẳng hạn như không được so sánh một số với một chuỗi. Kết quả của các câu lệnh OCL thuộc một kiểu cụ thể, được xác định bằng các quy tắc của OCL để kết hợp các phần tử của một biểu thức.

12.3.1 Qui ước

Tất cả các câu lệnh OCL phải ở trong một ngữ cảnh nào đó, có thể là một lớp hay một use case. Các câu lệnh có thể là các *invariant*, *pre-condition* hoặc *post-condition*. UML qui ước từ khóa ***context*** được in đậm và khuôn dạng của ràng buộc cũng được viết ở dạng in đậm như ***inv*** cho <<***invariant***>>, ***pre*** cho <<***pre-condition***>> và ***post*** cho <<***post-condition***>>. Ví dụ:

context *sharingAgreement* **inv**:

startDate < *finishDate*

12.3.2 Context

Ngữ cảnh (context) của một biểu thức OCL phải là một lớp (đối với các *invariant*) hoặc là một thao tác (đối với *pre-condition* và *post-condition*). Với *invariant*, ngữ cảnh đơn giản chỉ là tên của lớp nếu không bị mơ hồ hoặc tên lớp cùng với tên gói chứa lớp. Với *pre-condition* và *post-condition*, ngữ cảnh là tên lớp và tên thao tác cùng với *chữ ký* của nó (do việc nạp chồng, tên của một thao tác không đủ để xác định thao tác). Cú pháp của một *invariant* như sau:

context *Typename* **inv**:

Trong đó *Typename* là tên của kiểu hoặc lớp mà ràng buộc *invariant* áp dụng. Tên gói được kèm theo bằng cách dùng cú pháp như sau:

context *Packagename::Typename* **inv**:

Các gói có thể được lồng vào nhau, vì thế trong trường hợp gói 1 bị lồng bên trong gói 2, cú pháp là:

context *Packagename1::Packagename2::Typename* **inv**:

Với *pre-condition* và *post-condition* trên thao tác, cú pháp như sau:

context *Typename::operationName (param1: Type1,…): ReturnType*

pre: *constraint*

post: *constraint*

Ví dụ để xác định thao tác *subscribe* trên *CarSharer* trong gói *CarSharers*, ta viết:

context *CarSharers::CarSharer::subscribe (d: Date, fee: Integer): Boolean*

pre: *fee >= SubscriptionRate.fee*

post: *active = true*

Trong đó *SubscriptionRate* là một lớp được kết hợp với *CarSharers* và chứa phí thành viên hằng năm cho các thành viên và *active* là một giá trị boolean được dùng để cho biết thành viên đang hoạt động hay tạm ngưng.

12.3.3 Navigation (điều hướng)

Bên trong một ngữ cảnh, việc chuyển hướng đến một phần tử của mô hình là từ trái sang phải của một đường đi được ngăn cách bởi các dấu chấm. Ví dụ, để tham chiếu đến tài khoản của thành viên, với kết hợp như trong hình 12.3, và đặt một ràng buộc *invariant* trên lớp *CarSharer* là số dư không được âm quá 100$, bạn viết như sau:

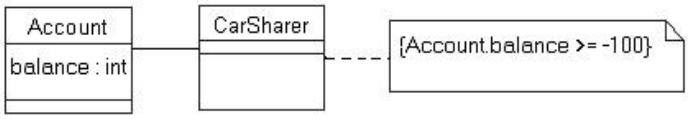
context *CarSharer* **inv:**

Account.balance >= -100

Hình 12.3: Mối kết hợp giữa lớp Account *và* CarSharer

Lưu ý là tên vai trò trên mối kết hợp được dùng để chuyển đến lớp (bị bỏ qua trong ví dụ trên), sau đó đến tên kiểu ở cuối kết hợp, là tên vai trò mặc định được cung cấp để tránh mơ hồ (là *Account* trong ví dụ trên).

Đánh dấu ràng buộc trên mô hình bằng một ghi chú như trong hình 12.4.



Hình 12.4: Mối kết hợp giữa các lớp Account *và* CarSharer
với ràng buộc được đính kèm như một ghi chú

Có 3 cách để tham chiếu đến một phần tử bên trong một ngữ cảnh. Nếu phần tử rõ ràng, tham chiếu như trong ví dụ trên là đủ. Tuy nhiên, nếu biểu thức phức tạp, từ khóa *self* có thể được dùng để tham chiếu đến thể hiện của lớp có tên theo sau ***context***. Vì thế, biểu thức sau tương đương với ví dụ trên là:

context *CarSharer* **inv:**

self.Account.balance>= -100

Cách này có thể được dùng trong các trường hợp có các thể hiện khác trong cùng một lớp liên quan đến các vai trò khác nhau và cần thiết phân biệt cả hai. Trường hợp tham chiếu đến một thể hiện có tên ta dùng cú pháp sau:

context *c*:*CarSharer* **inv:**

c.Account.balance>= -100

Thông thường không nên làm phức tạp các biểu thức theo cách này, mặc dù việc dùng từ *self* làm cho biểu thức dễ đọc hơn.

Với các kết hợp, tên vai trò có thể là một phần của đường dẫn. Giả sử *CarMatch* đưa ra các tài khoản riêng biệt cho thành viên và cho bảo

hiểm. Như thế, có hai kết hợp giữa *CarSharer* và *Account* như trong hình 12.5, với tên vai trò riêng biệt. Để mô tả số dư tài khoản bảo hiểm không âm quá 500$, ta viết biểu thức như sau:

context *CarSharer* **inv**:

insurance.balance>= *-500*

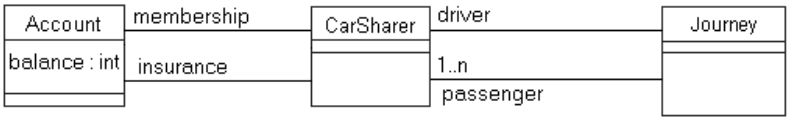
Hình 12.5: Các mối kết hợp giữa CarSharers *và* Account

Lưu ý không cần xác định tên kiểu ở cuối kết hợp khi không có sự mơ hồ.

Có thể điều hướng ngang qua một số kết hợp.Ví dụ, nếu có điều kiện là người lái xe trong lộ trình đã trả tiền bảo hiểm của anh ta, với kết hợp như trong hình 12.6 ta có thể mô tả điều này như sau:

context *Journey* **inv**:

driver.insurance.balance>= 0



Hình 12.6: Các mối kết hợp giữa Journeys, CarSharers *và* Account

12.3.4 Kiểu và biểu thức

Các phần tử của một biểu thức OCL được tạo bởi các kiểu. OCL cung cấp bốn kiểu cơ bản là *Boolean*, *Integer*, *Real* và *String*. Các kiểu khác được lấy từ mô hình. Các giá trị điển hình của bốn kiểu trên được cung cấp trong bảng 12.1.

Kiểu	*Giá trị điển hình*
Boolean	true, false
Integer	0, 1, -1, 123, -256
Real	12.7. –3.6
String	'Fifth Avenue'

Bảng 12.1: Các giá trị điển hình của bốn kiểu cơ bản trong OCL

Các kiểu cơ bản có các phép toán được trình bày trong bảng 12.2. Ngoài ra, có thể sử dụng các phép toán định nghĩa trong mô hình.

Kiểu	*Các phép toán điển hình*
Boolean	And, or, xor, not, implies, if-then-else
Integer	+, - , *, /, và các phép so sánh
Real	+, - , *, /, và các phép so sánh
String	ToUpper, concat

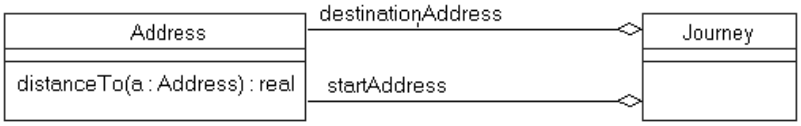
Bảng 12.2: Các phép toán điển hình trên bốn kiểu cơ bản

Kết quả của biểu thức OCL chứa các kiểu cơ bản là một kiểu cơ bản. Kết quả của một biểu thức liên quan đến các kiểu trong mô hình có thể là một kiểu trong mô hình hoặc kiểu cơ bản.

Xem mô hình trong hình 12.7. Chúng ta muốn diễn tả khoảng cách của lộ trình bất kỳ dài hơn hai dặm. Để thực hiện điều này, ta xây dựng một thao tác trên địa chỉ cho phép tính toán khoảng cách tới một địa chỉ bất kỳ. Chúng ta làm rõ điều này bằng biểu thức sau:

context Journey **inv**:

startAddress.distanceTo(*self.destinationAddress*) >= 2



Hình 12.7: Các kết hợp giữa Journey *và các thể hiện* Address *kết tập*

Biểu thức này gọi thao tác *distanceTo* trên địa chỉ bắt đầu, xác định bởi tên vai trò là *startAddress* trên mối kết hợp với một *Address*. Đối số là *địa chỉ đích*, xác định bằng *self.destinationAddress*. Kết quả là một số thực rồi đem so sánh với 2.

Tất cả các kiểu trong một biểu thức được xem là các kiểu con của kiểu *OclType* bất kể chúng là kiểu cơ bản, kiểu tập hợp hay kiểu trong mô hình. OCL có một tập đầy đủ các thao tác trên các kiểu, nhưng với mục đích lập mô hình thông thường ta không cần xử lý trên các kiểu cho nên, chúng không được khảo sát trong chương này. Tuy nhiên, khả năng kiểm tra kiểu của một đối tượng là cần thiết nên được khảo sát.

Một tập các phép toán trên các kiểu cơ bản được cho trong bảng 12.3. Bảng 12.4 là một số biểu thức mẫu và kết quả của chúng.

a	*b*	*Toán tử*	*Kết quả*	*Ý nghĩa*
Integer or Real	Integer or Real	a=b	Boolean	bằng
Integer or Real	Integer or Real	a<>b	Boolean	khác
Integer or Real	Integer or Real	a<b	Boolean	nhỏ hơn
Integer or Real	Integer or Real	a>b	Boolean	lớn hơn
Integer or Real	Integer or Real	a<=b	Boolean	nhỏ hơn hoặc bằng
Integer or Real	Integer or Real	a>=b	Boolean	lớn hơn hoặc bằng
Integer or Real	Integer or Real	a+b	Integer or Real	cộng
Integer or Real	Integer or Real	a-b	Integer or Real	trừ
Integer or Real	Integer or Real	a*b	Integer or Real	nhân
Integer or Real	Integer or Real	a/b	Real	chia
Integer	Integer	a.mod(b)	Integer	chia lấy dư
Integer	Integer	a.div(b)	Integer	chia lấy nguyên
Integer or Real		a.abs	Integer or Real	giá trị tuyệt đối
Integer or Real	Integer or Real	a.max(b)	Integer or Real	lớn nhất
Integer or Real	Integer or Real	a.min(b)	Integer or Real	nhỏ nhất
Integer or Real		a.round	Integer	làm tròn tới số nguyên gần nhất
Integer or Real		a.floor	Integer	làm tròn xuống (cắt bỏ phần dư)
Boolean	Boolean	a or b	Boolean	hoặc
Boolean	Boolean	a and b	Boolean	và
Boolean	Boolean	a xor b	Boolean	loại trừ
Boolean	Boolean	not a	Boolean	phủ định
Boolean	Boolean	a<>b	Boolean	không bằng
Boolean	Boolean	a implies b	Boolean	bao hàm
Boolean	anything	if a then b else b'	Kiểu của b hoặc b'	if then else
String	String	a.concat(b)	String	nối hai chuỗi
String		a.size	Integer	kích thước chuỗi
String		a.toLower	String	chuyển sang chữ thường
String		a.toUpper	String	chuyển sang chữ hoa
String	String	a = b	Boolean	bằng
String	String	a<>b	Boolean	khác

Bảng 12.3: Các phép toán trên các kiểu cơ bản trong OCL

Biểu thức	*Kết quả*
1 + 2	3
3 + 4.5	7.5
(3 + 4) = 7	true
3<4	true
(7.8).floor	7
(7.8)round	8
7.mod (3)	1
7.div(3)	2
true or true	true
true or false	true
false or false	false
(1>2) or (7>6)	true
if (7>6) then 2 else 3	2
if (7<6) then 2 else 3	3
'Washington'.concat('State')	'Washington State'
'Washington'.toLower	'washington'
'New York' = 'new york'	false
'New York'.toLower = 'new york'	true
'Ohio'.size	4

Bảng 12.4: Một số biểu thức mẫu trên các kiểu cơ bản trong OCL

12.3.5 Các kiểu *Set, Bag* và *Sequence* (tập, gói và dãy)

Các kết hợp hiếm khi là *một-một*, bạn thường phải làm việc với *collection* (tập hợp) trong các mô hình hướng đối tượng. OCL cung cấp ba kiểu *collection* như sau:

- *Set*: gồm các phần tử phân biệt, nghĩa là không có sự lặp lại trong, và không có tính thứ tự.
- *Bag*: là một tập hợp không có thứ tự và cho phép lặp lại.
- *Sequence*: là một tập hợp có thứ tự và cho phép lặp lại. Thứ tự ở đây là thứ tự mà các phần tử được lưu chứ không phải thứ tự về giá trị.

Các collection được ghi trong cặp dấu ngoặc móc với tên kiểu đứng trước. Ví dụ:

set{4, 7, 2, 9}

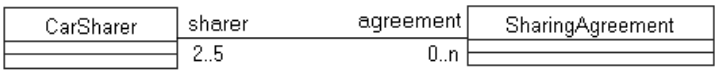
sequence{2, 3, 8, 9, 5, 7, 7, 9}

bag{3, 5, 5, 4, 8, 2, 5}

set{'orange', 'apple', 'banana'}

Nếu có một dãy các số nguyên liên tiếp nhau, có thể biểu diễn chúng theo dạng miền con *m..n*. Ví dụ dãy {2..6} tương đương với {2, 3, 4, 5, 6}

Xem kết hợp trong hình 12.8. Một thành viên có thể có nhiều bản thoả thuận. Một bản liên quan từ 2 đến 5 thành viên. Biểu thức *self.agreement* trong ngữ cảnh *CarSharer* sẽ trả về một *tập* (set) các bản thoả thuận của thành viên này. Biểu thức *self.agreement.sharer* sẽ trả về một *gói* (bag) gồm các thành viên dùng chung xe với anh ta, bao gồm cả thể hiện *CarSharer* của anh ta được lập lại trong *gói* với mỗi bản thoả thuận.



Hình 12.8: Kết hợp giữa các lớp CarSharer *và* SharingAgreement

Các *collection* trong OCL là *phẳng*, nghĩa là nếu bạn đặt một *collection* vào bên trong một *collection* khác, các phần tử sẽ được đưa ra ngoài để hình thành một *collection* kết quả. Ví dụ, {*set*{1, 2}, *set*{3, 4}} trở thành *set*{1, 2, 3, 4}. Lý do trong thực tế các trường hợp như thế này rất hiếm xảy ra và rất khó giải thích về các *collection* nằm bên trong các *collection* khác.

Bảng 12.5 là danh sách các phép toán cho *collection*, còn bảng 12.6 là một số ví dụ.

a	*b*	*Phép toán*	*Kiểu trả về*	*Ý nghĩa*
set	set	a->union(b)	set	tất cả các phần tử trong a hoặc b, không có các phần tử bị trùng lặp
set	bag	a->union(b)	bag	tất cả các phần tử trong a hoặc b, kể cả các phần tử bị trùng lặp
set	set or bag	a->intersection(b)	set	tất cả các phần tử có trong cả a lẫn b, không có các phần tử bị trùng lặp.
set	any	a->including(b)	set	tất cả các phần tử của a thêm vào các phần tử trong b mà không có trong a.
set	any	a->excluding(b)	set	tất cả các phần tử của a trừ ra các phần tử cũng có trong b.
set	set	a – b	set	tất cả các phần tử thuộc a nhưng không thuộc b
set	set	a = b	boolean	true nếu a và b có các phần tử giống nhau, nếu không là false
set	set	a.symetricDifference(b)	set	tất cả các phần tử thuộc a nhưng không thuộc b hoặc thuộc b nhưng không thuộc a
set		a->asBag	bag	bag chứa tất cả các phần tử của a

set		a->asSequence	sequence	sequence gồm các phần tử của a
bag	bag	a->union(b)	bag	tất cả các phần tử trong a và b, kể cả các phần tử trùng lặp
bag	set or bag	a->intersection(b)	set	tất cả các phần tử có trong cả a lẫn b, không có các phần tử bị trùng lặp
bag		a->asSet	set	các phần tử bị trùng lặp trong a sẽ bị loại bỏ
sequence		a->first	kiểu phần tử đầu tiên của a	phần tử đầu tiên của a
sequence		a->last	kiểu phần tử cuối cùng của a	phần tử cuối cùng của a
sequence	integer	a->at(b)	kiểu phần tử tại vị trí b	phần tử của a tại vị trí b
sequence	any	a->append(b)	sequence	nối b vào cuối dãy a
sequence	any	a->prepend(b)	sequence	đặt b vào đầu dãy a
collection	expression	a->select(b)	collection	tập con của a thoả điều kiện b
collection	expression	a->reject(b)	collection	tập con của a không thoả điều kiện b
collection	expression	a->collect(b)	collection	tập kết quả của biểu thức b áp dụng trên mỗi phần tử của a
collection	expression	a->forall(b)	boolean	true nếu b được định trị là true đối với mỗi phần tử trong a, ngược lại là false
collection	expression	a->exists(b)	boolean	true nếu b được định trị là true với ít nhất một phần tử của a, ngược lại là false
collection		a->size	integer	số lượng phần tử của a
collection	any	a->count(b)	integer	số lần b xuất hiện trong a
collection	any	a->includes(b)	boolean	true nếu a chứa b
collection	collection	a->includeAll(b)	boolean	true nếu a chứa tất cả các phần tử của b
collection		a->isEmpty	boolean	là true nếu a rỗng
collection		a->notEmpty	boolean	là true nếu a không ràng buộcỗng
collection		a->sum	integer or real	tổng của tất cả các phần tử trong a (tất cả các phần tử phải là integer hoặc real)
any		a.oclType	OclType	kiểu của a
any	OclType	a.isTypeOf(b)	boolean	là true nếu a có kiểu của b
any	OclType	a.isKindOf(b)	boolean	là true nếu a có kiểu b hoặc kiểu con của b

Bảng 12.5: Một số phép toán trên collection *trong OCL*

a	*b*	*Phép toán*	*Kết quả*
set {1, 2, 3}	set {3, 4, 5}	a->union(b)	set {1, 2, 3, 4, 5}
set {1, 2, 3}	bag {2, 3, 3, 4, 5}	a->union(b)	bag {1, 2, 2, 3, 3, 3, 4, 5}
set {1, 2, 3}	set {3, 4, 5}	a->intersection(b)	set {3}
set {1, 2, 3}	bag {3, 3, 4, 5}	a->intersection(b)	set {3}
set {1, 2, 3}	5	a->including(b)	set {1, 2, 3, 5}
set {1, 2, 3}	1	a->excluding(b)	set {2, 3}
set {1, 2, 3}	set {3, 4, 5}	a – b	set {1, 2}

set {1, 2, 3}	set {3, 4, 5}	a = b	false
set {1, 2, 3}	set {3, 4, 5}	a.symetricDifference(b)	set {1, 2, 4, 5}
bag {1, 2, 2, 3}	bag {3, 4, 5, 5}	a->union(b)	bag {1, 2, 2, 3, 3, 4, 5, 5}
bag {1, 2, 2, 3}	set {3, 4, 5}	a->union(b)	bag {1, 2, 2, 3, 3, 4, 5}
bag {1, 2, 2, 3, 3}	bag {3, 3, 4, 4, 5}	a->intersection(b)	set {3}
bag {3, 3, 4, 5}	set {1, 2, 3}	a->intersection(b)	set {3}
bag {1, 1, 2, 3}	5	a->including(b)	bag {1, 1, 2, 3, 5}
bag {1, 1, 2, 2, 3}	1	a->excluding(b)	bag {2, 2, 3}
bag {1, 1, 2, 2, 3}		a->asSet	set {1, 2, 3}
bag {1, 2, 3}	bag {3, 2, 1}	a = b	true
sequence {1, 2, 3, 2, 1}	sequence {3,4,5,4,3}	a->union(b)	sequence {1,2,3,2,1,3,4,5,4,3 }
sequence {1, 2, 3, 2, 1}	3	a->including(b)	sequence {1, 2, 3, 2, 1, 3}
sequence {1, 2, 3, 2, 1}	3	a->excluding(b)	sequence {1, 2, 2, 1}
sequence {4, 5, 6}		a->first	4
sequence {4, 5, 6}		a->last	6
sequence {4, 5, 6}	integer	a->at(2)	5
sequence {7, 2, 3}	sequence {5, 3, 9}	a->append(b)	sequence {7, 2, 3, 5, 3, 9}
sequence {7, 2, 3}	sequence {5, 3, 9}	a->prepend(b)	sequence {5, 3, 9, 7, 2, 3}
set {2, 4, 5, 7, 9}	x | x>5	a->select(b)	set {7, 9}
set {2, 4, 5, 7, 9}	x | x>5	a->reject(b)	set {2, 4, 5}
set {2, 4, 5, 7, 9}	x | x+1	a->colect(b)	set {3, 5, 6, 8, 10}
set {2, 4, 5, 7, 9}	x | x>1	a->forall(b)	true
set {2, 4, 5, 7, 9}	x | x>3	a->forall(b)	false
set {2, 4, 5, 7, 9}	x | x>1	a->exists(b)	true
set {2, 4, 5, 7, 9}	x | x>9	a->exists(b)	false
set {2, 4, 5, 7, 9}		a->size	5
set {2, 4, 5, 7, 9}		a->include(7)	true
set {2, 4, 5, 7, 9}		a->includesAll(set{2,3})	false
set {2, 4, 5, 7, 9}		a->isEmpty	false
set {2, 4, 5, 7, 9}		a->notEmpty	true
set {2, 4, 5, 7, 9}		a->sum	27
set {2, 2, 4, 5, 7, 7, 9}	x | x>5	a->select(b)	bag {7, 7, 9}
set {2, 2, 4, 5, 7, 7, 9}	x | x>5	a->reject(b)	bag {2, 2, 4, 5}
set {2, 2, 4, 5, 7, 7, 9}	x | x+1	a->colect(b)	bag {3, 3, 5, 6, 8, 8, 10}
set {2, 2, 4, 5, 7, 7, 9}	x | x>1	a->forall(b)	true
set {2, 2, 4, 5, 7, 7, 9}	x | x>3	a->forall(b)	false
set {2, 2, 4, 5, 7, 7, 9}	x | x>1	a->exists(b)	true
set {2, 2, 4, 5, 7, 7, 9}	x | x>9	a->exists(b)	false
set {2, 2, 4, 5, 7, 7, 9}		a->size	7
set {2, 2, 4, 5, 7, 7, 9}		a->include(7)	true
set {2, 2, 4, 5, 7, 7, 9}		a->includesAll(bag{2,3})	false
set {2, 2, 4, 5, 7, 7, 9}		a->isEmpty	false
set {2, 2, 4, 5, 7, 7, 9}		a->notEmpty	true

set {2, 2, 4, 5, 7, 7, 9}		a->sum	36
sequence{2,3,2,4,5,9,6}	x | x>5	a->select(b)	sequence {9, 6}
sequence{2,3,2,4,5,9,6}	x | x>5	a->reject(b)	sequence {2, 3, 2, 4, 5}
sequence{2,3,2,4,5,9,6}	x | x+1	a->colect(b)	sequence {3, 4, 3, 5, 6, 10, 7}

Bảng 12.6: Một số ví dụ về các phép toán trên tập hợp

Ý nghĩa của các phép toán này tương đối dễ hiểu. Tuy nhiên có một số phép toán phức tạp cần được giải thích thêm, sẽ được trình bày dưới đây.

12.3.6 Phép toán *Select*

Phép toán này nhặt ra các phần tử từ một *collection* cho trước thoả một điều kiện nào đó. Cú pháp của *select* như sau:

collection-> select(v | boolean-expression-with-v)

Phép toán thực hiện như sau: gán v với mỗi giá trị trong *collection*, định trị biểu thức và nếu kết quả đúng thì thêm giá trị này vào *collection* kết quả. *Collection* kết quả có cùng kiểu *collection* ban đầu. Ví dụ, chọn ra các phần tử lớn hơn một giá trị cho trước như sau:

a->select(x | x>100)

cho kết quả là một *collection* bao gồm các phần tử có giá trị lớn hơn 100.

Vì vậy, nếu áp dụng phép toán này cho:

bag{12, 86, 342, 3, 12, 567, 432}

cho kết quả là:

bag{342, 567, 432}

12.3.7 Phép toán *Reject*

Phép toán này nhặt ra các phần tử từ một *collection* cho trước không thoả một điều kiện nào đó. Cú pháp như sau:

collection-> reject(v | boolean-expression-with-v)

Phép toán thực hiện như sau: gán v vào mỗi giá trị trong *collection*, định trị biểu thức, nếu kết quả sai thì thêm giá trị này vào *collection* kết quả. *Collection* kết quả có cùng kiểu với *collection* ban đầu. Ví dụ

a->reject(x | x>100)

cho kết quả là một *collection* gồm các phần tử có giá trị không lớn hơn 100.

Vì vậy, nếu áp dụng phép toán này cho:

bag{12, 86, 342, 3, 12, 564, 987}

cho kết quả là:

bag{12, 86, 3, 12}

12.3.8 Phép toán *Collect*

Phép toán này tạo một *collection* mới bằng cách áp dụng một biểu thức cho mỗi phần tử của *collection* ban đầu. Cú pháp như sau

collection-> select(v | expression-with-v)

Phép toán này thực hiện như sau: gán v vào mỗi giá trị trong *collection*, định trị biểu thức và đặt kết quả vào *collection* trả về. *Collection* trả về có cùng kiểu với *collection* ban đầu. Ví dụ, để tạo một *collection* là bình phương tất cả các số trong một *collection* cho trước, dùng biểu thức sau:

*a->collect(x | x*x)*

12.4 Làm thế nào để đưa ra các ràng buộc

Việc thu thập các ràng buộc được thực hiện trong qui trình phân tích thiết kế. Bắt đầu bằng việc ghi nhận lại các phát biểu dưới dạng ngôn ngữ tự nhiên, sau đó tinh chế lại để có thể biểu diễn chúng trong một ngôn ngữ hình thức chẳng hạn như OCL. Cuối cùng, lập trình viên dùng các ràng buộc để viết các đoạn *code* rõ ràng, dễ đọc, phù hợp với đặc tả của hệ thống. Các ràng buộc được xác định bằng cách:

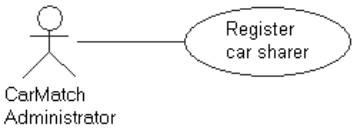
- Xác định *pre-condition* và *post condition* trên các use case;
- Xác định *invariant* trên các đối tượng trong mô hình phân tích;
- Chuyển các ràng buộc trên use case sang các ràng buộc trên thao tác;
- Chuyển các *pre-condition*, *post condition* và *invariant* thành code ở giai đoạn cài đặt.

Việc tinh chế sẽ tiếp tục và lặp lại trong suốt qui trình phân tích thiết kế. Sự mơ hồ của ngôn ngữ tự nhiên làm cho bạn phải thường xuyên đặt ra các câu hỏi xa hơn để làm rõ vấn đề. Cho đến khi việc thiết kế hoàn tất, không thể biểu diễn đầy đủ các ràng buộc bằng OCL, bởi vì lúc này bạn chưa có được đầy đủ các đối tượng và thao tác.

12.4.1 Xác định các ràng buộc trên use case

Các use case chỉ được gọi một cách hợp lệ dưới các điều kiện nào đó. Ghi nhận các điều kiện như vậy là một phần quan trọng trong định nghĩa use case. Có một cách thực hiện là cung cấp một mẫu tài liệu mô tả use case. Mẫu này mô tả các scenario, như đã minh họa trong hình 3.11, và các yêu cầu bổ sung bất kỳ, bao gồm *pre-condition* và *post-condition*.

Xét use case *Register car sharer* trong hình 12.9. Mô tả chi tiết trong hình 12.10. Trong giai đoạn này không thể dùng OCL để đặc tả các *pre-condition* và *post-condition* khi cấu trúc đối tượng cơ sở chưa hoàn tất.



Hình 12.9: Use case Register car sharer

12.4.2 Xác định ràng buộc trên đối tượng

Các use case được thực hiện qua các dòng công việc, kết quả được một mạng các đối tượng giao tiếp với nhau. Các đối tượng này được cấu hình để cung cấp chức năng. Vì vậy, các *pre-condition* và *post-condition* cho các use case cuối cùng sẽ được chuyển thành các ràng buộc trên đối tượng. Trong use case *Register car sharer*, chúng ta có lớp *CarSharer* có thuộc tính ngày sinh *dateOfBirth*, lớp này không lưu trữ thuộc tính tuổi, vì thế cần có một thao tác trên *CarSharer* trả về số tuổi của một thực thể *CarSharer*. Chuyển ràng buộc từ use case thành một ràng buộc *invariant* của lớp *CarSharer* như sau:

context *CarSharer* **inv**:

self.age() >= 21.

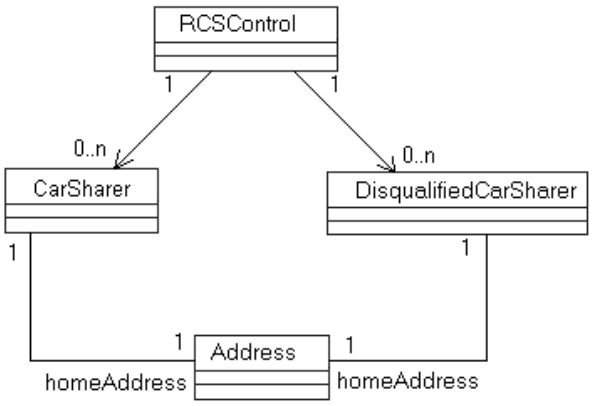
Biểu diễn không chính xác của ngôn ngữ tự nhiên đã chính xác hơn trong OCL.

Use case: *Register car sharer*
Pre-condition: 1. Người chia sẻ xe phải lớn hơn 21 tuổi. 2. Nếu người chia sẻ xe tỏ ý muốn lái, anh ấy phải có bằng lái và bảo hiểm còn hiệu lực. 3. Người chia sẻ xe phải chưa đăng ký trước đó. 4. Người chia sẻ xe không phải là thành viên phạm luật trong quá khứ.
Post-condition: 1. Người chia sẻ xe được đăng ký chi tiết. 2. Người chia sẻ xe đã trả phí thành viên. 3. Gói chào mừng đã được gởi đến người chia sẻ xe 4. Sự đăng ký lộ trình đối với người chia sẻ xe hiệu lực.
Decription: ....

Hình 12.10: Mô tả use case Register car sharer

12.4.3 Chuyển ràng buộc use case sang ràng buộc thao tác

Lớp *CarSharer* là một lớp *entity*, các thể hiện của nó có các thao tác được gọi từ một đối tượng điều khiển của lớp *RCSControl*, cung cấp cài đặt của use case *Register car sharer*. Để ép thoả *invariant* trên *CarSharer*, ta cài đặt một *pre-condition* trên *RCSControl* khởi tạo các *CarSharer*.



Hình 12.11: Một phần biểu đồ lớp cài đặt use case Register car sharer

Ràng buộc các thành viên phải đủ tư cách trong quá khứ hàm ý giữ một đăng ký củ các thành viên *không đủ tư cách* (disqualify). Biểu đồ lớp trong hình 12.11 cho thấy các kết hợp đối tượng hỗ trợ use case này. Bây giờ chúng ta viết một tập các *pre-condition* cho thao tác *register*() trên lớp *RCSControl* như sau:

context *RCSControl*::*register*(*name*, *address*)

seft.CarSharer->forall(*seft.name<>name* and *seft.address<>address*) and

seft.DisqualifiedCarSharer->forally(*seft.name<>name*

and *seft.address<>address*)

12.4.4 Chuyển các ràng buộc thành code

Các *pre-condition* và *invariant* tự chúng không thể thi hành được. Tuy nhiên, vì chúng xác định các điều kiện hợp lệ bên dưới hệ thống, cần viết *code* kiểm tra để bảo đảm rằng các ràng buộc này không bị vi phạm. Nếu chúng bị vi phạm, thông thường nó sẽ phát ra một ngoại lệ để hệ thống hoặc người dùng sửa lại hành động cho đúng. Các *pre-condition* trên thao tác có thể được chuyển thành các kiểm tra đơn giản ở đầu các đoạn mã cài đặt hàm. Các *invariant* có thể được kiểm tra bằng cách đặt

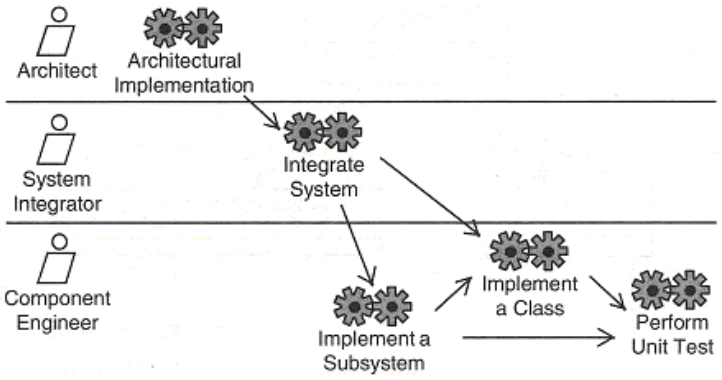
thêm một đoạn mã trước khi thay đổi các thuộc tính liên quan đến *invariant*.

Các *post-condition* là một phần yêu cầu rút gọn của thao tác. Mã nguồn nên thoả các *post-condition* bảo đảm thao tác được thi hành hợp lệ. Các *post-condition* được lập trình viên sử dụng như một phần của đặc tả. Nếu được, hãy kiểm tra chúng khi kết thúc một thủ tục; nhất là khi đoạn mã phức tạp hoặc có sự cấp bách về tính an toàn.

12.5 Quan hệ với các biểu đồ khác

Các biểu thức OCL được lưu trữ riêng biệt hoặc được gắn trên các biểu đồ theo nhiều cách. Các *invariant* có thể được đặt trong một ghi chú và gắn với đối tượng liên quan. Các điều kiện bảo vệ được viết bằng OCL và được gắn trên các đường chuyển tiếp trong biểu đồ hoạt động và biểu đồ trạng thái. Không có nơi nào trên các biểu đồ dành cho việc ghi nhận các *pre-condition* và *post-condition* của thao tác; việc đặt chúng trong các ghi chú sẽ làm rối biểu đồ, vì thế cách hiệu quả nhất là lưu trữ riêng. Các công cụ CASE hỗ trợ nhiều phương tiện khác nhau để kết hợp các chuỗi văn bản với các phần tử trong mô hình UML và những phương tiện này có thể được dùng để ghi lại các biểu thức OCL.

12.6 OCL trong UP



Hình 12.12: Implementation workflow

OCL không được tham khảo một cách đặc biệt trong UP. Vai trò chủ yếu của nó là cung cấp việc lưu lại chính xác các ràng buộc. Vì các biểu thức OCL được xây dựng bên ngoài các phần tử trong mô hình, mô hình cần được mô tả chi tiết trước khi biểu diễn các ràng buộc. OCL được dùng để mô tả các ràng buộc trong mô hình nghiệp vụ một khi các đối tượng nghiệp vụ đã được định nghĩa. Các yêu cầu thường được ghi nhận bằng

ngôn ngữ tự nhiên sẽ được chuyển giao một khi các mô hình đối tượng đã được xác định thông qua qui trình phân tích thiết kế. Việc sát hạch (test) các đơn vị có thể dùng các điều kiện *pre-condition* và *post-condition* để phát triển các kế hoạch sát hạch. Trong *Implementation Workflow* (dòng cài đặt) các hoạt động *Implement a Class* (cài đặt một lớp) và *Perform Unit Test* (thực hiện sát hạch đơn vị), trước hết các ràng buộc được dùng để xác định các điều kiện trong mã chương trình của mỗi lớp, thứ hai cung cấp các sát hạch áp dụng cho các thể hiện của mỗi lớp. Hình 12.12 biểu diễn dòng công việc này.

Câu hỏi ôn tập

12.1 Ràng buộc là gì?

12.2 Tại sao thêm các ràng buộc vào mô hình?

12.3 OCL là gì?

12.4 Việc dùng ngôn ngữ tự nhiên để định nghĩa ràng buộc có vấn đề gì?

12.5 Pre-condition là gì?

12.6 Post-condition là gì?

12.7 Invariant là gì?

12.8 Thiết kế gọn là gì?

12.9 Ngữ cảnh của một ràng buộc là gì?

12.10 Các kiểu cơ bản trong một biểu thức OCL là gì?

12.11 Ngoài các kiểu cơ bản, còn có các kiểu nào được dùng trong biểu thức OCL?

12.12 Định trị biểu thức: *if* ((3+4)>=7) *then* 1 *else* 2

12.13 Có mấy kiểu tập hợp (collection)?

12.14 Làm cách nào để tìm ra phần tử đầu tiên trong một *sequence* (dãy)?

12.15 Triển khai tập hợp *set* { *set* {2, 4, 7}, *set* {3, 8, 9}}

12.16 Định trị biểu thức *bag* {2, 5, 2, 3, 8, 3, 4, 6} -> *select*(x | x<5).

12.17 Định trị biểu thức *bag* {2, 8, 4, 6, 0, 4, 8, 2}->*forall*(x | x.*mod*(2) = 0).

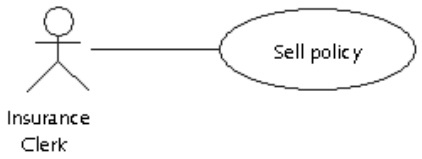
12.18 Bạn có thể biểu diễn các điều kiện *pre-condition* và *post-condition* của use case bằng OCL không?

Bài tập có lời giải

12.1 Hãy xem xét use case bán các hợp đồng bảo hiểm, hình 12.13. Như một phần của các thảo luận với các công ty bảo hiểm, các yêu cầu sau đây được thoả thuận:

- Thành viên phải không còn nợ trước khi nhận được hợp đồng.
- Lần thanh toán đầu tiên của lịch thanh toán phải được thực hiện trước khi hợp đồng được công nhận.
- Với một thành viên, hai hợp đồng không được phép cùng hiệu lực trong cùng thời điểm.

Hãy viết các ràng buộc OCL để quản lý các yêu cầu này.



Hình 12.13: Use case Sell policy

Lời giải:

Các yêu cầu này được thêm vào use case như là các *pre-condition* và được biểu diễn như trong hình 12.10 ở trên. Sau đó, như một phần của qui trình thiết kế, một phần của biểu đồ lớp được biểu diễn như trong hình 12.14. Trong hình có một đối tượng điều khiển, *SPControl*, được dùng để quản lý các giao tác của use case, và có ba thao tác thu thập các chi tiết bảo hiểm, tạo mới hợp đồng và để tạo mới lịch thanh toán. Nên kiểm tra *pre-condition* thứ nhất trước khi thu thập các chi tiết bảo hiểm - sẽ không thích hợp nếu như thu thập thông tin trong khi có một cái gì đó, được kiểm tra một cách dễ dàng, có thể sẽ khoá giao tác. Vì thế ta đặt một *pre-condition* trên thao tác *gatherInsuranceDetails* như sau:

context *SPControl*::*gatherInsuranceDetails*()

pre: *CarSharer.membership.balance* >= 0.

Như vậy ta đã dùng *CarSharer* được liên kết với *SPControl*, và dò theo vai trò thành viên để đến tài khoản giữ số dư trên phí thành viên, và kiểm tra điều kiện.

Yêu cầu thứ hai được ánh xạ vào thao tác *createPaymentSchedule*. Do ngày tháng không phải là kiểu cơ bản, nên một thao tác phải có sẵn trên kiểu ngày để kiểm tra nó có trước một ngày khác không, dùng ngày khác làm đối số và trả về một giá trị kiểu boolean. Ràng buộc như sau:

context *SPControl*::*createPaymentSchedule*(*startDate*,*regularDate*,*amount*, *frequency*)

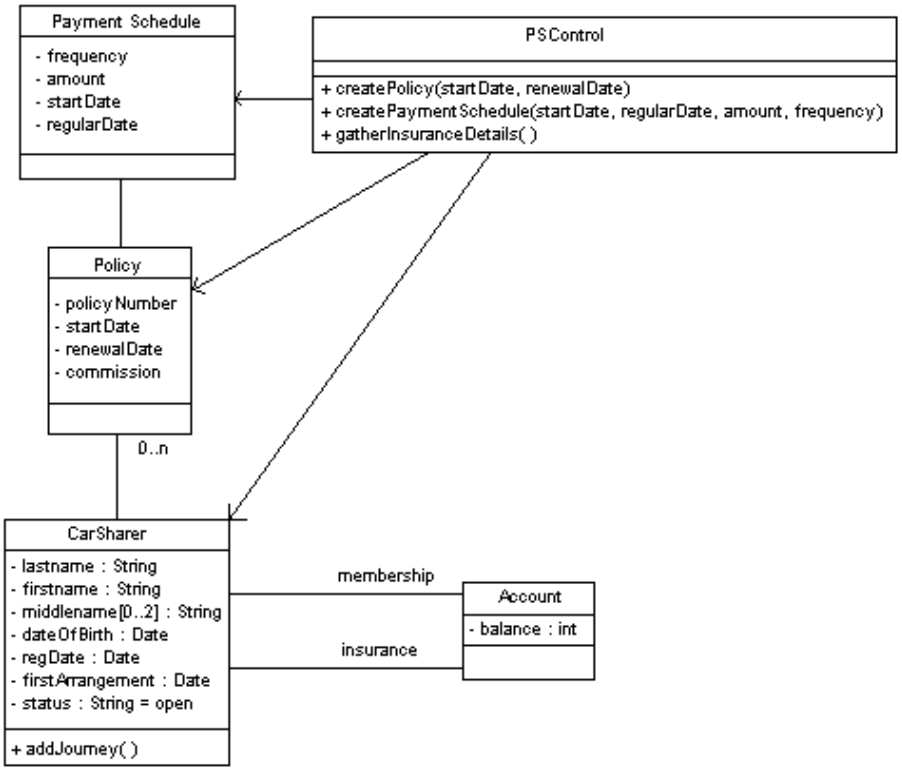
pre: *startDate.before*(*Policy.startDate*)

Yêu cầu thứ ba là một *invariant* trên *CarSharer*, *invariant* này có thể được biểu diễn như sau "*ngày gia hạn của một hợp đồng phải diễn ra trước ngày bắt đầu của bất kỳ một hợp đồng nào trễ hơn nó*, và ràng buộc bằng OCL như sau:

context *CarSharer*

inv: *Policy->forall*(*p1*, *p2* | *p1* <> *p2* and
p1.startDate.before(*p2.startDate*)

implies p1.*renewalDate.before*(p2.*startDate*)).



Hình 12.14: Một phần biểu đồ lớp cài đặt use case Sell policy

Xét kỹ điều này, chúng ta thấy cần đặt một ràng buộc trên *Policy* để bảo đảm ngày gia hạn phải xảy ra sau ngày bắt đầu.

context *Policy*

inv: *startDate.before*(*renewDate*).

Xem xét xa hơn đến thiết kế của hệ thống, *invariant* trên *CarSharer* được bảo đảm bằng cách đặt một *pre-condition* trong thao tác *createPolicy* như sau:

context *SPControl*::*createPolicy*(*startDate*, *renewalDate*)

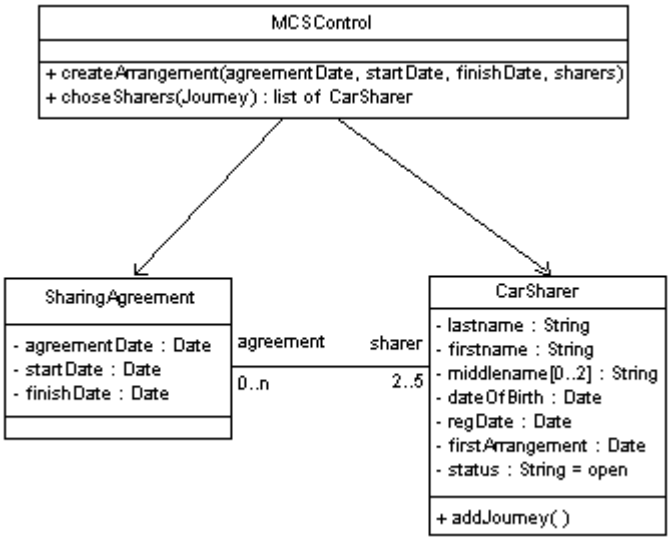
pre: *CarSharer.Policy -> forall*(*p* | *p.startDate.before*(*self.startDate*)

implies p.renewalDate.before(*self.startDate*)).

Quá trình chuyển các *pre-condition* này thành mã nguồn tuỳ thuộc rất nhiều vào ngôn ngữ lập trình mà chúng ta chọn. Tuy nhiên, trong từng trường hợp thao tác sẽ bắt đầu với một số kiểm tra đơn giản để bảo đảm rằng các *pre-conditon* là đúng. Ngược lại thao tác sẽ kết thúc kèm giá trị trả về hoặc sẽ phát sinh ra một ngoại lệ.

12.2 Xét use case *Match car sharers*. Các yêu cầu cho việc chia sẻ bao gồm điều kiện tạo hợp đồng, ít nhất phải có một tài xế và có ít nhất là hai người trong hợp đồng. Hãy viết các ràng buộc OCL để quản lý các yêu cầu này.

Lời giải:



Hình 12.15: Một phần biểu đồ lớp cài đặt use case Match car sharers

Đây có thể là một *post-condition* của use case *Match car sharers*. Chúng ta sẽ thiết kế một lớp điều khiển *MCSControl* với một thao tác chọn người chia sẻ một lộ trình rồi tạo ra một hợp đồng chia sẻ. Hình 12.15 là mô hình đối tượng. Yêu cầu của bài tập có nghĩa là thao tác *chooseSharers* phải trả về danh sách những người dùng, trong số đó có một là tài xế. Rõ ràng chúng ta cần một phương tiện để kiểm tra nếu như một người nào đó có thể lái xe được, và chúng ta sẽ đặt một thuộc tính *canDrive* vào trong đối tượng. Yêu cầu có thể được biểu diễn bằng một *post-condition* như sau:

context *MCSControl*::*chooseSahrers*(*journey*): *list of CarSharer*

post: *self->exists* (*x* | *x.canDrive*) and (*self->size*) >= 2.

Điều kiện này rồi trở thành *pre-condition* trên thao tác *createArrangement*:

context *MCSControl*::*createArrangement*(*agreementDate*, *startDate*, *finishDate*, *sharers*)

pre: *sharers->exists* (*x* | *x.canDrive*) and (*sharers->size*) >= 2.

Sự gặp nhau giữa các điều kiện *pre-condition* và *post-condition* này sẽ là phương tiện trong việc cài đặt *invariant* trên hợp đồng chia sẻ như sau:

context *SharingAgreement*

inv: *CarSharer->exists* (*x* | *x.canDrive*) and (*CarSharers->size*) >= 2.

Bài tập bổ sung

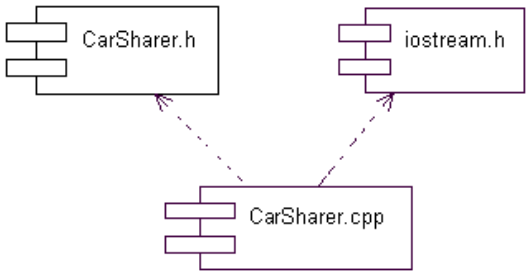
12.3 Trong *VolBank*, một yêu cầu đặt ra là một người không được phép đăng ký quá 200 giờ, và cũng không được rút lại hơn 100 giờ. Hãy dùng OCL để biểu diễn ràng buộc này.

12.4 Trong *VolBank*, yêu cầu một tình nguyện viên không thể làm việc cho ba tổ chức tại một thời điểm. Đối với use case kết hợp tình nguyện viên với tổ chức, hãy biểu diễn điều này bằng một *pre-condition* trên use case này. Hãy tìm ra một tập các đối tượng để cài đặt use case này, và biểu diễn ràng buộc này bằng OCL theo cả hai cách một dùng *invariant* và một dùng *pre-condition* trên thao tác cài đặt việc kết hợp.

Chương 13
BIỂU ĐỒ CÀI ĐẶT

13.1 Giới thiệu

Có hai loại biểu đồ trong UML được dùng để mô hình việc cài đặt hệ thống máy tính là *biểu đồ thành phần* (component diagram) và *biểu đồ triển khai* (deployment diagram). Biểu đồ thành phần được dùng để mô hình mối quan hệ của các thành phần *phần mềm* trong hệ thống. Thông thường chúng là các quan hệ giữa các tập tin chương trình nguồn, giữa các phần mềm đang chạy hoặc giữa tập tin nguồn với tập tin thi hành tương ứng. Tuy nhiên, chúng cũng có thể được dùng để sưu liệu cho bất kỳ thành phần phần mềm tạo nên hệ thống máy tính và mối quan hệ giữa chúng. Biểu đồ triển khai được dùng để mô hình các phần cứng được dùng để thi hành hệ thống và liên kết các thành phần này với nhau. Các thành phần được biểu diễn trong biểu đồ triển khai theo cho phép mô hình việc khai triển các thành phần thực thi trên các bộ xử lý trong một hệ thống phẳng. Hình 13.1 là một ví dụ về biểu đồ thành phần mô tả mối quan hệ giữa một tập tin nguồn C++ và hai tập tin *header*.

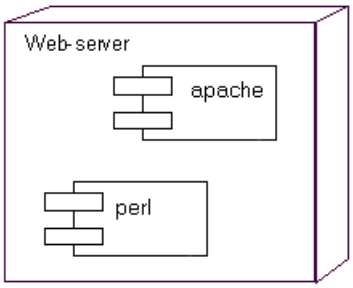


Hình 13.1: Biểu đồ thành phần biểu diễn các phụ thuộc của các tập tin nguồn C++

Hình 13.2: Biểu đồ triển khai biểu diễn sự truyền thông giữa PC và Web server

Hình 13.2 là ví dụ về biểu đồ triển khai mô tả phần cứng của một hệ thống và cách liên kết các bộ xử lý khác nhau.

Các thành phần được dùng trong biểu đồ triển khai cho biết vị trí của các thành phần thực thi của hệ thống sẽ được triển khai ở đâu, như trong hình 13.3.



Hình 13.3: Một biểu đồ triển khai hiển thị các thành phần

13.2 Mục đích của kỹ thuật

Cả hai loại biểu đồ trên đều mô hình khía cạnh cài đặt vật lý của hệ thống.

Biểu đồ thành phần được dùng để mô hình một khung nhìn tĩnh về các thành phần tạo nên hệ thống. Các thành phần có thể là *tập tin*, *chương trình thi hành*, *tài liệu*, *thư viện* (lib) hoặc *bảng dữ liệu*. Chúng được liên kết với nhau trong biểu đồ bằng các mối quan hệ phụ thuộc, tổng quát hóa, hiện thực hoá và các kết hợp khác. Mục đích chính của biểu đồ thành phần là:

- Mô hình vật lý các thành phần phần mềm và các mối quan hệ giữa chúng.
- Mô hình các tập tin mã nguồn và mối quan hệ giữa chúng.
- Mô hình cấu trúc của phiên bản phần mềm.
- Xác định các tập tin được biên dịch thành tập tin thi hành.

Tuy nhiên, chúng có thể được dùng để mô hình các đối tượng phần mềm bất kỳ khác tạo nên một hệ thống máy tính, ví dụ như quan hệ giữa các tập tin giúp đỡ với các tập tin chương trình dùng các tập tin này, hoặc cấu trúc của các bảng trong một cơ sở dữ liệu.

Biểu đồ triển khai được dùng để mô hình cấu hình của các thành phần phần cứng tạo nên hệ thống. Bao gồm các máy tính (*client* và *server*), các bộ xử lý và các thiết bị ngoại vi. Chúng cũng được dùng để biểu diễn các *nút* nơi cư trú của các thành phần phần mềm của hệ thống thực thi. Biểu đồ triển khai được dùng để:

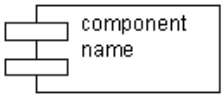
- Mô hình vật lý các thành phần phần cứng và các kênh liên lạc giữa chúng.
- Lập kế hoạch kiến trúc của một hệ thống.
- Lập tài liệu việc triển khai các thành phần phần mềm trên các *nút* phần cứng.

Các thành phần trong biểu đồ thành phần và các nút trong biểu đồ triển khai có thể được tạo thành khuôn dạng bằng cách dùng các biểu tượng để biểu diễn loại thành phần đặc biệt hoặc loại nút đặc biệt.

13.3 Ký hiệu

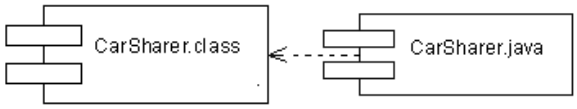
13.3.1 Biểu đồ thành phần

Gồm các thành phần và các quan hệ phụ thuộc giữa chúng. Ký hiệu thành phần như sau:

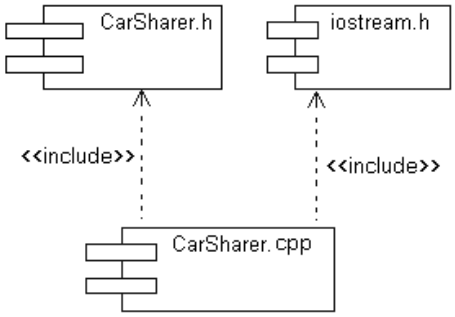


Hình 13.4: Ký hiệu một thành phần

Tên thành phần là tên của tập tin vật lý hoặc tên hệ thống con được thiết kế để trở thành một thành phần. Quan hệ giữa các thành phần là quan hệ phụ thuộc. Hình 13.5 mô tả một phụ thuộc giữa tập tin *.java* và tập tin *.class* của nó.



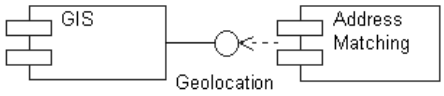
Hình 13.5: Phụ thuộc giữa các thành phần



Hình 13.6: Các loại phụ thuộc giữa các thành phần

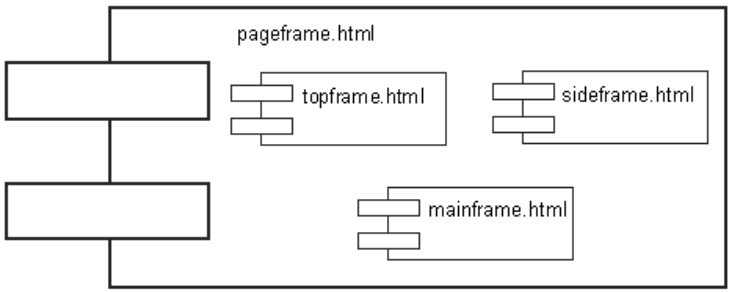
Cạnh mũi tên có thể có một nhãn cho biết bản chất của phụ thuộc, xem hình 13.6. Trong UML, có một số loại phụ thuộc chuẩn như: <<*include*>>, <<*derive*>>, <<*friend*>> và <<*import*>>. Các loại khác có thể được định nghĩa để thoả mãn nhu cầu của một dự án hoặc tổ chức.

Một thành phần có thể cài đặt một giao diện. Giao diện được biểu diễn hoặc như một vòng tròn nối với thành phần cài đặt bằng một đường thẳng hoặc như một lớp giao diện. Các thành phần sử dụng giao diện được biểu diễn với một mũi tên phụ thuộc đi đến giao diện hơn là đến thành phần cài đặt giao diện. Hình 13.7 minh họa thành phần GIS (*geographical information system*) cài đặt giao diện *Geolocation* được dùng bởi thành phần *Address Matching*.

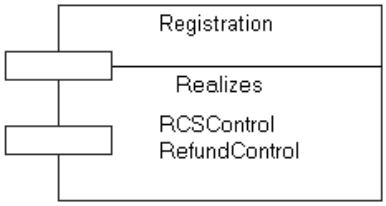


Hình 13.7: Biểu đồ thành phần mô tả một phụ thuộc vào giao diện

Việc chứa một hoặc nhiều thành phần bên trong một thành phần khác có thể được biểu diễn bằng cách đặt các thành phần được chứa bên trong thành phần chứa như trong hình 13.8. Hình này biểu diễn một mối quan hệ hợp thành (composition) giữa thành phần chứa và các thành phần mà nó chứa.



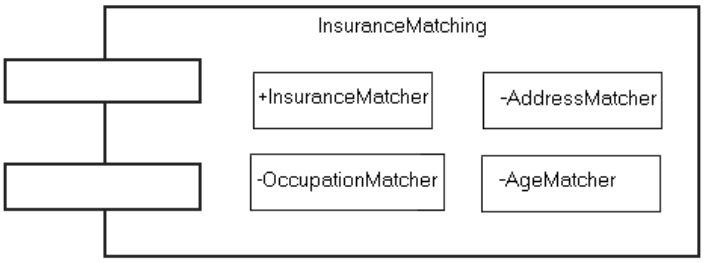
Hình 13.8: Một biểu đồ mô tả các thành phần được chứa



Hình 13.9: Thành phần với một ngăn mô tả các lớp được hiện thực

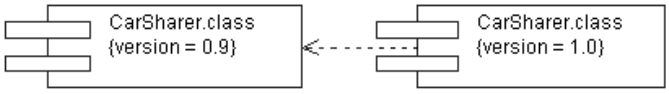
Theo đặc tả UML, thành phần là một loại *classifier* (xem chương 14), nó có thao tác và thuộc tính. Các thông tin khác về thành phần, chẳng hạn như các lớp mà nó hiện thực có thể được mô tả trong các ngăn bên trong biểu tượng thành phần, xem hình 13.9

Các lớp được một thành phần cài đặt cũng có thể được mô tả bên trong. Tên lớp có thể kèm dấu cộng hoặc trừ đặc tả tính khả kiến. Hình 13.10 cho một ví dụ về điều này với lớp mẫu *Facade* trong hình 8.12. Thành phần *InsuranceMatching* cài đặt 4 lớp nhưng chỉ có một lớp đóng vai trò là lớp *mặt tiền* (public) của mẫu này; ba lớp còn lại, đóng vai trò các lớp hệ thống con, cung cấp các dịch vụ xác định nhưng chỉ có thể được truy cập thông qua lớp mặt tiền.



Hình 13.10: Một thành phần hiển thị các lớp được chứa

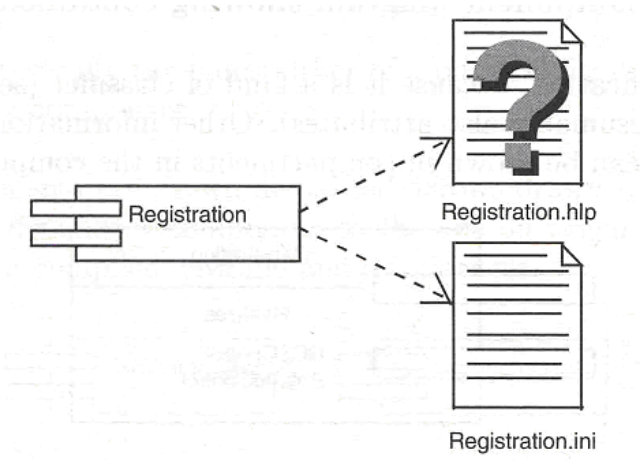
Các ràng buộc cũng được thêm vào các thành phần và được dùng để hiển thị các thông tin chẳng hạn như số phiên bản, hình 13.11. Một ràng buộc có thể được biểu diễn bên trong hoặc bên cạnh thành phần trên biểu đồ.



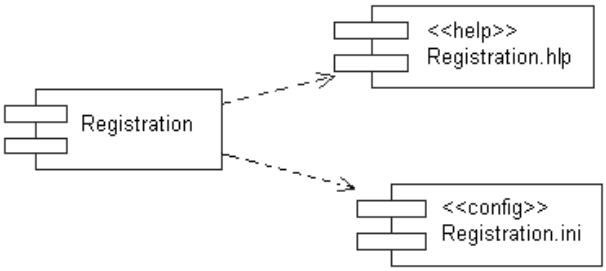
Hình 13.11: Các phiên bản khác nhau
của một thành phần được biểu diễn bằng cách dùng ràng buộc

Biểu tượng thành phần gồm một hình chữ nhật lớn với hai hình chữ nhật nhỏ trên cạnh trái. Đây là biểu tượng chuẩn. Nếu bạn muốn nhấn mạnh một thành phần không phải là một chương trình, có thể dùng các biểu tượng khuôn dạng thay cho biểu tượng chuẩn. Hình 13.12 cho thấy các biểu tượng khuôn dạng cho tập tin cấu hình *Registration.ini* và tập tin giúp đỡ *Registration.hlp* cho thành phần *Registration*. Các khuôn dạng cũng có thể được đặt bên trong thành phần như trong hình 13.3

Các thành phần trong biểu đồ thành phần thường là các kiểu hơn là các thể hiện. Thể hiện của thành phần có thể được biểu diễn trong biểu đồ triển khai.



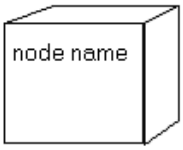
Hình 13.12: Các biểu tượng thành phần với khuôn dạng



Hình 13.13: Các thành phần khuôn dạng

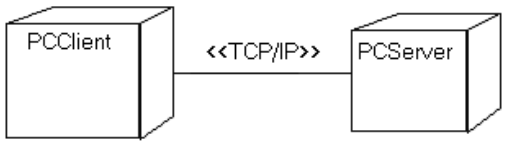
13.3.2 Biểu đồ triển khai

Biểu đồ triển khai biểu diễn các nút (node) được kết nối với nhau bằng các kết hợp *truyền thông* (communication) giữa các nút. Ký hiệu nút là một hình khối như hình 13.14.



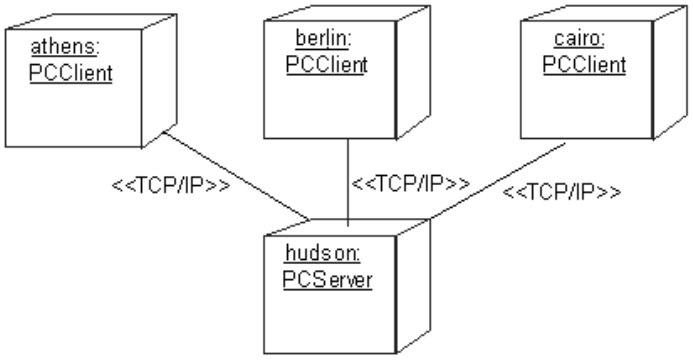
Hình 13.14: Ký hiệu một nút

Các nút đại diện cho các tài nguyên xử lý trong hệ thống, thường là các máy tính có bộ xử lý và bộ nhớ. Tuy nhiên chúng cũng có thể được dùng để biểu diễn các thiết bị ngoại vi, cảm ứng và các hệ thống được nhúng bên trong. Các nút được kết nối với nhau bằng mối kết hợp truyền thông, được biểu diễn bằng các khuôn dạng để cho biết bản chất của truyền thông giữa hai nút. Hình 13.15 biểu diễn một máy khách và một máy chủ kết nối với nhau qua giao thức mạng TCP/IP.



Hình 13.15: Mối truyền thông giữa một máy chủ và một máy khách

Biểu đồ triển khai được dùng để biểu diễn các kiểu nút, như trong hình 13.15, hoặc thể hiện của nút, như trong hình 13.16. Hình 13.16 mô tả các máy chủ và máy khách cụ thể trong hệ thống.

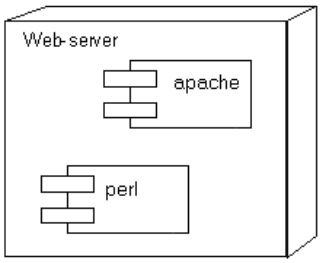


Hình 13.16: Các thể hiện trong biểu đồ triển khai

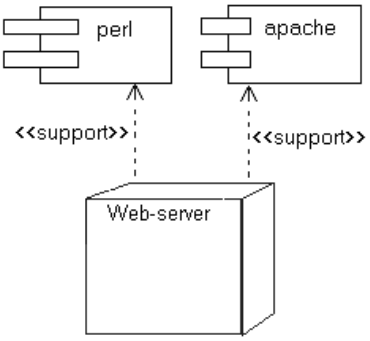
Các thể hiện của thành phần có thể được mô tả bên trong nút. Chỉ có các thành phần thực thi mới được hiển thị. Theo đó các tập tin nguồn không xuất hiện trong biểu đồ còn các tập tin thi hành thì được. Các thành phần có thể được biểu diễn bên trong nút hoặc nối với nút bằng phụ thuộc <<*support*>>, như trong hình 13.17 và 13.18.

Các đối tượng cũng được mô tả bên trong thể hiện của nút và chúng thường là các đối tượng hoạt động (xem chương 8). Các đối tượng cũng có thể được biểu diễn bên trong các thể hiện của thành phần. Một biểu đồ thành phần chỉ biểu diễn các thành phần; nên các thể hiện của thành

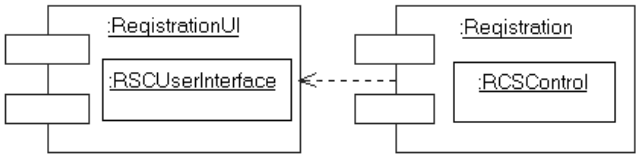
phần sẽ được biểu diễn trong biểu đồ triển khai, và bạn có thể tạo ra một biểu đồ triển khai không có nút nào mà chỉ có các thể hiện của thành phần. Hình 13.19 là một ví dụ về biểu đồ triển khai như thế, trong đó chỉ có các thành phần và các đối tượng liên quan với use case *Register car sharer*.



Hình 13.17: Một biểu đồ triển khai với các thành phần bên trong nút

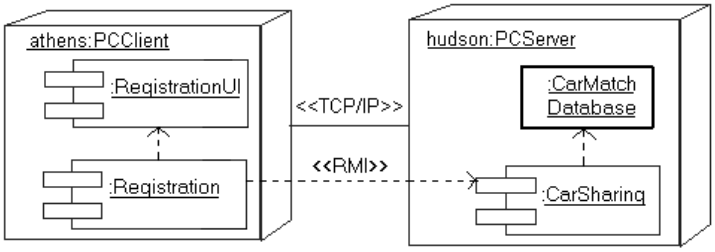


Hình 13.18: Một biểu đồ triển khai với phụ thuộc <<support>> trên các thành phần



Hình 13.19: Biểu đồ triển khai chỉ có các thành phần và đối tượng

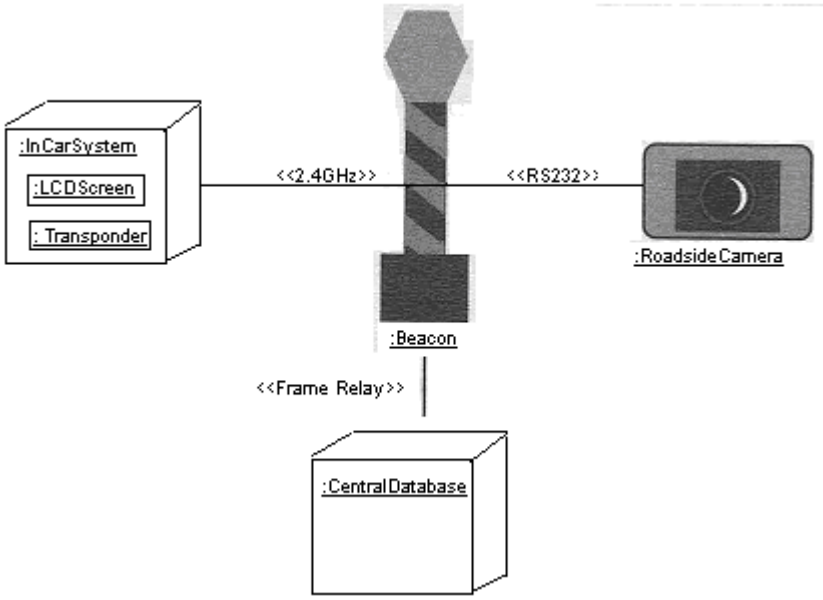
Hình 13.20 minh họa hai thành phần của hình 13.19 trong nút *PCClient* và thành phần *CarSharing*, thành phần hiện thực các lớp *CarSharer*, *Journey* và *Address*, trên nút *PCServer* cùng với đối tượng hoạt động *CarMatchDatabase*.



Hình 13.20: Một biểu đồ triển khai biểu diễn các thành phần và một đối tượng hoạt động

Lưu ý các mối quan hệ truyền thông giữa các nút là TCP/IP trong khi phụ thuộc giữa thành phần *Registration* và *CarSharing* là *RMI* bởi vì nó dùng *Remote Method Invocation* (gọi phương thức từ xa) của Java để gọi các thao tác trên các thể hiện của *CarSharer*, *Journey* và *Address*.

Các nút trong một biểu đồ triển khai có thể được biểu diễn bằng cách dùng các khuôn dạng hoặc các biểu tượng khuôn dạng nếu cần. Đây là cách hữu ích nhất khi có nhiều loại nút khác nhau được biểu diễn trong cùng một biểu đồ. Hình 13.21 là một ví dụ.



Hình 13.21: Một biểu đồ triển khai với các nút khuôn dạng

13.4 Cách tạo biểu đồ cài đặt

13.4.1 Cách tạo biểu đồ thành phần

Biểu đồ thành phần được dùng cho một số mục đích, và những gì tạo nên thành phần phụ thuộc vào môi trường phát triển hệ thống. Ví dụ, thành phần có thể là *package* trong Java hoặc *project* trong Visual Basic. Trong suốt quá trình cài đặt, biểu đồ thành phần được đưa ra để mô tả các thành phần ở mức thấp hơn, như các tập tin *.java* và *.class* của Java hoặc *form* và *module* của Visual Basic. Tuy nhiên, cách tiếp cận thông thường được bắt đầu như sau:

1. Xác định mục đích của biểu đồ.
2. Thêm các thành phần vào biểu đồ, nhóm chúng lại trong các thành phần khác nếu thích hợp.
3. Thêm các phần tử như lớp, đối tượng hoặc giao diện vào biểu đồ.
4. Thêm các phụ thuộc giữa các phần tử của biểu đồ.

13.4.1.1 Xác định mục đích của biểu đồ

Bước đầu tiên là quyết định mục đích của biểu đồ. Nó được dùng để mô hình quan hệ giữa các lớp và các thành phần, giữa các thành phần mã nguồn với nhau, giữa các tập tin mã nguồn và các tập tin thi hành hoặc giữa các tập tin thi hành và các thành phần hỗ trợ? (Trong bài tập chúng ta sẽ mô hình tất cả bốn quan hệ này)

Ví dụ 13.1

Trong ví dụ này, chúng ta sẽ mô hình các thành phần liên quan đến việc đăng ký mới. Mục đích của biểu đồ là biểu diễn mối quan hệ giữa các thành phần có thể thi hành và các thành phần hỗ trợ khác. Hệ thống sẽ được cài đặt là Java, vì thế các thành phần này sẽ là các tập tin *.class* và các tập tin khác chẳng hạn như các tập tin cấu hình và giúp đỡ.

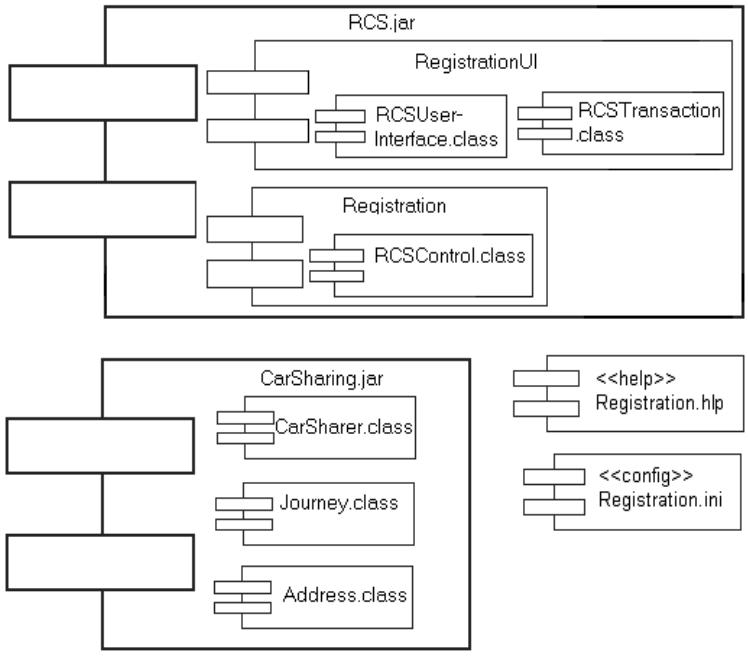
13.4.1.2 Thêm các thành phần vào biểu đồ

Bước đầu tiên tạo biểu đồ là quyết định những thành phần của nó. Chúng là các tập tin được đóng gói cùng nhau để bảo đảm chức năng đăng ký được thực hiện.

Ví dụ 13.2

Java tạo ra một tập tin *.class* cho mỗi lớp trong hệ thống. Chúng được gói lại trong các *package* lưu trong các tập tin *.jar* (tập tin lưu trữ của Java). Cấu trúc lồng nhau của các thành phần phản ánh cấu trúc lồng nhau của các lớp trong *package*. Có hai nhóm lớp, các lớp dùng để xử lý

qui trình đăng ký và các lớp được dùng trong các use case khác. Các lớp liên quan đến việc đăng ký có thể chia thành lớp điều khiển *RCSControl*, hai lớp thu thập dữ liệu *RCSTransaction* và *RCSUserInterface*. Thêm hai tập tin, một để lưu trữ các thông số thiết lập cấu hình và một để giúp đỡ người dùng. Tất cả được biểu diễn như các thành phần, một vài thành phần được lồng trong các thành phần khác. Xem hình 13.22.



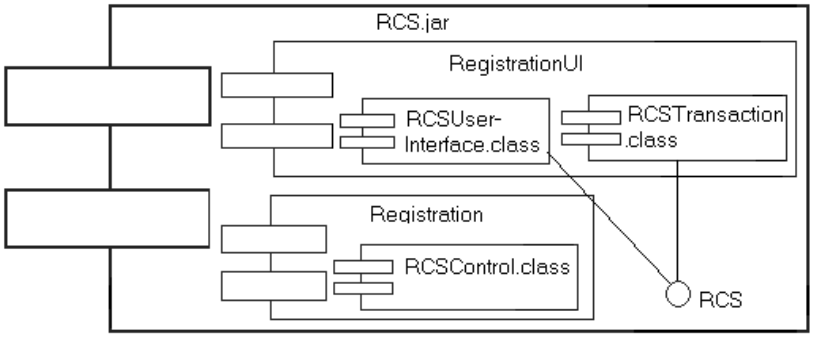
Hình 13.22: Các thành phần cho qui trình đăng ký

13.4.1.3 Thêm các phần tử khác

Bước kế tiếp là thêm các phần tử cần thiết khác vào biểu đồ, đó là các giao diện, lớp và đối tượng.

Ví dụ 13.3

Ví dụ này không có lớp và đối tượng, cả *RCSUserInterface* và *RCSTransaction* đều cài đặt giao diện *RCS*. Giao diện này được dùng bởi lớp *RCSControl* (phụ thuộc này sẽ được thêm vào trong bước kế tiếp). Xem hình 13.23.

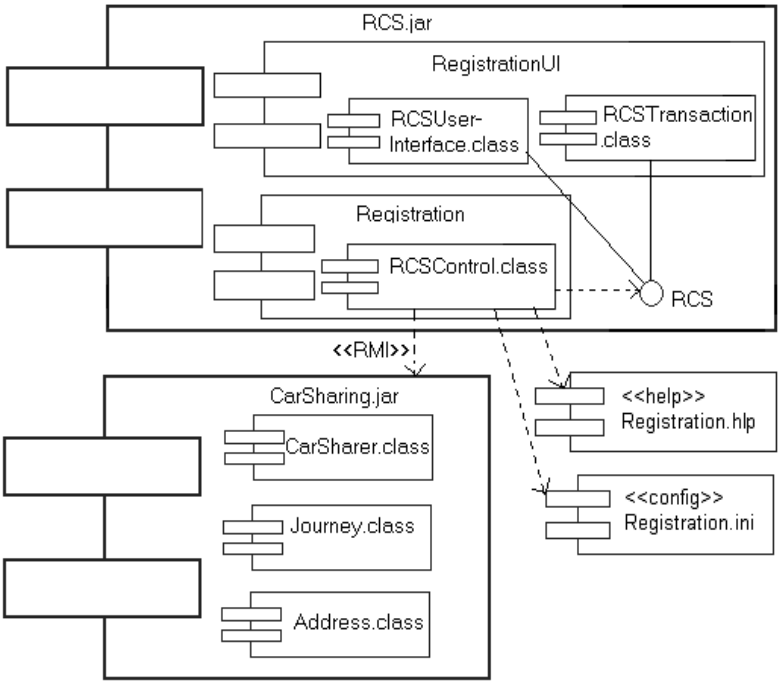


Hình 13.23: Giao diện RCS được thêm vào biểu đồ

13.4.1.4 Thêm các phụ thuộc

Bước cuối cùng là thêm các phụ thuộc giữa các thành phần hoặc giữa các thành phần với phần tử khác trong biểu đồ.

Ví dụ 13.4



Hình 13.24: Các phụ thuộc được thêm vào biểu đồ thành phần

Trong trường hợp này, có các phụ thuộc giữa thành phần *RCSControl* với giao diện *RCS* (để dùng các dịch vụ của các thành phần giao dịch và giao diện người dùng), giữa thành phần *RCSControl* với thành phần *CarSharing* (để dùng các dịch vụ bên trong thành phần *CarSharing*) và giữa thành phần *RCSControl* với các tập tin cấu hình và giúp đỡ. Hình 13.24 biểu diễn các phụ thuộc này trong biểu đồ. Ở đây ta chỉ mô tả khuôn dạng <<RMI>> cho phụ thuộc giữa *RCSControl.class* với các lớp của thành phần *CarSharing*.

13.4.2 Cách tạo biểu đồ triển khai

Biểu đồ triển khai có thể được dùng cho một số mục đích. Cách xây dựng biểu đồ này như sau:

1. Quyết định mục đích của biểu đồ.
2. Thêm các nút vào biểu đồ.
3. Thêm các kết hợp truyền thông vào biểu đồ.
4. Thêm các phần tử khác vào biểu đồ, chẳng hạn như các thành phần hoặc các đối tượng hoạt động, nếu cần.
5. Thêm các phụ thuộc giữa các thành phần và đối tượng, nếu cần.

13.4.2.1 Quyết định mục đích của biểu đồ

Bước đầu tiên là quyết định mục đích của biểu đồ. Nó sẽ được dùng để mô hình các nút hoặc mô hình việc triển khai các thành phần và các đối tượng khác trên nút.

Ví dụ 13.5

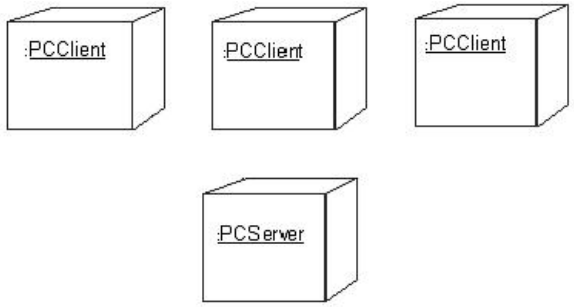
Trong ví dụ này, chúng ta sẽ mô hình việc triển khai các thành phần liên quan đến việc đăng ký mới. Mục đích của biểu đồ là biểu diễn vị trí của các thành phần, đây là một biểu đồ thể hiện.

13.4.2.2 Thêm các nút

Các nút thường là các bộ xử lý mà tập tin thi hành sẽ chạy trên đó.

Ví dụ 13.6

Có hai loại nút trong hệ thống này là các máy khách và máy chủ. Một văn phòng thông thường sẽ có 3 máy khách và một máy chủ. Chúng ta chưa biết tên của các máy này, vì vậy chúng sẽ là các thể hiện vô danh. Xem hình 13.25.



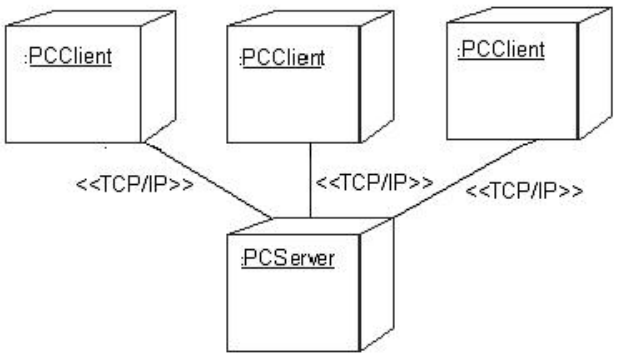
Hình 13.25: Các nút trong biểu đồ triển khai

13.4.2.3 Thêm các kết hợp truyền thông

Kênh truyền thông giữa các cặp nút cần phải được thêm vào biểu đồ. Các khuôn dạng có thể được dùng để biểu diễn giao thức liên lạc.

Ví dụ 13.7

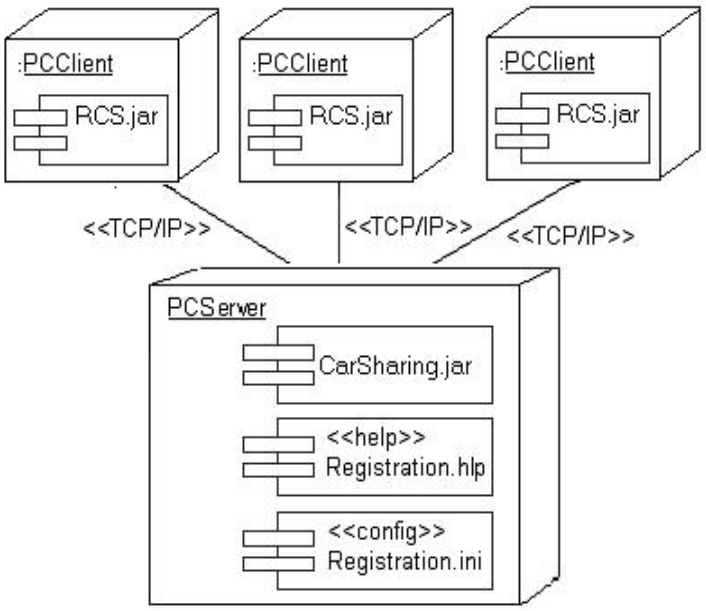
Các máy khách liên lạc với máy chủ bằng giao thức TCP/IP. Chúng ta có thể biểu diễn một mạng như một nút để minh hoạ việc các máy khách có thể liên lạc với nhau. Tuy nhiên, trong biểu đồ này chúng ta chỉ quan tâm đến sự liên lạc giữa các máy khách và máy chủ. Biểu đồ có dạng như trong hình 13.36



Hình 13.26: Các mối liên lạc trong biểu đồ triển khai

13.4.2.4 Thêm các phần tử khác

Các thành phần và đối tượng hoạt động có thể được thêm vào nếu mục đích của biểu đồ là biểu diễn vị trí của chúng trong hệ thống.



Hình 13.27: Các thành phần được thêm vào biểu đồ triển khai

Ví dụ 13.8

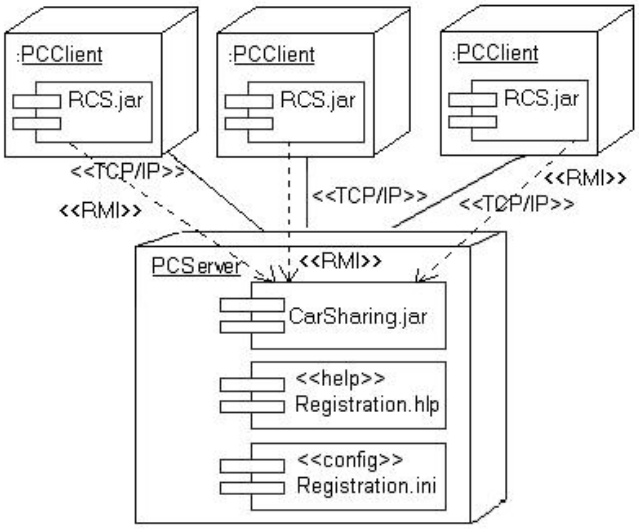
Trong ví dụ này, thành phần *RCS.jar* sẽ nằm trên các máy khách nhưng các thành phần cài đặt các lớp *thực thể* chỉ nằm trên máy chủ. Tập tin giúp đỡ nằm trên máy chủ. Tuỳ thuộc vào mục đích của mình (cấu hình cho toàn hệ thống hay cấu hình cho từng máy riêng), các tập tin cấu hình có thể nằm ở máy khách hoặc máy chủ. Trong trường hợp này, chúng ta quyết định tập tin cấu hình sẽ cấu hình toàn hệ thống, vì vậy nó sẽ nằm trên máy chủ. Kết quả như trong hình 13.27.

13.4.2.5 Thêm các phụ thuộc

Các đường phụ thuộc giúp biểu diễn các phụ thuộc giữa các thành phần với nhau hoặc giữa thành phần với đối tượng trong biểu đồ. Tuy nhiên, biểu đồ có thể trở nên lộn xộn với quá nhiều đường nối. Vì vậy tốt hơn hết là bạn tạo ra nhiều biểu đồ triển khai, mỗi biểu đồ minh hoạ một khía cạnh trong kiến trúc cài đặt.

Ví dụ 13.9

Trong trường hợp này, chúng ta chọn phụ thuộc <<*RMI*>> giữa mỗi thành phần *RCS.jar* và thành phần *CarSharing.jar* vì RMI được dùng để gọi các thao tác của lớp *CarSharing*. Xem hình 13.28

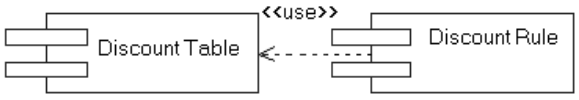


Hình 13.28: Các phụ thuộc được thêm vào biểu đồ triển khai

13.5 Các biểu đồ cài đặt đối với mô hình nghiệp vụ

Cả hai loại biểu đồ cài đặt thường được dùng để mô hình các thao tác nghiệp vụ hơn là các hệ thống phần mềm.

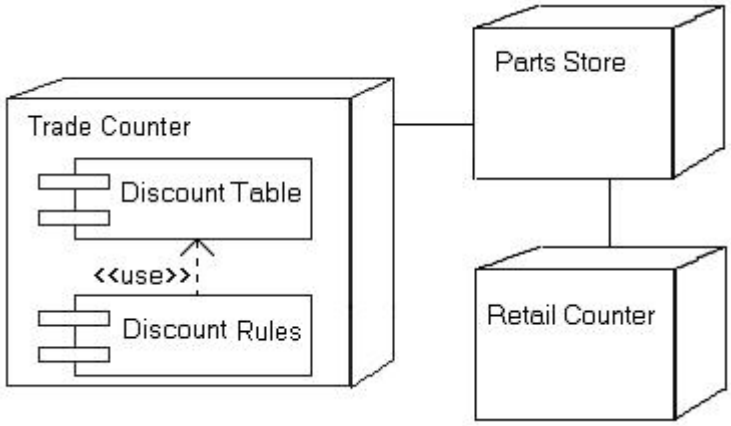
Các thành phần trong biểu đồ thành phần có thể được dùng để mô hình các thủ tục nghiệp vụ và các tài liệu. Ví dụ: nghiệp vụ bán phụ tùng xe sẽ xử lý khách hàng mua lẻ và khách hàng thương mại (các tổ chức nghiệp vụ khác) theo các cách khác nhau. Tại các quầy trong cửa hàng, có một bảng qui định giảm giá cho một danh sách các khách hàng thương mại và tỉ lệ giảm giá của mỗi người. Mặc dù là những tài liệu trên giấy nhưng chúng cũng có thể được mô hình hóa thành các thành phần, như trong hình13.29.



Hình 13.29: Các thành phần nghiệp vụ không phải là các thành phần phần mềm

Biểu đồ triển khai được dùng để mô hình hoá các cấu trúc thi hành không dùng máy tính của nghiệp vụ. Các nút có thể là các đơn vị tổ chức và tài nguyên khác, kể cả người. Trong ví dụ bán xe trên, các nút có thể

được dùng để biểu diễn quầy thương mại (*Trade Counter*), quầy bán lẻ (*Retail Counter*) và kho phụ tùng (*Parts Store*). Xem hình 13.30, các thành phần trong hình 13.29 nằm trong nút *Trade Counter*.



Hình 13.30: Dùng biểu đồ triển khai để mô hình hoá nghiệp vụ

13.6 Quan hệ với các biểu đồ khác

Có mối liên hệ chặt chẽ giữa biểu đồ thành phần và biểu đồ triển khai khi các thể hiện của thành phần được vẽ trong biểu đồ triển khai.

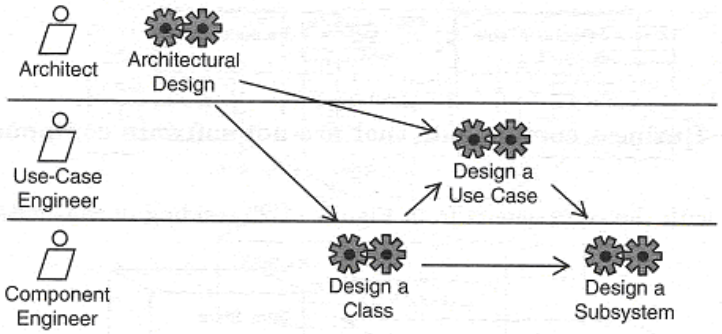
Các thành phần cung cấp một cơ chế cài đặt các lớp và thường được dùng để biểu diễn phụ thuộc <<*trace*>> giữa thành phần và lớp. Các thể hiện của thành phần có thể bao gồm các đối tượng hoạt động, được vẽ bên trong biểu tượng thành phần.

Trong qui trình thiết kế thành phần cho một hệ thống, biểu đồ use case và cộng tác có thể được dùng để xác định lớp nào thuộc về thành phần nào. Những lớp luôn được dùng cùng nhau trong một số ít các use case có quan hệ với nhau thường được đóng gói thành một hoặc một vài thành phần; những lớp được dùng trong nhiều use case khác nhau sẽ được đóng gói thành các thành phần có thể tái sử dụng trong hệ thống mà không phải gắn vào một hoặc một nhóm use case nhất định nào.

13.7 Biểu đồ cài đặt trong UP

Biểu đồ triển khai được dùng trong *Design Workflow* (dòng thiết kế, xem hình 13.31) trong hoạt động *Architectural Design* (thiết kế kiến trúc). Một trong các kết quả của hoạt động này là *Outline Deployment Model* (mô hình triển khai tổng quan). Bước đầu tiên trong hoạt động là *Identifying Nodes and Network Configurations* (xác định các nút và cấu hình mạng), trong đó nhà kiến trúc sẽ tạo ra một mô hình kiến trúc bậc

cao của các nút trong hệ thống và các kênh liên lạc mà các nút này dùng. Các nhân tố như yêu cầu về hiệu suất thi hành, dung lượng, giao thức mạng và bảo mật dữ liệu sẽ được xem xét và các quyết định được ghi nhận lại trong các tài liệu văn bản hỗ trợ.



Hình 13.31: Design workflow

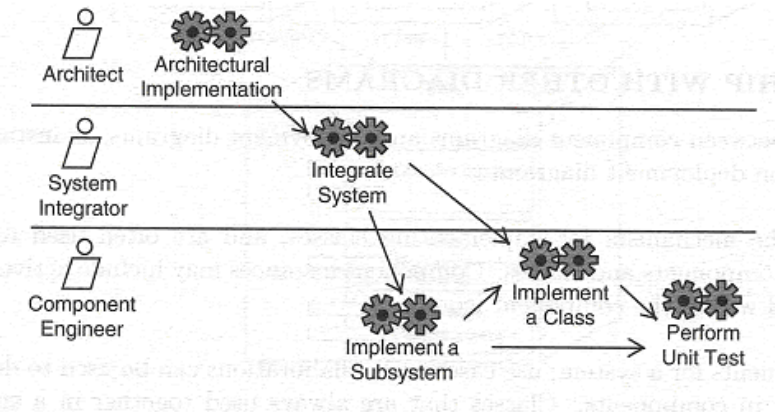
Biểu đồ triển khai cũng được dùng trong bước *Identifying Subsystems and their Interfaces* (xác định các hệ thống con và giao diện của chúng), suốt qui trình này sẽ tạo ra một vị trí ban đầu của các hệ thống con đối với các nút. Trong bước *Identifying Architecturally Significant Design Class* (xác định các lớp thiết kế có ý nghĩa về mặt cấu trúc), ta xác định các lớp hoạt động và định vị chúng trong các nút. Các lớp này sẽ được thể hiện như các đối tượng hoạt động sẽ thi hành trên tiểu trình hoặc tiến trình của riêng chúng trên một nút và phối hợp thao tác với các đối tượng khác.

Cả hai loại biểu đồ thành phần và triển khai đều được dùng trong *Implementation Workflow* (dòng cài đặt) như trong hình 13.32.

Biểu đồ thành phần được dùng trong hầu hết các hoạt động trong dòng này. Trong hoạt động *Architectual Implementation* (cài đặt kiến trúc), chúng được dùng trong bước *Identifying Architecturally Significant Components* (xác định các thành phần có ý nghĩa về mặt cấu trúc), khi các thành phần quan trọng được xác định và quyết định nút chứa chúng. Biểu đồ triển khai cũng được dùng để chỉ ra vị trí của các thành phần trong nút.

Trong hoạt động *Integrate System* (tích hợp hệ thống), các thành phần được dùng trong việc lập kế hoạch xây dựng hệ thống. Hoạt động *Implementation a Subsystem* (cài đặt một hệ thống con) dùng biểu đồ thành phần để mô hình cách dùng thành phần cài đặt các lớp thiết kế trong một hệ thống con. Biểu đồ thành phần còn được dùng trong hành

động *Implementation a Class* (cài đặt một lớp) để mô hình các thành phần tập tin cài đặt lớp (file header, file mã nguồn...). Cuối cùng thành phần là đơn vị trong hoạt động *Perform Unit Test* (kiểm tra). Các thành phần được xem như là các hộp đen để chắc rằng chúng cho các kết quả chính xác với các dữ liệu vào khác nhau, cũng được xem là hộp trắng để chắc rằng nó làm việc như được thiết kế. Các kiểm tra đủ bảo đảm các đoạn mã được thi hành ít nhất một lần dưới vài tình huống.



Hình 13.32: Implementation workflow

Câu hỏi ôn tập

13.1 Thành phần là gì?

13.2 Nút là gì?

13.3 Liên kết truyền thông là gì?

13.4 Ký hiệu cho một thành phần là gì?

13.5 Ký hiệu một nút là gì?

13.6 Mục đích chính của việc dùng biểu đồ thành phần là gì?

13.7 Mục đích chính của việc dùng biểu đồ triển khai là gì?

13.8 Sự khác nhau giữa các thành phần trong biểu đồ thành phần và trong biểu đồ triển khai là gì?

13.9 Cách biểu diễn phụ thuộc trong biểu đồ thành phần?

13.10 Cho hai ví dụ về cách dùng khuôn dạng trong biểu đồ thành phần

13.11 Cho hai ví dụ về cách dùng khuôn dạng trong biểu đồ triển khai.

13.12 Cho một ví dụ về cách dùng ràng buộc trong biểu đồ thành phần.

13.13 Nêu những bước chính để tạo một biểu đồ thành phần.

13.14 Nêu những bước chính để tạo một biểu đồ triển khai.

13.15 Biểu đồ thành phần được dùng trong dòng công việc nào của UP?

13.16 Biểu đồ triển khai được dùng trong hai dòng công việc nào của UP?

Bài tập có lời giải

13.1 Trong phần bài tập này, chúng ta sẽ vẽ bốn biểu đồ thành phần khác nhau. Chúng ta sẽ mô hình các mối quan hệ giữa các lớp và các thành phần, giữa các thành phần mã nguồn, giữa mã nguồn và các tập tin thi hành, giữa các tập tin thi hành và các thành phần hỗ trợ. Chúng ta sẽ đi vào chi tiết trường hợp đầu tiên, sau đó cung cấp giải pháp cho ba trường hợp còn lại.

Chúng ta đã phát triển việc hiện thực use case *Match car sharers* trong chương 8. Use case này được cài đặt trong Java.

Mục đích của biểu đồ thành phần đầu tiên là gì?

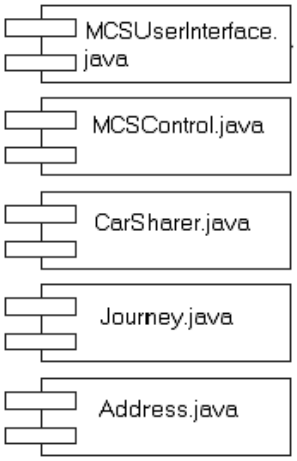
Lời giải:

Biểu đồ đầu tiên mô hình hóa các thành phần sẽ được cài đặt và các phụ thuộc của chúng trong các lớp thiết kế.

13.2 Vẽ các thành phần được yêu cầu.

Lời giải:

Có ba lớp thực thể được dùng trong cộng tác là: CarSharer, Journey và Address. Mỗi lớp này sẽ được cài đặt trong một tập tin .java nguồn. Chúng ta đã biết rằng các lớp này được dùng trong một số các use case và được đặt trong thành phần CarSharing như một tập tin .class. Ngoài ra còn có hai lớp khác là MCSUserInterface và MCSControl. Các lớp này được cài đặt trong một tập tin .java. Hình 13.13 biểu diễn các thành phần này.

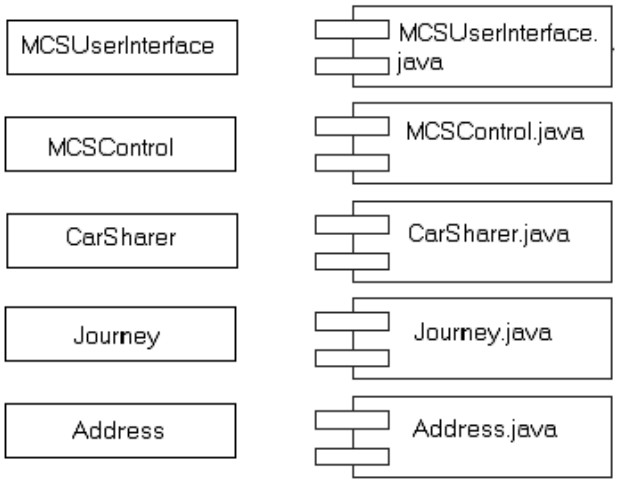


Hình 13.33: Các thành phần mã nguồn của use case Match car sharer

13.3 Có những thành phần nào khác được yêu cầu trong biểu đồ này?

Lời giải:

Các lớp được cài đặt bởi các thành phần này cần được vẽ, đó là *CarSharer*, *Journey*, *Address*, *MCSUserInterface* và *MCSControl*, được biểu diễn trong hình 13.4.

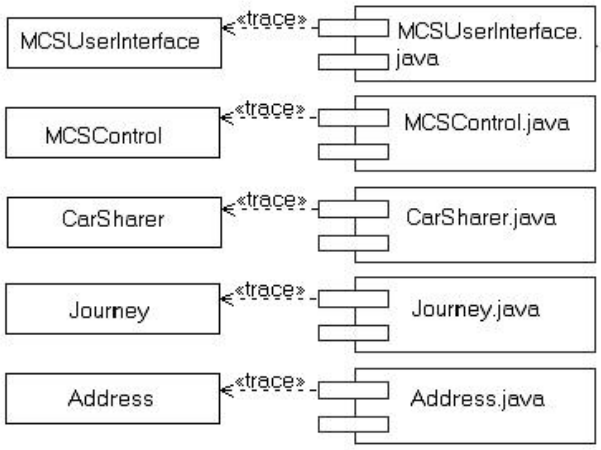


Hình 13.34: Các thành phần mã nguồn và các lớp

13.4 Những phụ thuộc nào cần được thêm vào biểu đồ?

Lời giải:

Có một phụ thuộc *<<trace>>* giữa mỗi lớp và thành phần cài đặt nó. Các phụ thuộc này được vẽ trong hình 13.35. Lưu ý rằng ở đây chỉ có các phụ thuộc đơn giản vì có sự tương ứng một-một giữa các lớp và các tập tin nguồn. Điều này không phải luôn luôn đúng trong mọi trường hợp.

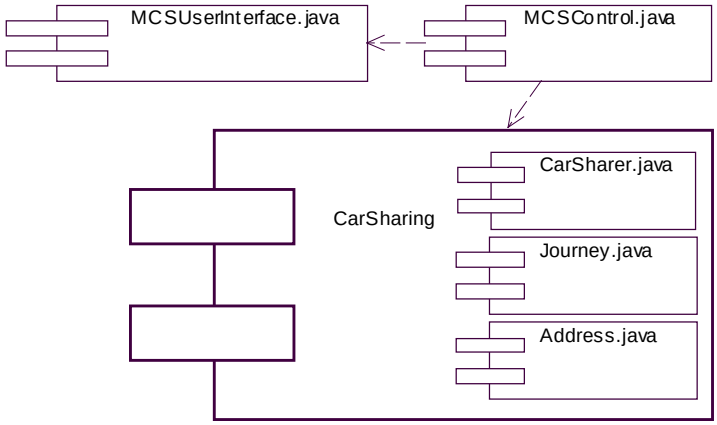


Hình 13.35: Các thành phần mã nguồn và phụ thuộc <<trace>>

13.5 Biểu đồ thứ hai được dùng để mô hình các phụ thuộc giữa các thành phần mã nguồn. Hãy vẽ biểu đồ thành phần.

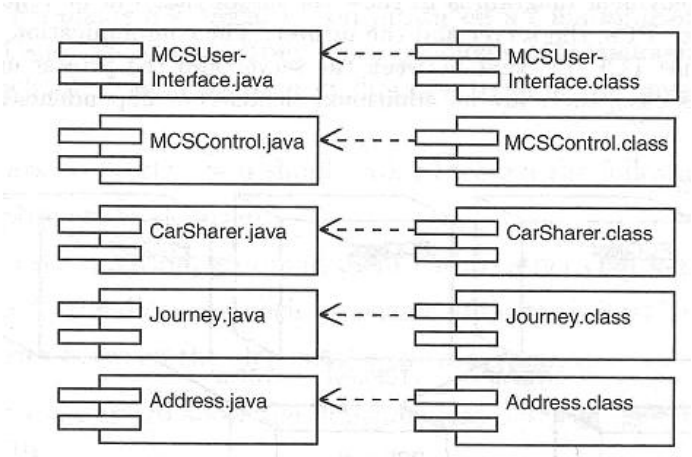
Lời giải:

Mục đích của biểu đồ thành phần này là mô hình các phụ thuộc giữa các tập tin nguồn. các thành phần là các tập tin nguồn mà chúng ta đã xác định bên trên. Các tập tin nguồn sẽ được biên dịch thành thành phần CarSharing có thể được đặt chung nhóm với nhau trong một thành phần. Thành phần MCSControl có phụ thuộc với thành phần CarSharing thành phần MCSUserInterface (Hình 13.36).



Hình 13.36: Các phụ thuộc mã nguồn

13.6 Biểu đồ thứ ba được dùng để mô hình mã nguồn và các tập tin thi hành như các thành phần; và phụ thuộc giữa chúng. Hãy vẽ biểu đồ thành phần.

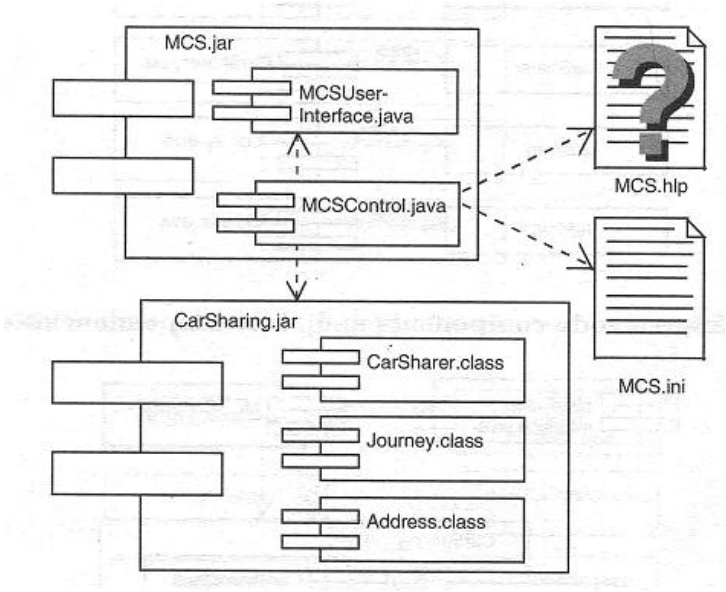


Hình 13.37 Phụ thuộc của các tập tin .class *vào mã nguồn*

Lời giải:

Mục đích của biểu đồ này là mô hình các phụ thuộc giữa các tập tin thi hành vào các thành phần mã nguồn. Mỗi thành phần *.java* sẽ được biên dịch thành một tập tin *.class* (Xem hình 13.37)

13.7 Biểu đồ cuối cùng được dùng để mô hình các thành phần hỗ trợ khác và các phụ thuộc giữa các tập tin thi hành với các thành phần này. Một tập tin giúp đỡ và một tập tin cấu hình được yêu cầu trong trường hợp này. Hãy vẽ biểu đồ thành phần biểu diễn chúng.



Hình 13.38: Các phụ thuộc thành phần khi thi hành.

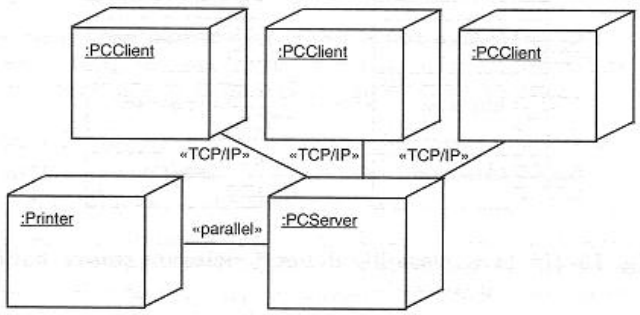
Lời giải:

Mục đích ở đây là biểu diễn các phụ thuộc giữa các thành phần trong hệ thống khi thi hành. Các tập tin .class sẽ được đóng gói vào hai tập tin .jar. Thành phần *MCSControl* sẽ cần đọc tập tin cấu hình và hiển thị tập tin giúp đỡ khi được yêu cầu. Vì vậy, thành phần này phụ thuộc vào hai thành phần bổ sung này. Ngoài ra nó còn phụ thuộc vào thành phần CarSharing.jar và thành phần *MCSUserInterface* (xem hình 13.38).

13.8 Công việc kế tiếp là vẽ một biểu đồ triển khai chỉ biểu diễn các *node* trong hệ thống này. Việc kết hợp các thành viên diễn ra trên một máy *client* bất kỳ, các máy *client* được kết nối tới *server* bằng giao thức TCP/IP, nhưng các đối tượng *CarSharing* sẽ nằm trên *server*, vì thế, cần vẽ thêm *server* trong biểu đồ. Một máy in song song được nối với *server* để in ra chi tiết của những thành viên đã được kết hợp. Hãy vẽ biểu đồ triển khai.

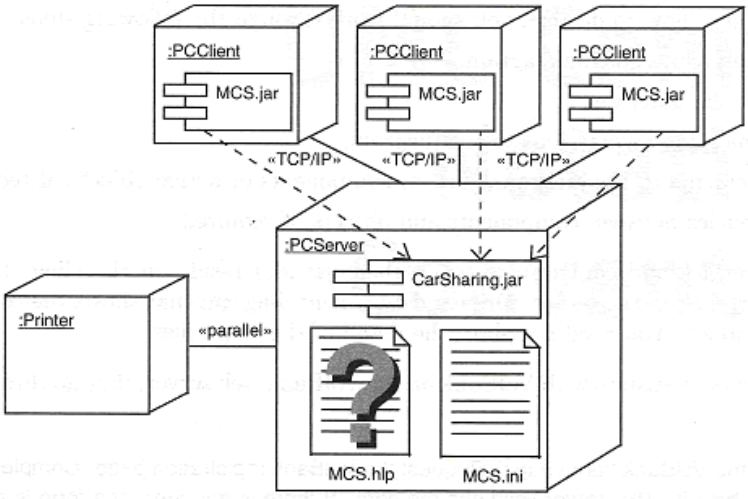
Lời giải:

Mục đích của biểu đồ triển khai này là biểu diễn các node sẽ được yêu cầu trong hệ thống. Các node bao gồm ba máy *client*, *server* và máy in (*printer*). *Server* và *client* sẽ liên lạc với nhau bằng giao thức TPC/IP; máy in và *server* dùng giao thức máy in song song chuẩn để truyền thông. Trong trường hợp này, không bổ sung các thành phần hoặc phụ thuộc nào khác giữa các *node*.



Hình 13.39: Biểu đồ triển khai chỉ mô tả các node.

13.9 Cuối cùng, chúng ta cần mô hình sự triển khai các thành phần trên các nút trong hệ thống. Hãy vẽ một biểu đồ triển khai cho các thành phần trong bài tập 13.7 trên các nút của bài tập 13.8.



Hình 13.40: Biểu đồ triển khai với các thành phần.

Lời giải:

Mục đích của biểu đồ này là biểu diễn việc triển khai các thành phần cụ thể. (Lưu ý rằng có nhiều thành phần thực thi khác phải được triển khai để hệ thống này hoạt động: *môi trường thực thi Java*, các *driver máy in*, *cơ sở dữ liệu*, ngay cả *hệ điều*

hành trên mỗi PC. Tuy nhiên, chúng ta giới hạn biểu đồ này ở các đối tượng chúng ta quan tâm đối với mục đích riêng ở đây).

Các đối tượng *interface* và các đối tượng *control* sẽ chạy trên các *client*, vì vậy thành phần *MCS.jar* cần phải có trên mỗi *client*. Các thành phần khác nằm ở *server*, xem hình 13.40.

Bài tập bổ sung

13.10 Kể từ bài tập 8.9 trở đi, bạn đã giới thiệu một biểu đồ cộng tác cho use case *Process payment*. Dùng các lớp từ các bài tập này (*PPUserInterface*, *PPControl*, *CarSharer*, *Account* và *Transaction*), hãy vẽ một biểu đồ thành phần biểu diễn các tập tin *.java* cài đặt mỗi lớp vừa nêu.

Nếu bạn không biết cách thực hiện, bạn nên thực hiện các bước sau:

- Xác định mục đích của biểu đồ.
- Thêm các thành phần vào biểu đồ, gom nhóm chúng lại bên trong các thành phần khác nếu thích hợp.
- Thêm các thành phần khác vào biểu đồ, chẳng hạn như các lớp, các đối tượng hoặc các giao diện.
- Thêm các phụ thuộc giữa các thành phần của biểu đồ.

13.11 Hãy vẽ một biểu đồ thành phần mô tả các phụ thuộc giữa các tập tin *.java* cho các thành phần trong phần lời giải của bài tập 13.10.

13.12 Hãy vẽ một biểu đồ thành phần mô tả các phụ thuộc giữa các tập tin *.class* và các tập tin mã nguồn *.java* từ lời giải của bạn trong bài tập 13.11.

13.13 Tập tin *CarSharer.class* đã ở trong gói *CarSharing.jar*. Use case này yêu cầu hai lớp ở trong một gói Java khác. Hãy chọn tên thích hợp cho gói và mô tả các lớp này trong một thành phần độc lập để biểu diễn gói Java này.

13.14 Trong môi trường thực thi đối với use case *Process payments*, sẽ có một tập tin cấu hình, một tập tin tỷ lệ thuế (tax rate file) và một tập tin trợ giúp. Có các phụ thuộc giữa tập tin *PPControl.class* và các tập tin cấu hình và trợ giúp, giữa tập tin *Transaction.class* và tập tin tỷ lệ thuế. Hãy vẽ một biểu đồ thành phần để chứa tất cả các tập tin *.class* trong các tập tin *.jar* thích hợp và các phụ thuộc trên ba tập tin bổ sung này.

13.15 Use case *Process payments* chỉ chạy trên một *PC Client* đơn. Nó được kết nối đến một *server* thông qua TCP/IP. Một *modem* được kết nối tới cổng *RS232* trên *server* để truyền dữ liệu đến hệ thống ngân hàng. Hãy vẽ một biểu đồ triển khai.

Nếu bạn không biết cách làm, hãy thực hiện theo các bước sau:

- Xác định mục đích của biểu đồ.
- Thêm các nút vào biểu đồ.
- Thêm các kết hợp truyền thông vào biểu đồ.
- Thêm các thành phần khác vào biểu đồ, chẳng hạn như các thành phần hoặc các đối tượng hoạt động (nếu cần).
- Thêm các phụ thuộc giữa các thành phần và các đối tượng nếu cần.

13.16 Các lớp *interface* và lớp *control* (trong tập tin *.jar* của chúng) thuộc về *Client PC*; tất cả các thành phần khác thuộc về *server*. Hãy vẽ một biểu đồ triển khai biểu diễn các thành phần trên các nút thích hợp. Bạn không cần phải biểu diễn chi tiết của các tập tin *.class*.

13.17 Tình nguyện viên đăng ký với *VolBank* trên *web-server* qua tương tác sau:

> Đến trang chủ *VolBank*. Truy cập trang đăng ký. Điền thông tin vào biểu mẫu và *submit*. *Server* xác nhận tính hợp lệ của dữ liệu. Nếu có bất kỳ lỗi nào thì biểu mẫu sẽ được hiển thị lại với nội dung vừa nhập và trường có thông tin sai sẽ được nổi bật lên; ngược lại, dữ liệu được *submit* vào cơ sở dữ liệu và một xác nhận được hiển thị.

Hãy vẽ một biểu đồ thành phần biểu diễn các thành phần được hiển thị trong *trình duyệt web* (web-browser) trên máy *client* với tất cả các phụ thuộc mà bạn thấy thích hợp.

13.18 Đối với *scenario* của bài tập 13.17, hãy vẽ một biểu đồ thành phần biểu diễn các thành phần trên *server*. Chúng là một chương trình *web-server*, một *hệ quản trị cơ sở dữ liệu* (DBMS) và một *Java servlet* (một ứng dụng *Java* chạy trên *server*) để xử lý đăng ký.

13.19 Đối với *scenario* của bài tập 13.17, hãy vẽ một biểu đồ triển khai biểu diễn các thành phần được triển khai trên *client* hoặc trên *server*. Hãy đặt một kết hợp truyền thông dùng giao thức HTTP.

Chương 14
CÁC QUI ƯỚC KÝ HIỆU CHUNG CỦA UML

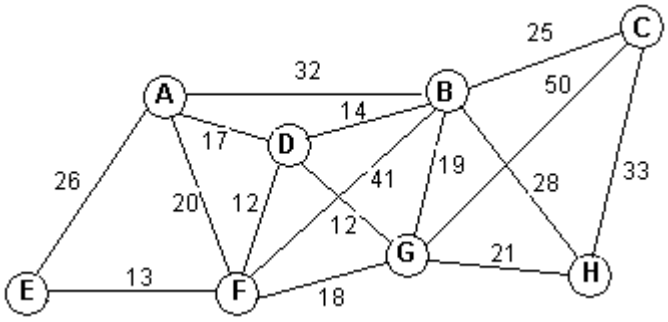
14.1 Giới thiệu

Nhiều biểu đồ UML có các đặc điểm chung với nhau. Trong phạm vi ký hiệu, có một số cách biểu diễn chuẩn, mô tả những gì mà người phát triển hệ thống quan tâm, và các thành phần chuẩn này đã được xác định ngay từ đầu trong đặc tả UML (OMG, 1999a). Chúng ta thêm chương này để tập hợp các đặc trưng chung lại để bạn đọc có thể tham chiếu khi làm việc với các chương khác.

Trong chương này, trước hết chúng ta điểm qua các thành phần biểu đồ thông dụng được sử dụng rộng rãi trong UML. Sau đó, chúng ta đi tìm hiểu các cơ chế mở rộng tồn tại bên trong UML làm cho UML có khả năng mở rộng hoặc chuyên biệt nó cho các loại hệ thống đặc biệt. Chương này cũng trình bày một ví dụ về cách áp dụng các cơ chế này vào công việc của *Jim Conallen* trong việc dùng UML để lập mô hình các ứng dụng *web* (Conallen 1999).

14.2 Các thành phần phổ biến trong một biểu đồ

14.2.1 Đồ thị

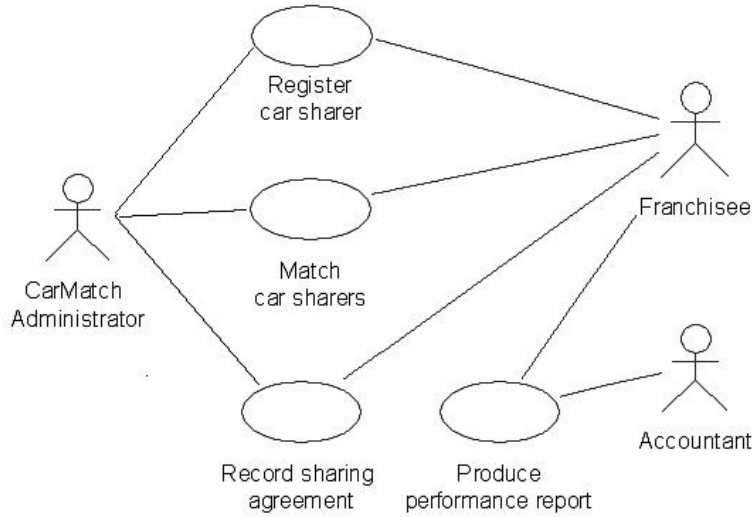


Hình 14.1: Đồ thị biểu diễn bài toán người bán hàng

Hầu hết các biểu đồ trong UML đều là *đồ thị*. Đồ thị ở đây không giống với biểu *đồ số* (histogram) hoặc một *đồ thị x-y* (x-y graph) được dùng để biểu diễn thông tin thống kê. Một đồ thị là một tập các *nút* (node) được

nối với nhau bằng các *đường* (path). Đồ thị được dùng để biểu diễn thông tin về nhiều loại vấn đề. Thường thì thuật ngữ *đỉnh* (vertex) được dùng thay cho nút, và *cạnh* (edge) hoặc *cung* (arc) được dùng thay cho đường. Hình 14.1 là một ví dụ về đồ thị biểu diễn *bài toán hành trình của người bán hàng*, TSP (travelling salesperson problem). Cụ thể người bán hàng muốn *thăm* tất cả các *địa phương* sao cho hành trình là ngắn nhất, và mỗi nơi chỉ một lần. Trong hình, các nút được biểu diễn bằng các ký tự, các nhãn số trên cạnh là khoảng cách giữa các nút.

Đồ thị được sử dụng rộng rãi trong UML. Hình 14.2 mô tả một biểu đồ use case, trong đó có hai loại nút – *actor* và *use case* – và các đường nối giữa chúng.



Hình 14.2: Ví dụ về một đồ thị - một biểu đồ use case

14.2.2 Các mối quan hệ trực quan trong đồ thị

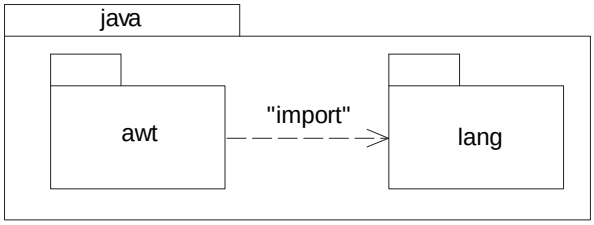
Với hầu hết các *đồ thị* trong UML, mối quan hệ giữa các biểu tượng quan trọng hơn kích thước của chúng: việc một lớp được vẽ lớn hơn một biểu đồ lớp khác không có nghĩa là chứa nhiều thông tin hơn. Tuy nhiên cũng có một ngoại lệ trong một số biểu đồ tuần tự, trong đó trục thời gian được vẽ theo tỷ lệ, ví dụ một giây được biểu diễn là một centimet.

Có 3 loại quan hệ trực quan quan trọng trong các biểu đồ UML:

- *Connection* (kết nối): các *symbol* (biểu tượng) hoặc các hình hai chiều được kết nối với nhau bằng các đường thẳng.
- *Containment* (chứa): thường là các *symbol* hình hai chiều.

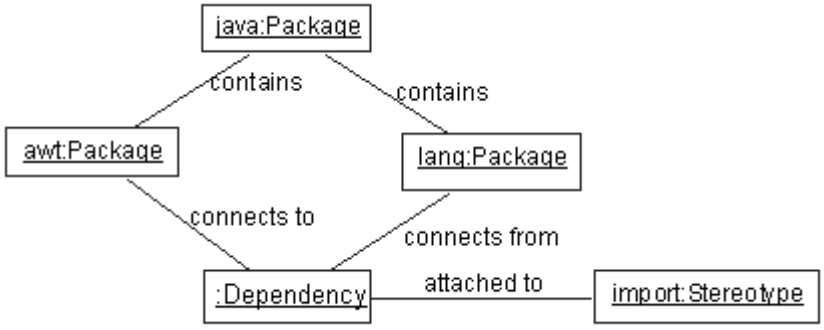
- *Visual attachment* (đính kèm): văn bản hoặc một biểu tượng cạnh một biểu tượng trên biểu đồ.

Các loại quan hệ này được biểu diễn bởi các đồ thị. Trong chương 2, chúng ta đã biết UML là một ngôn ngữ trực quan. Nếu phân tích biểu diễn trực quan vào trong một biểu diễn nội bằng cách dùng các quy tắc ngữ pháp của UML, thì kết quả là một đồ thị. Hình 14.3 trình bày các ví dụ về ba loại quan hệ trực quan này trong một biểu đồ gói.



Hình 14.3: Ví dụ về một biểu đồ gói trình bày các quan hệ trực quan

Hình 14.4 biểu diễn các quan hệ của biểu đồ trong hình 14.3 thành đồ thị.



Hình 14.4: Đồ thị biểu diễn các quan hệ trực quan giữa các phần tử của hình 14.3

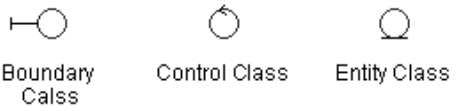
Lưu ý phần tử nằm bên trong hoặc được gắn vào phần tử khác (ví dụ các nhãn), có thể được chuyển thành các đường trong đồ thị. Các công cụ CASE dùng một số loại biểu diễn nội tại như thế để giữ chi tiết quan hệ giữa các cặp phần tử mô hình. Đối với hầu hết các công cụ CASE, metamodel của UML được sử dụng cho mục đích này, xem chương 2.

Có 4 loại phần tử được dùng trong ký hiệu UML: *biểu tượng* (icon), *biểu tượng hai chiều* (two-dimensional symbol), *đường* (path) và *chuỗi* (string). Sau đây chúng ta sẽ tìm hiểu chi tiết từng phần tử:

- *Icon*: một phần tử đồ hoạ có hình dạng và kích thước cố định. Trong UML, *icon* thường được sử dụng như các khuôn dạng của các lớp (xem mục 14.3.1). Các icon được dùng như các phần tử đứng một mình trong các biểu đồ, được dùng bên trong các biểu đồ khác, hoặc được dùng làm các điểm kết thúc của đường đi.
- *Two-dimensional symbol*: được sử dụng để chứa các biểu tượng khác và sẽ thay đổi kích thước một cách thích hợp. Chúng có thể được chia thành các ngăn riêng. Đường dùng để nối các *two-dimensional symbol* và các *icon* lại với nhau. Trong một công cụ CASE, nếu bạn di chuyển một *two-dimensional symbol*, tất cả nội dung và đường nối đều di chuyển theo.
- *Path*: dùng để nối các *symbol* và các *icon*. *Path* được tạo từ đường thẳng. Các đầu mút của nó phải luôn được gắn với các phần tử. *Path* nối các phần tử kết thúc để truyền đạt ý nghĩa cho nó.
- *String*: dùng để mô tả một dãy thông tin dạng văn bản. *String* phải được viết bởi một ngôn ngữ nào đó, có thể phân tích cú pháp thành thông tin bên trong mô hình.

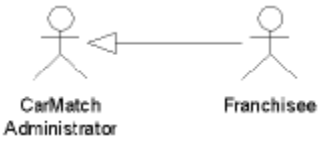
14.2.3 Biểu tượng (*Icon*)

Biểu tượng được sử dụng để biểu diễn các thành phần đồ hoạ khuôn dạng trong UML. Ví dụ, biểu tượng hình người được dùng để biểu diễn *actor*. UP dùng biểu tượng biểu diễn ba loại lớp khác nhau: *boundary*, *control* và *entity* (hình 14.15).



Hình 14.15: Ví dụ về các biểu tượng trong UP

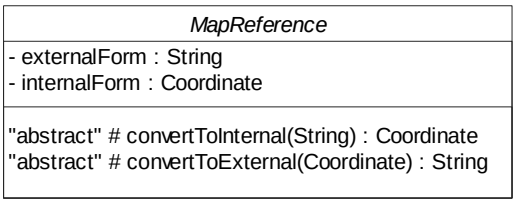
Biểu tượng cũng được dùng trên các đầu mút của đường, ví dụ trong hình 14.6 hình tam giác được dùng để biểu diễn quan hệ generalization.



Hình 14.6: Một biểu tượng làm đầu cuối của đường

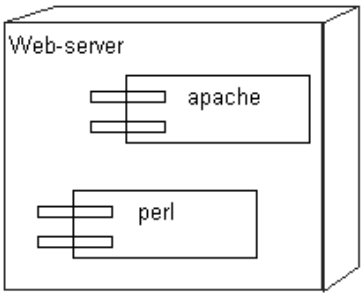
14.2.4 Biểu tượng hai chiều (two-dimensional symbol)

Biểu tượng hai chiều được dùng để trình bày nhiều thành phần bên trong UML. Ví dụ như lớp trong biểu đồ lớp. Hình 14.7 là một ví dụ về lớp được chia thành các ngăn: tên, thuộc tính và thao tác.



Hình 14.7: Ví dụ về một lớp như một biểu tượng hai chiều có các ngăn

Biểu tượng hai chiều có thể chứa các biểu tượng khác, kể cả icon. Hình 14.8 trình bày một biểu đồ triển khai có các icon đại diện với các tập tin khả thi được triển khai trên nút Web-Server.



Hình 14.8: Biểu đồ triển khai làm biểu tượng hai chiều chứa các icon

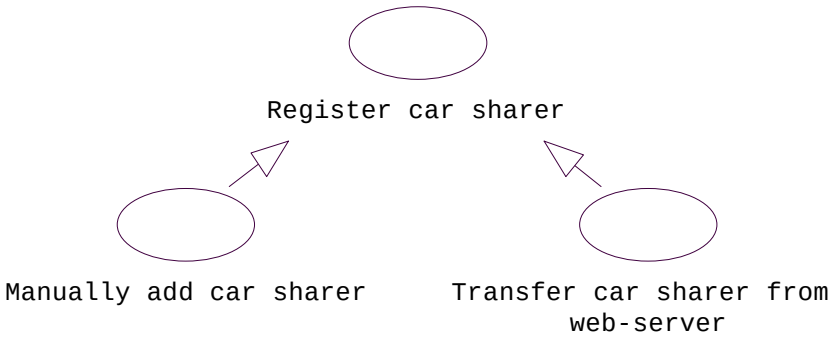
Dĩ nhiên, biểu tượng này biểu diễn nút *Web Server* là một hình chiếu hai chiều của một biểu tượng ba chiều. Hiện tại UML chỉ muốn các biểu đồ được biểu diễn ở mức độ hai chiều.

14.2.5 Đường (path)

Đường được dùng để biểu diễn tất cả các liên kết giữa các biểu tượng trong biểu đồ UML. Đường được tạo từ các đoạn thẳng. Một công cụ CASE có thể sử dụng một đoạn thẳng riêng, nhưng một đoạn thẳng riêng không thể tồn tại nếu không có phần còn lại của đường. Như đã đề cập, các đường phải luôn được gắn với một vài symbol ở cả hai đầu mút.

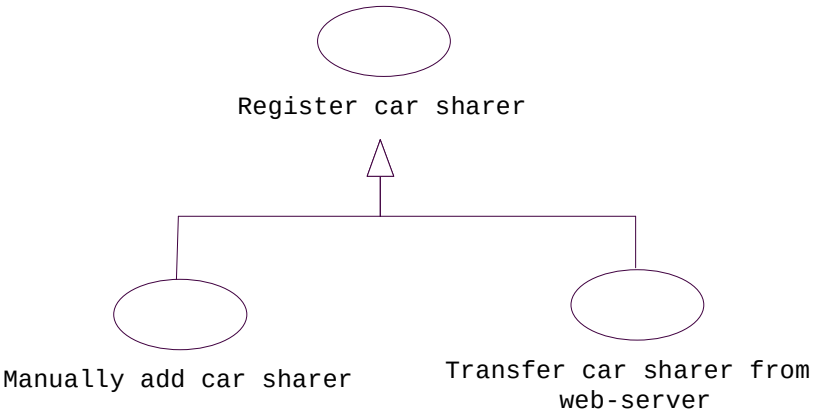
Trong một biểu đồ không thể tồn tại một đường đi có một đầu tự do không gắn với đối tượng nào. Đường có thể có các icon kết thúc ngay điểm liên kết với các symbol khác.

Hình 14.9 biểu diễn hai đường, mỗi đường được tạo ra từ một đoạn (có thể nhiều đoạn) và một đầu mút là một icon biểu diễn tổng quát hoá.



Hình 14.9: Các đường tạo từ các đoạn thẳng và kết thúc bằng các icon

Một số đường có thể được kết hợp thành cấu trúc cây phân nhánh như hình 14.10. Tuy nhiên biểu hiện vật lý khi kết hợp hai đường, che giấu biến cố chúng vẫn là hai đường đi riêng biệt về mặt khái niệm.

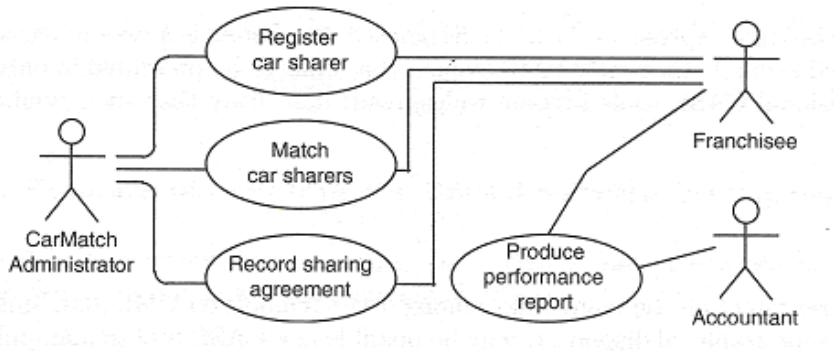


Hình 14.10: Ví dụ về đường rẽ nhánh

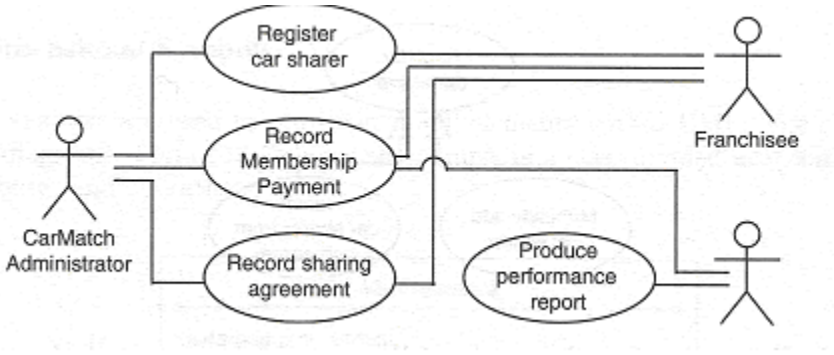
Việc chọn lựa cách bố trí các đường trên biểu đồ chỉ là vấn đề biểu diễn và có thể được cấu hình trong một công cụ CASE. Hình 14.11 minh hoạ một vài tuỳ chọn điển hình. Bình thường chỉ có một kiểu sẽ được sử dụng trong một mô hình.

Nếu được, đường có thể được vẽ lại để tránh việc chồng chéo nhau, trường hợp gặp những đoạn thẳng chéo nhau không thể tránh được thì ta dùng một *'jog'* (đường cong hình khủy tay) nhỏ chèn vào chỗ chéo đó, ý nghĩa của việc làm này ngụ ý rằng các đường thẳng chéo nhau chứ không nối nhau.

Ví dụ về 'jog' được minh họa trong hình 14.12 (biểu đồ này cũng có thể được vẽ lại mà không cần dùng *'jog'*).



Hình 14.11: Các kiểu đường



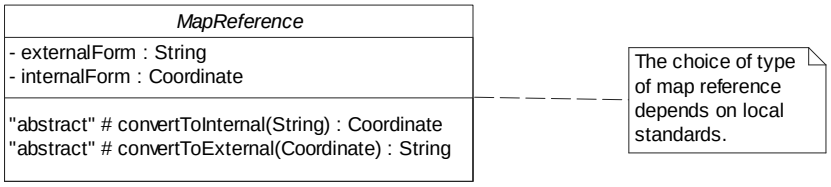
Hình 14.12: Jog được dùng để biểu diễn nơi các đường đi giao nhau

14.2.6 Chuỗi (*string*)

Chuỗi được dùng để mô tả thông tin cho mô hình. Chuỗi là một dãy các ký tự trong một bộ ký tự nào đó, không giới hạn trong tập ký tự *Roman*. UML cho rằng bộ ký tự là khả năng quản lý các ký tự nhiều byte. Các bộ ký tự 8-bit, chẳng hạn như *ASCII* (American Standard Code for Information Interchange) không thể quản lý nhiều hơn 256 mã ký tự khác nhau, nên không thể biểu diễn đầy đủ dãy ký tự mà con người sử

dụng. Unicode có được khả năng này mặc dù nó không có ý định dùng trong đặc tả UML.

Chuỗi được hiển thị trong UML thành các hình ảnh chuỗi văn bản. Những ký tự có khả năng in thì nên được in một cách trực tiếp. Những ký tự không thể in được (các mã điều khiển) là các ký tự *phụ thuộc* (platform-dependent). Chuỗi có thể được in ra thành một dòng hoặc một đoạn có dấu ngắt dòng. Việc chọn *font* chữ có thể được dùng để cho biết thuộc tính của mô hình. Ví dụ, trong UML các chữ in nghiêng được dùng để chỉ định tên lớp trừu tượng. Hình 14.14 biểu diễn cách dùng của nhiều loại chuỗi khác nhau.



Hình 14.13: Ví dụ về cách dùng chuỗi trong UML

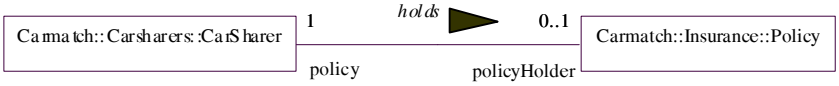
14.2.7 Tên (*name*)

Chuỗi được dùng làm *tên* để xác định các thành phần của mô hình. Các thành phần của mô hình được xác định một cách duy nhất trong một phạm vi nào đó. Phạm vi có thể là một hệ thống, một gói hoặc một lớp. Ví dụ, các lớp khác nhau có thể chứa các thuộc tính hoặc các thao tác cùng tên nhưng có thể phân biệt được với nhau nhờ vào tên lớp mà chúng phụ thuộc. Thuật ngữ *namespace* được dùng để xác định phạm vi của một tên

Pathname (tên đường dẫn) có thể được dùng tường minh để mô tả phạm vi của tên. Trong UML, các gói là các cơ chế bình thường để định nghĩa *namespace* (xem chương 2), và tên gói được liên kết với nhau bằng cách dùng một cặp dấu hai chấm (::) làm dấu phân cách, ví dụ *java*::*awt*::*Applet* hoặc *CarMatch*::*Insurance*::*Policy*.

14.2.8 Nhãn (label)

Chuỗi cũng có thể được dùng để làm *nhãn* trên các phần tử của biểu đồ. Nhãn được gắn (dạng hình ảnh) vào các symbol trên của biểu đồ (đặt vào bên trong hoặc hiển thị kề với chúng). Trong một công cụ CASE, nếu symbol bị di chuyển, thì nhãn gắn với symbol cũng di chuyển theo. Hình 14.14 là các ví dụ về nhãn chứa bên trong và kề với các symbol khác, kể cả tên của hai lớp có đầy đủ đường dẫn.



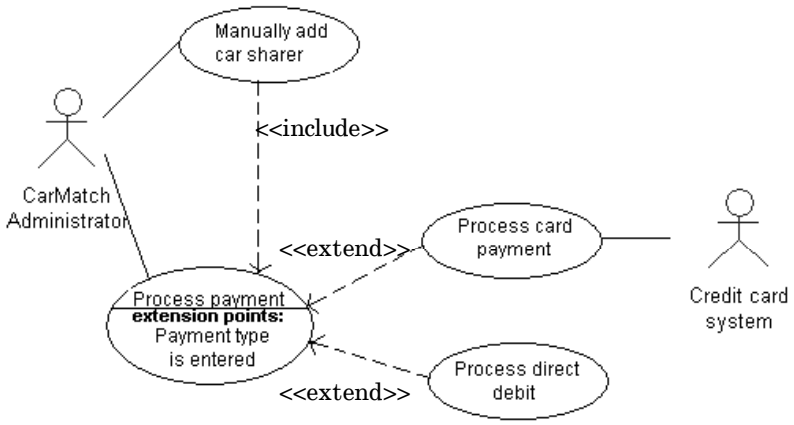
Hình 14.14: Ví dụ về nhãn và tên trong UML

14.2.9 Từ khoá (*keyword*)

Từ khoá được dùng trong UML để phân biệt các phần tử của mô hình nếu như chúng có các trình bày trực quan giống nhau không thể phân biệt. Ví dụ, có nhiều loại phụ thuộc trong một mô hình, tất cả chúng đều được biểu diễn bằng các mũi tên mở đứt nét. Việc dùng từ khoá sẽ phân biệt được những loại phụ thuộc khác nhau này.

Hình 14.15 mô tả hai loại quan hệ khác nhau trong một biểu đồ use case, các quan hệ này được phân biệt nhau nhờ có dùng từ khoá.

Phụ lục A của đặc tả UML sẽ liệt kê danh sách các từ khoá như là các phần tử chuẩn. Chúng là các từ dành riêng cho UML và không nên dùng cho mục đích định nghĩa. Người sử dụng có thể dùng những từ khoá khác như là các *stereotype* để mở rộng cách dùng UML (xem phần 14.3.1).



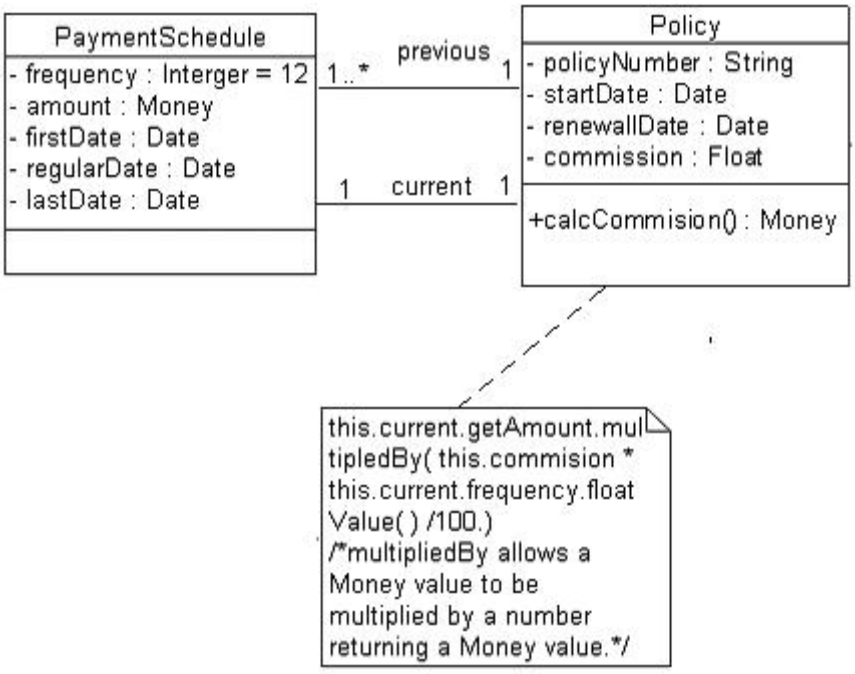
Hình 14.15: Ví dụ về từ khóa trong UML

14.2.10 Biểu thức (expression)

Một *biểu thức* là một chuỗi được hiển thị trong hình chữ nhật. Giả sử biểu thức có thể được định trị thích hợp hoặc có thể được phân tích để tạo ra một kết quả có ý nghĩa. Biểu thức có thể là một ngôn ngữ sẽ được công cụ CASE sử dụng để phát sinh ra mã, và sẽ được đặt trực tiếp vào đoạn mã phát sinh, một số ngôn ngữ đặc tả sẽ được biến đổi thành một

biểu thức trong ngôn ngữ đích, hoặc có thể được sử dụng bởi công cụ lúc thực thi để mô phỏng một số thành phần đang chạy của hệ thống.

Việc chọn lựa ngôn ngữ tuỳ thuộc vào người dùng và các tuỳ chọn có sẵn trong các công cụ CASE đang được sử dụng. Các biểu thức có thể được dùng trong các đặc tả của thao tác, làm các giá trị mặc định cho các thuộc tính, các ràng buộc (xem mục 14.3.3) hoặc như các đoạn mã trong ghi chú (xem mục 14.2.11). Hình 14.16 biểu diễn các ví dụ của một vài biểu thức. Ghi chú này có chứa một phần mã Java để tính hoa hồng cho các hợp đồng bảo hiểm.



Hình 14.16: Ví dụ về biểu thức trong UML

Biểu thức có thể được định nghĩa trong ngôn ngữ ràng buộc đối tượng OCL, xem chi tiết về OCL trong chương 12. Đặc tả cho OCL là một phần của đặc tả UML. UML đặc biệt được dùng để diễn tả các ràng buộc, chẳng hạn như *pre-condition* và *post-condition* cho các thao tác.

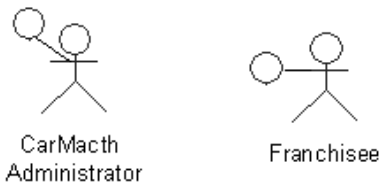
14.2.11 Ghi chú (note)

Trong chương này, bạn đã thấy cách sử dụng các *ghi chú* trong các biểu đồ. Ghi chú là một biểu tượng hai chiều chứa một chuỗi. Chuỗi có thể là một ràng buộc, một đoạn mã của một ngôn ngữ nào đó, hoặc là một lưu ý về một ngôn ngữ tự nhiên. Hình 14.13 và 14.16 minh hoạ cho cách sử

dụng ghi chú trong biểu đồ lớp. Ghi chú được vẽ bằng cách dùng một hình chữ nhật bị xếp góc ở góc phải bên trên, chúng được liên kết với thành phần đang lập sưu liệu bằng một dấu gạch xuôi. Trong hình 14.13 ghi chú mô tả cho lớp, trong hình 14.16 ghi chú mô tả cho thao tác. Ta có thể bỏ ghi chú ra khỏi biểu đồ mà không làm mất thông tin. Ghi chú được đặt trên biểu đồ nếu không có dấu gạch xuôi thì nó diễn đạt biểu đồ theo một vài cách. Ví dụ, để cung cấp chi tiết thông tin về tác giả thì ghi chú ngày lập hoặc thông tin nguồn của mô hình.

14.2.12 Classifier

Classifier là một thuật ngữ mà chúng tôi đã cố gắng tránh sử dụng trong quyển sách này mặc dù nó được sử dụng rộng rãi trong đặc tả UML, bởi nó là trung tâm ngữ nghĩa của UML. Tuy nhiên, rất khó giải thích *classifier*. Một *classifier* là một thành phần mô hình mô tả các đặc điểm thuộc hành vi và cấu trúc của mô hình. *Classifier* gồm có các *lớp* (class), các *tác nhân* (actor), *use case*, *kiểu dữ liệu* (data type), *thành phần* (component), *giao diện* (interface), *tín hiệu* (signal), *nút* (node) và các *hệ thống con* (subsystem). Trong *metamodel* của UML, các *classifier* là các chuyên biệt hoá của cả *GeneralizableElement* và *Namespace*, cả hai đều là chuyên biệt hoá của *NodeElement*. Giống với các *namespace*, *classifier* có thể có các đặc trưng như thuộc tính và thao tác và được xác định duy nhất thuộc về classifier nào đó. Giống với các thành phần có khả năng tổng quát hoá, *classifier* có thể được chuyên biệt hoá xa hơn trong kiến trúc phân cấp thừa kế; khuôn dạng sẽ (xem mục 14.3.1) cung cấp một cơ chế cho điều này. *Boundary*, *Control* và *Entity* của UP là các đặc tả khuôn dạng của Class. Những gì áp dụng cho *classifier* cũng áp dụng cho tất cả các chuyên biệt hoá của nó. Ví dụ, classifier có thể tham gia trong kết hợp hoặc trong máy trạng thái của nó. Classifier cũng có thể kết hợp với giao diện. Ví dụ, một *actor* có thể được vẽ với một giao diện, mặc dù đặc tả UML không rõ ràng về ký hiệu và chúng ta không thể biết nên hiểu các ký hiệu trong hình 14.17 là một *tác nhân hai đầu* hay một *hành động chơi tennis*.

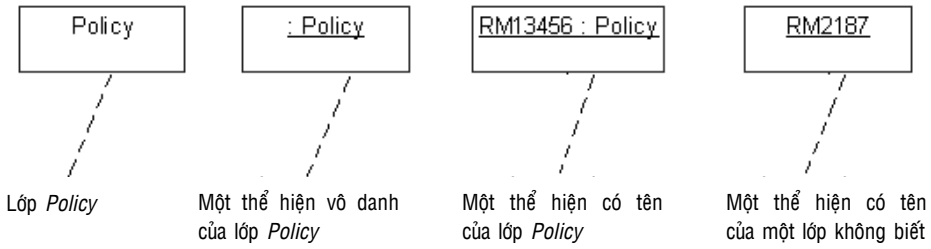


Hình 14.17: Các ký hiệu có thể dùng cho tác nhân và giao diện

Một số phần không phải là *classifier*. Chúng bao gồm các cộng tác, tương tác, quan hệ và trạng thái.

14.2.13 Kiểu (type), thể hiện (instance) và vai trò (role)

Nhiều mô hình UML định nghĩa các trường hợp tổng quát, chúng được tạo thể hiện trong hệ thống đang làm việc: lớp và đối tượng, thuộc tính và liên kết, tham số và giá trị. Cách phân biệt kiểu và giá trị được giải thích chi tiết trong các chương lớp và đối tượng (xem từ chương 2 đến chương 7). Trong các biểu đồ UML, sự phân biệt được tạo ra giữa các kiểu và thể hiện theo cách sau đây. Kiểu và thể hiện đều được vẽ bằng cách dùng chung biểu tượng hình học. Tên của thể hiện được gạch dưới. Tên của thể hiện có thể là chuỗi xác định thể hiện, hoặc là chuỗi và theo sau là dấu hai chấm rồi đến tên của kiểu (nếu biết), hoặc là một dấu hai chấm và theo sau là tên của kiểu. Chúng được hiển thị trong Hình 14.18.



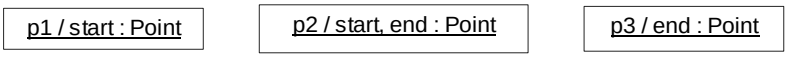
Hình 14.18: Lớp và một vài thể hiện

Trong UML ta không phân biệt tên thể hiện là định danh duy nhất của thể hiện hoặc tương đương với loại thuộc tính hoặc tên biến. Trong mục *kiến trúc ngôn ngữ* của đặc tả UML, các thể hiện đối tượng có tên như *<Acme_oftware_Share_98789>* (một thể hiện của *StockShare*), trong khi ở những mục khác thì có tên như *P1* (một thể hiện của *Point*) và *Jill* (một thể hiện của *Person*). Trường hợp thứ nhất có thể là một định danh duy nhất, trường hợp thứ hai là tên thuộc tính bên trong thể hiện của lớp *Rectangle*, còn trường hợp cuối rõ ràng không là một tên tốt cho một thể hiện đối tượng: không định danh duy nhất.

Hình 14.19: Ví dụ về vai trò

Thông thường các thể hiện vô danh được dùng để tránh đặt tên. Tuy nhiên, thường có các tình huống mà ở đó nhiều thể hiện của một lớp xuất hiện, như trong một biểu đồ cộng tác. Ví dụ, để vẽ một đường đi có thể dùng một mảng các thể hiện vô danh của *Point*; vẽ đoạn thẳng từ điểm thứ nhất đến điểm thứ hai, từ điểm thứ hai đến điểm thứ ba... cho đến điểm cuối cùng trong mảng. Trong mỗi trường hợp, một điểm đóng vai trò điểm đầu và một điểm đóng vai trò điểm kết thúc của đoạn thẳng. Sau đó điểm kết thúc này sẽ đóng vài trò điểm đầu của đoạn thẳng tiếp theo... cho đến lược điểm cuối cùng trong mảng. Thay vì đặt tên cho các thể hiện, ta nên đặc tả chúng bằng các vai trò như bạn vừa thấy: điểm bắt đầu và điểm kết thúc. Hình 14.19 biểu diễn ký hiệu của các vai trò. Tên vai trò được thể hiện trước tên lớp, cách nhau bởi dấu hai chấm (không có dấu gạch dưới). Vai trò thường được sử dụng trong cộng tác (xem chương 8).

Tên của vai trò có thể được kèm trong các thể hiện, chỉ biết một thể hiện nào đó nhận một vai trò trong cộng tác. Trong trường hợp này, tên của thể hiện được hiển thị trước, cách với tên vai trò bằng một dấu gạch chéo, sau đó là tên của vai trò được phân cách với tên lớp bằng một dấu hai chấm. Hình 14.20 là một ví dụ. Một thể hiện có thể đảm nhận nhiều hơn một vai trò, trong trường hợp này là một danh sách các vai trò cách nhau bởi dấu phẩy. Do chúng ta đang dùng một thể hiện nên tên được gạch dưới.



Hình 14.20: Ví dụ về các thể hiện trong vai trò

14.3 Các cơ chế mở rộng (extension mechanism) trong UML

Khả năng mở rộng là một đặc điểm được tính đến khi thiết kế UML. Mục đích cho phép người phát triển tạo ra các phiên bản chuyên biệt hóa của UML. Có khả năng thực hiện điều này nhờ ba cơ chế tồn tại trong MUL mà không cần phải phát minh ra các thành phần mới, đó là: *sterotype*, *tagged value* và *constraint*.

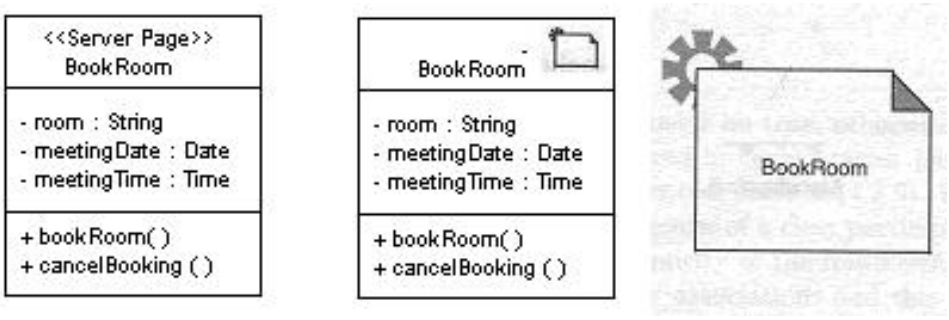
Một số ví dụ sử dụng ở đây lấy từ công việc thực tế của *Jim Conallen* khi dùng UML lập mô hình cho các ứng dụng *Web* (*Conallen*, 2000).

14.3.1 Khuôn dạng (Stereotype)

Một khuôn dạng là một lớp siêu *metamodel* mới, được sử dụng để lập mô hình cho một loại lớp đặc biệt trong hệ thống. Khuôn dạng được sử dụng

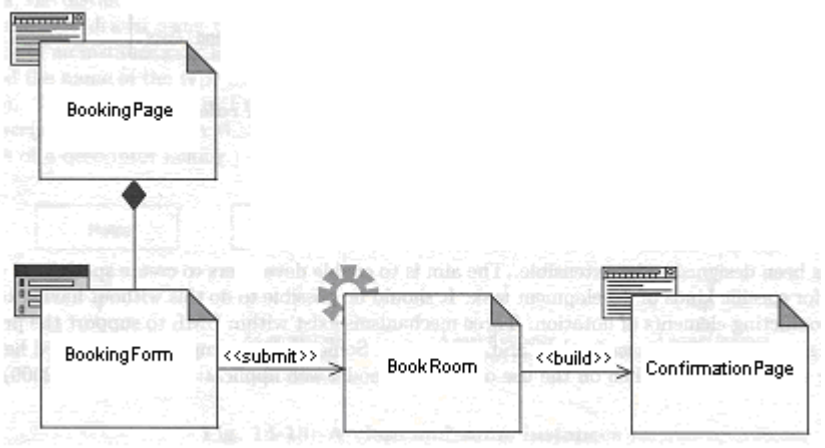
trong chính UML, ví dụ khuôn dạng «*abstract*» được sử dụng được xác định các lớp trừu tượng trong một phân cấp tổng quát hoá, khuôn dạng «*include*» và «*extend*» được sử dụng để phân biệt các phụ thuộc trong biểu đồ use case.

Khuôn dạng thường được sử dụng để tô điểm cho lớp và kết hợp. Tên của khuôn dạng có thể được hiển thị bên trên tên lớp hoặc bên cạnh các kết hợp trong <<.>>.



Hình 14.21: Các ký hiệu khác nhau của các khuôn dạng

Ngoài ra, khuôn dạng có thể được biểu diễn như một icon. Icon này có thể được đặt trong phần đỉnh của biểu tượng lớp, ở góc phải bên trên, hoặc biểu tượng lớp có thể được thu gọn lại thành một icon. Hình 14.21 biểu diễn ba tuỳ chọn của lớp dùng một trong các khuôn dạng từ mô hình ứng dụng *Web* của *Conallen*.

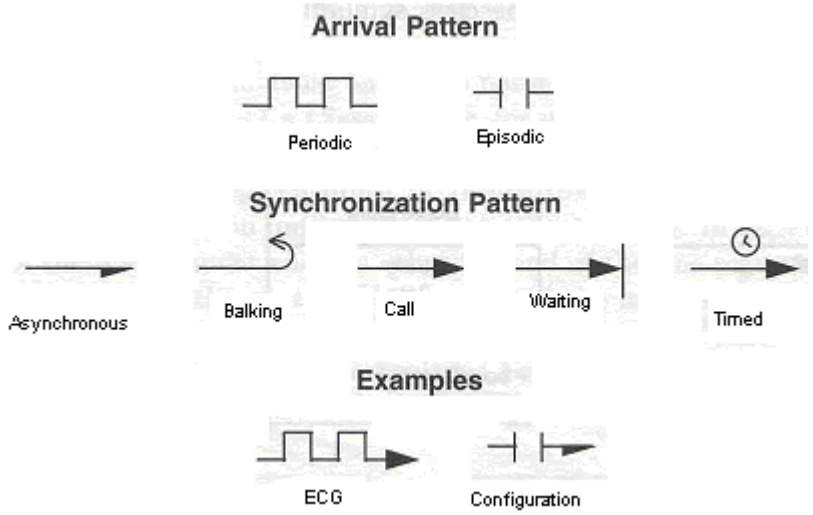


Hình 14.22: Các kết hợp stereotype

Các mở rộng của *Conallen* gồm «*Server page*», «*Form*», «*FrameSet*» và «*Target*». Các kiểu lớp này biểu diễn các đơn vị của ứng dụng Web. Ví dụ,

«*Form*» là một *biểu mẫu HTML* và «*Server Page*» là một trang HTML chứa các *script* được thi hành trên *Web-server*. Các icon cho «*Form*» và «*Client Page*» chứa các phần tử trông giống như các form HTML và các trang web, các icon khác có thiết kế trừu tượng hơn. Conallen cũng tạo khuôn dạng cho các kết hợp nhằm biểu diễn các mối quan hệ giữa các thành phần của một ứng dụng Web, chẳng hạn như «*link*», «*submit*», «*target link*» và «*build*». Hình 14.22 biểu diễn các phần tử này.

Các *stereotype* có thể được áp dụng cho các phần tử mô hình khác. Các mở rộng thời gian thực của *Bruce Powel Douglass* cho ký hiệu UML là các *stereotype* cho các thông điệp (Douglass, 1998). Theo đó, có hai loại *stereotype*: một loại liên quan với kiểu đến của thông điệp và một loại liên quan đến việc đồng bộ hóa chúng. Hai loại này được kết hợp vào các biểu tượng đơn, như biểu diễn trong hình 14.23.



Hình 14.23: Các thông điệp có khuôn dạng

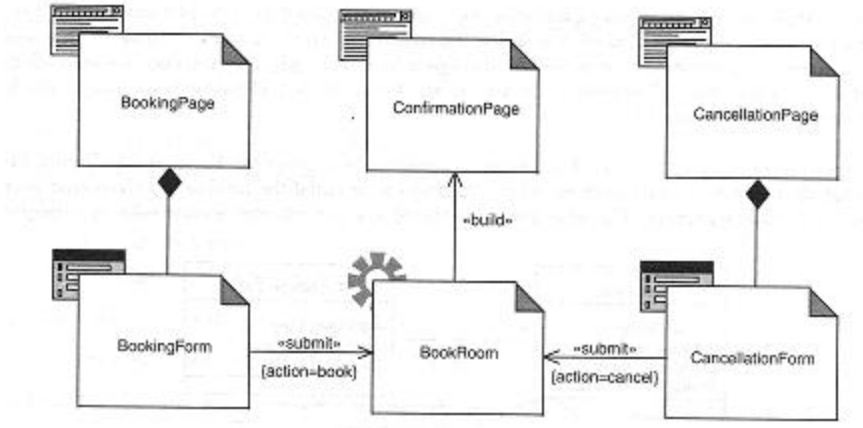
Trong hình, ví dụ về ECG (Electro-Cardiogram – biểu tượng điện tâm đồ) cho biết một giá trị được thiết lập bằng cách gọi một thao tác của đối tượng đích tại những khoảng thời gian đều nhau.

14.3.2 Giá trị thẻ (Tagged value)

Nhiều phần tử mô hình có những thuộc tính không được biểu diễn một cách trực quan. Chẳng hạn như các giá trị do người dùng định nghĩa. *Tagged value* bao gồm một *tag* hoặc tên và giá trị kết hợp. Các *tagged value* có thể được biểu diễn bằng các chuỗi nằm trong cặp dấu ngoặc nhọn.

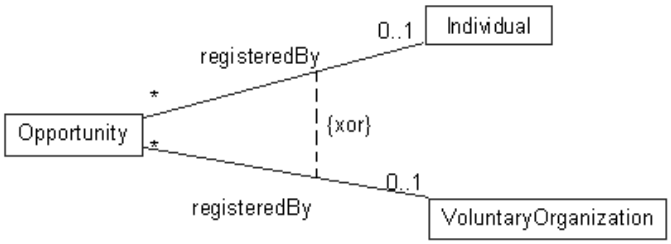
Hình 14.24 minh họa cách sử dụng *tagged value* để hiển thị tên của một trong những tham số truyền từ các *biểu mẫu web* đến chương trình *server* trong ứng dụng *web*, sử dụng các mở rộng của Conallen.

Nếu *tagged value* có giá trị *Boolean* (*true* hoặc *false*), thì giá trị *true* có thể được bỏ qua, ví dụ {*abstract*}. Các thuộc tính *Boolean* thường có dạng {*isPropertyName*}, trong đó *PropertyName* là tên của giá trị, ví dụ {*abstract*} tương đương với {*isAbstract = true*}. *Tagged value* cũng có thể được dùng để cung cấp thông tin về tình trạng của một mô hình, ví dụ {*author* = "*Simon Bennet*", *datw = 16-July-2000*, *status = draft*}. Các giá trị được cách nhau bởi dấu phẩy.



Hình 14.24: Ví dụ về tagged value

14.3.3 Ràng buộc (*Constraint*)



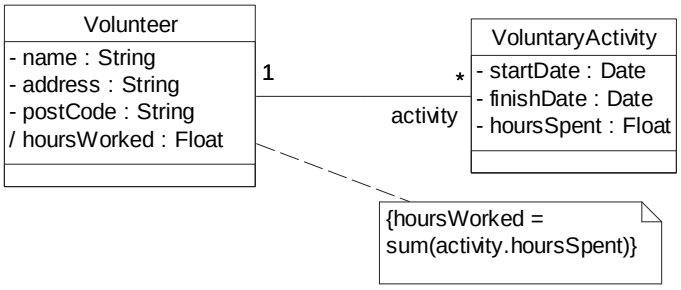
Hình 14.25: Ràng buộc {xor} trên kết hợp

Ràng buộc dùng để định nghĩa mối quan hệ giữa các phần tử mô hình cần phải thoả, nếu không hệ thống sẽ dừng. Các ràng buộc được hiển thị trong cặp dấu ngoặc, ví dụ {*subset*}. Một số ràng buộc được định nghĩa

trước trong UML như *từ khoá* hoặc các *phần tử chuẩn* (xem mục 14.2.9). Ví dụ, ràng buộc {*xor*} đuợc dùng trong biểu đồ lớp cho biết những thể hiện của lớp tham gia vào một trong hai kết hợp tuỳ chọn (không phải cả hai). Đây là một ràng buộc trên các kết hợp và được hiển thị bằng cách kết nối chúng lại với nhau bằng một đường thẳng đứt nét có nhãn là ràng buộc. Hình 14.25 là một ví dụ.

Ràng buộc trên lớp thường được hiển thị bên dưới biểu tượng lớp. Ràng buộc trên thuộc tính và thao tác được hiển thị dạng văn bản trong ngăn thích hợp của lớp chứa chúng. Những ràng buộc áp dụng cho một nhóm các thuộc tính hoặc thao tác có thể được trình bày như một điểm vào và áp dụng cho tất cả các phần tử kế tiếp cho đến khi gặp ràng buộc mới hoặc hết danh sách. Điều này ngụ ý các các thuộc tính hoặc thao tác không bị ràng buộc nên xuất hiện ở đầu danh sách. Khi các ràng buộc áp dụng cho hai biểu tượng trong một biểu đồ, các biểu tượng được nối với nhau bằng một đường thẳng đứt nét hoặc một mũi tên (nếu đường thẳng có hướng), và được đặt nhãn là ràng buộc. Khi có từ ba thành phần trở lên chịu ràng buộc, thì dùng một ghi chú gắn vào mỗi phần chịu ảnh hưởng bằng một đường thẳng đứt nét. Các ràng buộc thường được viết bằng OCL, xem chương 12.

Ràng buộc có thể được áp dụng cho các thuộc tính dẫn xuất để đảm bảo tính toàn vẹn dữ liệu. Ví dụ, hình 14.26 cho biết số giờ làm việc do *Volunteer* thực hiện phải bằng tổng số giờ mà người này dùng cho mỗi *VoluntaryActivity* (hoạt động tình nguyện). Hình này cũng trình bày cách sử dụng ghi chú để đặc tả ràng buộc.



Hình 14.26: Một ràng buộc trên thuộc tính dẫn xuất

Ràng buộc là cơ chế cuối cùng có thể được sử dụng để mở rộng ký hiệu UML. Có nhiều ràng buộc ngầm định trong các lớp khuôn dạng. Ví dụ, các lớp *stereotype* của UP trong hình 14.6 có các ràng buộc ngầm: các lớp *boundary* được ràng buộc để quản lý giao diện giữa hệ thống và người

dùng, trong khi các lớp *entity* được ràng buộc không tham gia vào giao diện người dùng.

Ràng buộc có thể được sử dụng cho các mô hình khác. Chẳng hạn như được dùng trong biểu đồ tuần tự để mô tả các ràng buộc giữa các biến cố.

Câu hỏi ôn tập

14.1 Hai phần tử của đồ thị là gì?

14.2 Hãy cho ba ví dụ về biểu đồ là đồ thị trong UML.

14.3 Ba loại quan hệ trực quan giữa các phần tử trong các đồ thị UML là gì?

14.4 (a) Những phần tử đồ thị trong UML là những phần tử nào?

(b) Giải thích ý nghĩa của mỗi phần tử trên.

(c) Ứng với mỗi phần tử hãy cho một ví dụ.

14.5 Hãy cho biết sự khác nhau giữa tên và nhãn trong UML.

14.6 Từ khoá được hiển thị trên biểu đồ UML như thế nào?

14.7 Cho hai ví dụ về từ khoá ở hai mô hình UML khác nhau.

14.8 Các chi chú được hiển thị như thế nào trên các biểu đồ UML?

14.9 Hãy định nghĩa *classifier*.

14.10 Cho 3 ví dụ về phần tử mô hình là các *classifier* trong UML.

14.11 Cho 3 ví dụ về phần tử mô hình không phải là *classifier* trong UML.

14.12 Ký hiệu của mỗi phần tử dưới đây là gì?

(a) Tên lớp.

(b) Tên của *thể hiện vô danh* của lớp.

(c) Tên của *thể hiện có tên* của lớp.

(d) Tên của *thể hiện có tên* mà không biết lớp.

(e) Tên của vai trò.

(f) Tên của *thể hiện có tên* trong một vai trò.

14.13 Hãy nêu định nghĩa về *stereotype* .

14.14 Cách sử dụng *stereotype* để mở rộng ký hiệu UML?

14.15 Mô tả ba cách dùng các lớp *stereotype* trong biểu đồ.

14.16 Cho một ví dụ về một lớp *stereotype*.

14.17 Cho một ví dụ về *stereotype* của loại phần tử mô hình khác.

14.18 *Tagged value* là gì?

14.19 Các *tagged value* được biểu diễn như thế nào trong biểu đồ?

14.20 Ràng buộc là gì?

14.21 Các ràng buộc được hiển thị như thế nào trong biểu đồ?

14.22 Cho một ví dụ về ràng buộc.

14.23 Tên của ngôn ngữ thường được dùng để biểu diễn ràng buộc là gì?

Bài tập có lời giải

14.1 Chúng ta muốn mở rộng UML để biểu diễn các ứng dụng dùng *giao thức truy cập không dây* WAP (Wireless Access Protocol) và *ngôn ngữ đánh dấu không dây* WML (Wireless Markup Language). Hai chuẩn này được dùng để truyền thông tin Internet cho các thiết bị không dây, như điện thoại di động và *Personal Digital Assistants* (PDAs). Say đây là tóm tắt cách WML làm việc.

WML dùng phép ẩn dụ *decks of cards*. Một ứng dụng được tạo từ một *deck* đơn. Mỗi *deck* gồm nhiều *card*; mỗi *card* hiển thị thông tin toàn màn hình trên màn hình của thiết bị di động. Toàn bộ *deck* được truyền vào thiết bị di động, sau đó người sử dụng di chuyển giữa các *card*. Điều này làm giảm khoảng thời gian chờ nạp các trang. Nội dung của *deck* được truyền vào thiết bị di động dưới dạng nhị phân để giảm thiểu băng thông (điện thoại GSM truyền dữ liệu với tốc độ 9.600 baud, đơn vị tốc độ điện báo).

Các lớp *stereotype* và các kết hợp nào cần tạo ra để mở rộng UML hầu bao phủ những phần tử WAP này?

Lời giải:

Đầu tiên là <<*Application*>>. Điều này dường như liên quan đến các trang phía *server* và *client*, nhưng chúng ta chỉ biết các trang phía *client*. Một *Application* <<*deliver*>> (chuyển) một <<*Deck*>> đến thiết bị di động và *Deck* gồm có các <<*Card*>>. Người sử dụng có thể di chuyển đến card kế tiếp (<<*next*>> Card) hoặc di chuyển trở lại card trước đó (<<*previous*>> Card).

Khi dùng các chuỗi khuôn dạng hiển thị cùng với tên lớp, chúng ta có thể phát triển một điều gì đó như hình 14.27 cho giả thuyết *deck of cards* có chứa nội dung tĩnh về *CarMatch* (bạn thử phát minh ra các icon thích hợp cho các stereotype ở đây xem sao).

Hình 14.27: Giải pháp khả thi cho WAP/WML

14.2 Những thiết bị di động khác nhau, điện thoại và PDAs, có một màn hình với kích thước và các kỹ thuật *điều hướng* khác nhau. Trình duyệt trên một thiết bị di động được gọi là *user-agent*, truyền thông tin về thiết bị đang chạy đến *server*, vì vậy những *card* có định dạng thích hợp được gởi đến thiết bị di động.

Chúng ta đã không thêm *user-agent* hoặc các thiết bị như là các lớp trong hình 14.27, nhưng các *tagged value* nào phải được gởi từ *client* đến *server* để bảo đảm rằng các trang *WML* có định dạng đúng đã được truyền đi?

Lời giải:

Có thể chúng ta cần đến hai *tagged value*, một cho *user-agent* và một cho thiết bị. Ở đây chúng ta giả sử rằng *server* giữ một cơ sở dữ liệu thiết bị cùng các đặc trưng của chúng. Hoặc chúng ta cần phải truyền đi những đặc điểm này. Sau đây là hai khả năng có thể:

{device="MyPDA Z99", useragent="Mobile Browser 1.21"}

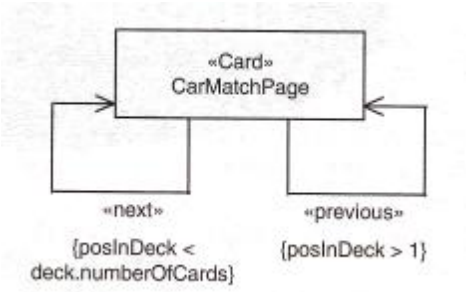
{width=640, height=240, colour=false, greyscale=true, shades=16, navigation="touchscreen"}

14.3 Người dùng duyệt qua tập các *card* trong *deck* từ đầu đến cuối. Người này sẽ di chuyển đến *card* kế tiếp cho đến khi *card* cuối cùng trong *deck* được hiển thị. Tương tự, người này sẽ duyệt lui trong deck trở về *card* đầu tiên.

Các ràng buộc nào được áp dụng cho kết hợp <<*next*>> và <<*previous*>> được định nghĩa ở trên?

Lời giải:

Ràng buộc hành vi là ràng buộc mà người dùng có thể di chuyển tới cho đến cuối hoặc di chuyển lùi cho đến đầu của *deck*. Khi di chuyển đến cuối, người sử dụng không được phép di chuyển tiếp (chẳng hạn như về đầu danh sách). Điều này có thể được biểu diễn bằng hai ràng buộc, một ràng buộc cho kết hợp <<*next*>> và một ràng buộc cho kết hợp <<*previous*>>. Xem hình 14.28.



Hình 1.28: Các ràng buộc có thể có trên các kết hợp

Bài tập bổ sung

14.4 Vẽ một lớp *PDA* theo khuôn dạng «*device*» và một lớp *WAP Explorer* theo khuôn dạng «*user-agent*».

14.5 Thiết kế biểu tượng riêng biểu diễn các lớp «*device*» và «*user-agent*».

14.6 Vẽ một biểu đồ lớp biểu diễn kết hợp «*run*» giữa lớp *PDA* và *WAP Explorer*.

14.7 Ứng dụng *WAP Explorer* chỉ chạy trên hệ điều hành *PDAOS*. Hãy thêm một ràng buộc vào kết hợp «*run*» trong lời giải ở trên để mô tả điều này.

14.8 Các *EIPs* (Enterprise Information Portals) đang trở nên phổ biến như là cách chuyển một loạt các kiểu thông tin khác nhau cho nhân viên của một tổ chức.

Điển hình là một ứng dụng *cổng* (portal) chạy trên server. Ứng dụng này phân phối các trang web đến client. Người sử dụng phải đăng nhập vào cổng, gởi username và password đến ứng dụng. Sau khi đăng nhập người này được cung cấp rất nhiều trang. Người sử dụng có thể di chuyển đến bất kỳ trang nào mà họ thích bằng cách nhắp chuột vào trang trong danh sách. Mỗi trang được đặt tạo từ các *khung* (pane). Khung là một cửa sổ nhỏ bên trong trang dùng để biểu diễn các loại thông tin khác nhau (ví dụ như lịch, bảng ghi chú, bộ máy tìm kiếm, bảng giá cổ phiếu). Nếu người dùng không tương tác với cổng trong một khoảng thời gian nào đó, thì người này bị đăng xuất, và phải đăng nhập lại nếu muốn thay đổi điều gì đó trên cổng.

Hãy phát thảo một vài lớp *stereotype* và vài kết hợp dùng cơ chế mở rộng của UML để mô hình các cổng.

14.9 Những *tagged value* nào cần thiết trong sự mở rộng?

14.10 Những ràng buộc nào có thể áp dụng cho kết hợp <<*moveTo*>> giữa các trang?

Chương 15
CÔNG CỤ CASE

15.1 Giới thiệu

Việc tạo ra mô hình phần mềm cho một hệ thống bất kỳ là một công việc mất nhiều thời gian, hầu như mọi người cần sử dụng các công cụ dựa trên máy tính, được biết như là các công cụ CASE. CASE là viết tắt của *Computer Assisted Software Engineering* (hoặc *Computer Aids Software Engineering* hoặc *Computer Aids System Engineering*). Các công cụ CASE tập trung vào mục đích là tự động hóa đến mức tối đa qui trình công nghệ phần mềm. *Pressman* (2000) mô tả đặc điểm một *workshop* (xưởng) tốt để tạo ra một sản phẩm bất kỳ, chẳng hạn như một chiếc xe hơi, một đồ dùng hoặc một chương trình máy tính, phải có ba phần: một *tập các công cụ* hỗ trợ việc xây dựng sản phẩm, một *bố cục có tổ chức* để cho phép các công cụ được áp dụng một cách hiệu quả và cuối cùng là một *đội ngũ nhân viên* lành nghề, những người biết cách dùng các công cụ. CASE cung cấp nhiều công cụ cho nhiều người khác nhau trong một qui trình công nghệ phần mềm. Thuật ngữ CASE rất phổ biến và chứa đựng phạm vi rộng lớn các hoạt động phát triển phần mềm, việc lập mô hình và việc dùng UML (hoặc các ký hiệu mô hình khác) chỉ là một phần của hoạt động này. Chương này sẽ trình bày các đặc điểm chính của các công cụ CASE trong việc lập mô hình dựa trên UML và cách chúng liên hệ với các công cụ khác được dùng trong việc phát triển của các hệ thống phần mềm quan trọng.

Mô hình hệ thống là một phần của công việc xây dựng hệ thống. Việc phát triển hệ thống máy tính là một công việc đòi hỏi nhiều người. Gần như mọi dự án đều yêu cần sự cộng tác giữa nhiều người, bao gồm *stakeholder* (nhà tài trợ và khách hàng đối với một hệ thống nào đó), người dùng tương lai, trưởng dự án, phân tích viên, lập trình viên, người thử chương trình, nhóm triển khai và những người dùng cuối. Một nhân tố cần thiết để phát triển hiệu quả là truyền thông tốt (*good communication*). Việc dùng một bộ công cụ CASE tích hợp dễ hiểu và nhất quán để hỗ trợ nhiều công việc khác nhau trong qui trình phát triển phần mềm là một yếu tố quan trọng của truyền thông tốt.

UML là chìa khoá cho việc truyền thông bên trong một dự án, cung cấp sự tích hợp giữa các công cụ CASE thông qua việc trao đổi các mô hình phần mềm và cung cấp một thể hiện chuẩn của các mô hình cho nhiều người có liên quan trong một dự án. Trong qui trình phát triển, có nhiều công cụ CASE liên quan, từ lúc bắt đầu dự án đến lúc chuyển giao dự án, bao gồm phân tích nghiệp vụ, phân tích yêu cầu, quản lý dự án, lập mô hình phần mềm, cài đặt, chạy thử chương trình và triển khai. Các công cụ CASE dùng để mô hình hóa phần mềm có thể tích hợp với nhau và với nhiều loại công cụ khác, chẳng hạn như các công cụ lập kế hoạch (*planning*) và kiểm tra (*testing*). Mục đích của chương này là sẽ tập trung vào các công cụ CASE có liên quan mật thiết với UML và nhấn mạnh vào việc phát sinh và quản lý các mô hình phần mềm.

Một công cụ CASE thích hợp có thể giúp bạn vẽ nhiều loại biểu đồ UML. Các công cụ khác nhau có mục tiêu khác nhau và kết quả đạt được cũng khác nhau, các công cụ này thường kết hợp với các ký hiệu bổ sung, không thuộc UML. Hiện nay, do sự phát triển của chuẩn UML, ngày càng có nhiều sự tương đồng trong cách trình bày và nội dung trong việc lập mô hình phần mềm của nhiều công cụ. Việc chuyển đổi giữa các công cụ mô hình hóa sẽ dễ dàng hơn nếu các công cụ tuân theo chuẩn UML. Một trong những mục tiêu quan trọng của UML là giảm thời gian lãng phí và công sức để chuyển đổi giữa các ký hiệu khi các nhà phát triển di chuyển qua lại giữa các dự án hoặc khi các dự án cần thay đổi công cụ.

Các công cụ CASE đóng góp vào việc phát triển mô hình theo nhiều cách khác nhau. Đầu tiên, các công cụ này hỗ trợ việc xây dựng chính xác các biểu đồ UML và cung cấp các kiểm tra để bảo đảm rằng các biểu đồ là hoàn chỉnh và nhất quán với nhau. Một công cụ CASE tốt sẽ tổ chức các *phần tử* (element) UML trong một nơi chứa, nơi này sẽ giữ các đối tượng, hoạt động, trạng thái, mối quan hệ, biểu đồ... được nhóm lại với nhau. Các loại biểu đồ khác nhau biểu diễn những góc nhìn khác nhau về nơi chứa bên dưới, là nơi lưu giữ các phần tử trong mô hình và sự liên kết của chúng. Qui trình phát triển thường tạo ra nhiều tài liệu khác hơn cả các biểu đồ UML (chẳng hạn như tài liệu mô tả use case), và một công cụ CASE có thể tích hợp các tài liệu này trong nơi chứa. Hầu như tất cả các dự án đều được thực hiện bởi một nhóm người cộng tác với nhau, khi đó nơi chứa cần có khả năng xử lý việc có nhiều người dùng chung các phần tử khác nhau và cho phép truy cập đến nơi chứa với một số phương tiện để ngăn chặn sự xung đột hoặc một số phương tiện để trộn và làm cho các nơi chứa tương thích với nhau. Kết quả cuối cùng của một dự án là một hệ thống và các công cụ CASE đưa ra nhiều loại phương tiện để phát

sinh mã chương trình hoặc *nhập* (import) mã chương trình, thậm chí là phát sinh các kế hoạch kiểm tra chương trình.

Ngày nay, không thể bắt đầu phát triển một dự án về tin học mà không dùng đến tập các công cụ dựa trên máy tính. Đương nhiên, không có nghĩa là cứ dùng các công cụ tin học là sẽ bảo đảm hiệu quả của việc phát triển, việc quản lý và tổ chức cũng quan trọng không kém. Jacobson (1992) đã trình bày về cách tổ chức ở các mức độ khác nhau. Ở mức ban đầu, không có các phương pháp sưu liệu nào là thích hợp và các nhà phát triển làm việc theo cách riêng của họ. Ở mức thứ hai, một phương pháp không hình thức được chấp nhận. Ở mức thứ ba, có một phương pháp hình thức thích hợp. Ở mức thứ tư, có một phương pháp theo dõi sát việc quản lý ứng dụng và hiệu suất của qui trình phát triển. Ở mức cao nhất, các thay đổi về ngữ nghĩa của phương pháp được thực hiện để tối ưu hóa phương pháp cho việc tổ chức. Một công cụ lập mô hình CASE có thể được dùng ở một mức bất kỳ nhưng có hiệu quả nhất ở mức thứ ba hoặc cao hơn. Cũng như trong các lãnh vực khác, công cụ tốt không có nghĩa là sẽ bảo đảm thành công nhưng chúng là yếu tố quan trọng để giúp con người tạo ra một sản phẩm tốt một cách hiệu quả.

15.2 UML và các công cụ CASE

UML có một số mục tiêu chính liên quan đến việc dùng các công cụ CASE. Đó là một tập ký hiệu cho phép đặc tả hệ thống độc lập với ngôn ngữ lập trình được chọn. UML được thiết kế để có thể mở rộng, vì thế các đặc điểm khác có thể được tích hợp vào các mô hình UML. Người ta dự tính khuyến khích sự lớn mạnh của thị trường công cụ đối tượng, và hỗ trợ các khái niệm phát triển ở mức cao hơn, chẳng hạn như *framework* và *pattern*.

Sự độc lập với ngôn ngữ lập trình là yếu tố cần thiết bởi vì các ngôn ngữ là *textual* và cần phải có một cách biểu diễn trực quan một hệ thống bởi vì các hệ thống thường được xây dựng từ các thành phần được cài đặt bằng các ngôn ngữ khác nhau. Tuy nhiên, có một sự tương ứng mạnh mẽ giữa các ngôn ngữ hướng đối tượng và UML. Một mô tả độc lập ngôn ngữ của hệ thống là điều quan trọng đối với *stakeholder* trong một dự án, trong khi việc liên kết giữa UML và ngôn ngữ là cực kỳ có giá trị trong giai đoạn xây dựng. Các công cụ CASE có thể hỗ trợ việc liên kết này thông qua việc phát sinh tự động mã chương trình và chuyển ngược lại từ mã chương trình sang cấu trúc mô hình UML.

Khả năng mở rộng là khía cạnh cốt lõi của UML, thông qua các *tagged value* và các *stereotype* (xem mục 14.3). Đây là điều quan trọng bởi vì UML không thể bao gồm tất cả mọi mặt của việc mô hình mà vẫn giữ

được sự ngắn gọn đủ quản lý. Các công cụ CASE cần hỗ trợ khả năng mở rộng và cũng tuỳ thuộc điều này trong việc thực hiện nhiều công việc khác nhau trong qui trình phát triển phần mềm, chẳng hạn như nhúng tài liệu, tích hợp mã chương trình vào mô hình và liên kết với các công cụ khác.

UML nâng cao thị trường công cụ đối tượng bằng cách thay đổi phạm vi cạnh tranh ra khỏi ký hiệu. Các công cụ CASE sẽ thêm giá trị cho qui trình phát triển phần mềm, bằng cách tuân theo các phương pháp và bằng cách tích hợp với các công cụ khác và với các ngôn ngữ lập trình, UML dựa trên CASE sẽ đầy đủ hơn là chỉ dựa vào các ký hiệu của chính nó. Việc dùng UML làm cho người phát triển dễ di chuyển giữa các đề án bằng cách dùng các công cụ khác nhau, và di chuyển giữa các công cụ khác nhau trong cùng một dự án.

15.3 Các đặc trưng của công cụ CASE

15.3.1 Các biểu đồ UML

Một công cụ CASE tốt sẽ cung cấp phương tiện để vẽ hầu hết các biểu đồ UML, tất nhiên cũng có cả các biểu đồ cốt lõi, chẳng hạn như biểu đồ use case, biểu đồ tuần tự, biểu đồ trạng thái và biểu đồ lớp. Một công cụ CASE tổng quát sẽ chỉ cho phép các biểu đồ hợp lệ, và cũng có thể thực hiện một số kiểm tra về tính nhất quán nhằm ngăn chận các chu trình thừa kế hoặc làm nổi bật các tên xung đột. Điều này rất quan trọng khi có nhiều người làm việc trên cùng một hệ thống và tính nhất quán giữa việc phát triển mô hình song song là điều cần thiết.

Việc xây dựng các biểu đồ thường là một hoạt động vẽ, thông tin chi tiết của các phần tử được được bổ sung vào thông qua các biểu mẫu. Ví dụ, một công cụ chuẩn cho phép vẽ một lớp bằng cách nhấp chuột trên một biểu tượng lớp và đặt biểu tượng đó lên nền cần vẽ. Để hoàn tất các chi tiết, chẳng hạn như các thuộc tính, ta phải gõ tên của thuộc tính vào một biểu mẫu và chọn kiểu cho thuộc tính từ một danh sách. Các phần tử được xây dựng trước có thể được đưa vào biểu đồ bằng cách kéo và thả nó từ một nơi chứa nào đó hoặc chọn nó từ các danh sách. Các biểu đồ và nơi chứa là các khung nhìn bổ sung của các phần tử UML chẳng hạn như các lớp và use case, và các công cụ CASE có thể cung cấp nhiều cách khác nhau để quản lý các phần tử này.

UML nổi lên từ một số các ký hiệu trước đây, và nhiều công cụ CASE đã phát triển từ những ký hiệu này. Ở thời điểm viết cuốn sách này, nhiều công cụ vẫn còn dùng các phần tử ký hiệu cũ (như Booch hoặc OMT). Một phần là để hỗ trợ các dự án bắt đầu trước khi UML được định nghĩa hoặc

được chấp nhận rộng rãi; một phần vì việc xây dựng các công cụ CASE là một công việc rất lớn và những nhà cung cấp các gói này vẫn đang trong quá trình nắm bắt. Cũng có một số ký hiệu khác, như mô hình qui trình nghiệp vụ, mà UML không hỗ trợ đầy đủ, được kết hợp chặc chẽ bên cạnh UML trong một công cụ CASE.

15.3.2 Theo chuẩn UML

UML là một chuẩn, và các công cụ CASE phải xác định được phạm vi của chuẩn này để tuân theo. Đặc tả UML cung cấp nhiều điểm phải theo, bao gồm các *biểu đồ* (use case, lớp, trạng thái, hoạt động, tuần tự, cộng tác, thành phần và triển khai), các *profile* (qui trình phát triển phần mềm và mô hình hóa nghiệp vụ), và *OCL*. Mỗi điểm này có thể được đánh giá là '*no/incomplete*' hoặc '*complete*' (không đầy đủ hoặc đầy đủ). Có một số phần tử khác từ *metamodel* của UML mà chúng ta không xem xét ở đây, các phần tử này bảo đảm ngữ nghĩa của các phần tử mô hình được dùng để tạo nên biểu đồ là đúng chuẩn. Sự tương thích với một biểu đồ còn bao gồm khả năng hỗ trợ tất cả các chi tiết phụ khác (như *stereotype* , *tagged value* và *constraint*).

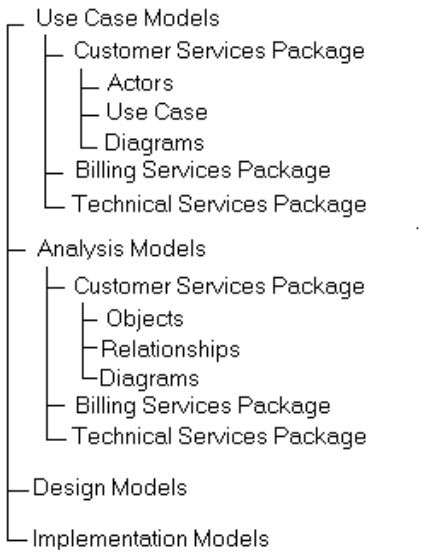
Để khai báo chính xác một công cụ CASE tuân theo đặc tả UML, người cung cấp công cụ CASE nên xem xét đến các phần tử ngôn ngữ và các điểm tương thích, thay vì đơn giản nói chúng đều tương thích. Các công cụ CASE không cần tuân theo toàn bộ đặc tả UML, nhưng nếu nó chỉ tương thích một phần sẽ làm giảm đáng kể khả năng hoán đổi giữa các mô hình. Vì vậy, nên cẩn thận khi chọn công cụ CASE, và cần xem xét kỹ lưỡng để bảo đảm rằng ứng dụng có khả năng độc lập với công cụ để tránh dự án mắc phải những vấn đề phát sinh từ nhà cung cấp công cụ.

15.3.3 Repository (nơi chứa)

Nơi chứa trong một công cụ CASE sẽ giữ tất cả các phần tử UML, như các đối tượng, thuộc tính, thao tác, hoạt động, trạng thái và các quan hệ. Các biểu đồ UML thực ra chỉ là một *view* (khung nhìn) về nơi chứa. Việc tạo ra nhiều *view* khác nhau về một nơi chứa với một phần tử (hoặc một tập các phần tử) được lặp lại trong nhiều biểu đồ khác nhau là điều khá phổ biến. Ví dụ, chúng ta thường xây dựng một biểu đồ lớp cho một use case để mô tả sự cộng tác của các đối tượng trong use case, và khi một lớp hỗ trợ cho nhiều use case, lớp này sẽ có mặt trong nhiều biểu đồ khác nhau.

Nơi chứa thường được tổ chức theo dạng cây phân cấp, giống như cách tổ chức tập tin trong một hệ điều hành, và các trình duyệt được cung cấp để xem cây phân cấp này. Hình 15.1 minh họa một cách tổ chức nơi chứa.

Nơi chứa được chia thành các mô hình yêu cầu, phân tích, thiết kế và cài đặt. Mỗi mô hình ở mức cao sau đó lại được chia thành các gói và bên dưới các gói là các phần tử UML và các biểu đồ tạo nên mô hình. Nơi chứa rất quan trọng trong việc chứa đựng hàng trăm, thậm chí hàng ngàn các phần tử của một dự án, và khung nhìn về nơi chứa sẽ cho phép mở và đóng các phần của cây phân cấp để phóng to và thu nhỏ các mặt khác nhau của dự án. Khái niệm gói của UML đặc biệt quan trọng trong việc xây dựng khung nhìn về nơi chứa.



Hình 15.1: Cấu trúc nơi chứa thông thường trong một công cụ CASE

Nơi chứa trong một số công cụ CASE cũng có thể chứa các phần tử không thuộc UML. Thông thường, các tài liệu khác chẳng hạn như các tài liệu Word có thể được liên kết vào nơi chứa và được truy cập thông qua nơi chứa. Một số công cụ CASE cũng có thể kết hợp các liên kết từ nơi chứa vào mã nguồn của ngôn ngữ lập trình.

15.3.4 Truy cập nơi chứa dùng chung

Đối với những dự án bình thường, việc phát triển các mô hình UML sẽ đòi hỏi nhiều người phát triển các mô hình đồng thời. Có hai cách xử lý điều này. Cách thứ nhất là cho phép mọi người có các bản sao của nơi chứa và sau khi thực hiện công việc xong, trộn các nơi chứa lại và kiểm tra tính nhất quán của nó. Cách tiếp cận thứ hai tốt hơn, đó là cung cấp một nơi chứa dùng chung để các nhà phân tích và phát triển kiểm tra

các phần nơi chứa mà họ đang làm việc. Nếu một nơi chứa được dùng, sau đó các thành viên trong nhóm sẽ được cấp phép để kiểm tra các phần tử của nơi chứa. Trong khi các phần tử này đang được kiểm tra, thì các thành viên khác trong nhóm không thể cập nhật mà chỉ có thể xem chúng. Một khi một người đã kiểm tra xong một tập các phần tử, người đó sẽ được phép trộn chúng trở lại nơi chứa, chép chồng lên các phiên bản cũ và giải phóng chúng để cho người khác có thể xem và cập nhật.

15.3.5 Tính toàn vẹn của nơi chứa

Một vai trò chính của nơi chứa là làm hợp lệ các đầu vào và bảo trì tính toàn vẹn. Các thay đổi trên một phần tử trong mô hình sẽ ảnh hưởng đến nhiều phần tử của các mô hình khác. Ví dụ, việc xóa một thao tác ra khỏi một lớp sẽ làm ảnh hưởng đến các cộng tác có liên quan được biểu diễn bởi một biểu đồ tuần tự. Một công cụ CASE sẽ có nhiều cách để bảo đảm tính nhất quán và toàn vẹn, tránh các thay đổi không hợp lệ do người dùng gây ra thông qua các cảnh báo.

15.3.6 Kiểm soát phiên bản

Trong các giai đoạn quan trọng của dự án, sẽ có các phiên bản của mô hình phần mềm cần được lưu trữ lại như là phiên bản mốc nhằm biểu diễn trạng thái hiện hành của dự án. Phiên bản này có thể được dùng cho việc phát hành sản phẩm, hoặc làm một bản sao mô hình dự phòng để có thể quay lại dự án nếu có vấn đề trong việc phát triển vì một lý do nào đó. Để thực hiện điều này, các công cụ CASE cần hỗ trợ việc kiểm soát phiên bản một cách trực tiếp hoặc cần tích hợp với một gói kiểm soát phiên bản để quản lý chuyện này. Việc kiểm soát phiên bản sẽ tiếp tục trong suốt thời gian sống của sản phẩm phần mềm.

15.3.7 Quản lý khả năng theo dõi và thay đổi

Trong suốt chu trình sống của bất kỳ một dự án nào, các thay đổi có thể xảy ra để đáp ứng yêu cầu. Thay đổi có thể xảy ra trước khi chuyển giao sản phẩm, và cũng xảy ra thường xuyên trong thời gian sống của sản phẩm (mỗi khi cần cải tiến hoặc điều chỉnh sản phẩm lại cho phù hợp để hỗ trợ nghiệp vụ). Các thay đổi đến đặc tả sản phẩm rất có thể sẽ ảnh hưởng đến mô hình.

Khi có yêu cầu thay đổi, cần tìm ra tất cả những khía cạnh trong hệ thống bị ảnh hưởng bởi thay đổi và bảo vệ chúng khỏi bị ảnh hưởng. Một phần lớn các yêu cầu có thể được giữ bên ngoài một công cụ CASE, ví dụ như trong các tài liệu văn bản. Ý niệm của use case là kết nối giữa các mô hình phân tích và yêu cầu của người dùng. Việc nhóm các lớp qua các

cộng tác để hỗ trợ một use case làm nó có thể lần theo yêu cầu đến use case hỗ trợ yêu cầu này. Từ use case có thể lần theo các lớp hỗ trợ thiết kế để hiện thực use case, và sau cùng là các lớp cài đặt thiết kế.

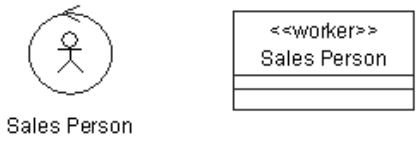
Các công cụ CASE giúp đỡ điều này bằng cách tích hợp với các yêu cầu lấy được và các công cụ lập trình và bằng cách xây dựng cấu trúc nơi chứa theo các cách hỗ trợ khả năng theo dõi dự án (bằng cách nhóm các biểu đồ cộng tác với những use case mà chúng hỗ trợ). Vấn đề khó khăn trong việc theo dõi là sự rộng lớn của dự án, không chỉ liên quan đến các mô hình, mà còn vì các yêu cầu áp dụng phương pháp một cách cẩn trọng, như UP. Các công cụ CASE và UML là các tiện nghi trong qui trình này.

15.3.8 Các giá trị thẻ (tagged value)

UML hỗ trợ các giá trị thẻ, chúng có thể được gắn vào bất kỳ phần tử mô hình nào (xem phần 14.3.2). Đây là môt kỹ thuật đơn giản để đính kèm thông tin nằm ngoài các ngữ nghĩa cơ bản của UML. Thông thường các giá trị này là một cặp và được biểu diễn dưới dạng {*tag* = *value*}, trong đó *tag* là tên thuộc tính và *value* là một giá trị có thể được biểu diễn dưới dạng chuỗi. Các công cụ CASE dùng các giá trị thẻ với nhiều mục đích khác nhau. Thông thường có một thẻ dành cho việc lập tài liệu cho phần tử trong mô hình, cho phép chứa một mô tả về lớp và use case. Một số công cụ CASE hỗ trợ các thẻ để ghi nhận các đường đi chính và đường đi phụ trong các use case, và nhúng các mô tả về use case vào nơi chứa bằng cách này. Các giá trị thẻ đôi khi được dùng để xây dựng cấu trúc các phần tử (chẳng hạn như các use case và các thao tác) bằng cách cho phép gắn kèm các *pre-condition* và *post-condition*. Các liên kết đến mã nguồn có thể được nhúng vào để cung cấp sự tích hợp chặt chẽ hơn giữa công cụ CASE mô hình hóa và môi trường lập trình.

15.3.9 Biểu tượng (icon)

Chúng ta đã thấy một số biểu tượng chuẩn được UML định nghĩa, chẳng hạn biểu tượng *actor*. UML cho phép đính kèm biểu tượng vào một *stereotype*. Tất cả các đối tượng với khuôn dạng sẽ được hiển thị tuỳ chọn như một biểu tượng. Hình 15.2 là khuôn dạng <<*worker*>> từ *profile* lập mô hình nghiệp vụ được hiển thị bằng hai cách: ký hiệu biểu tượng, và ký hiệu lớp chuẩn với tên khuôn dạng trong cặp dấu ngoặc nhọn. Các công cụ CASE cho phép thêm các mở rộng như thế này để cung cấp các mô tả trực quan có cải tiến của các mô hình.



Hình 15.2: Khuôn dạng <<worker>> từ profile *lập mô hình nghiệp vụ dưới dạng biểu tượng và ký hiệu chuẩn*

15.3.10 Phát sinh mã

Để nâng cao tốc độ và tính chính xác trong việc lập trình, nhiều công cụ CASE đã đưa ra các tiện ích phát sinh mã cho các chương trình và các lược đồ cơ sở dữ liệu. Ở mức đơn giản nhất là phát sinh các lớp theo cú pháp của ngôn ngữ lập trình, sau đó lập trình viên tự hoàn tất, và các định nghĩa của các bảng cơ sở dữ liệu. Một số công cụ CASE cũng hỗ trợ phát sinh mã logic lừ các mô hình (chẳng hạn như biểu đồ tuần tự). Hiện nay mức độ tích hợp giữa các công cụ CASE và các môi trường lập trình đã có những thay đổi lớn lao. Ở mức cao nhất, công cụ CASE có thể được xem như một môi trường lập trình. Còn đối với một số mức độ khác, công cụ CASE chỉ là một cách hỗ trợ viết mã.

Các công cụ CASE cũng có thể *import* chương trình và lược đồ cơ sở dữ liệu, từ đó xây dựng một nơi chứa. Số lượng thông tin được *import* không cố định, tuỳ thuộc vào công cụ CASE và ngôn ngữ lập trình. Tuy nhiên, tối thiểu công cụ CASE có thể phát sinh được một tập các lớp có định nghĩa các thao tác và thuộc tính.

Hầu hết các công cụ CASE đều đưa ra ý niệm *round trip engineering*. Nghĩa là mã chương trình được phát sinh từ công cụ CASE và được thay đổi trong môi trường lập trình. Sau đó chương trình đã được thay đổi này có thể được *import* và các thay đổi sẽ được phản ánh lại trong nơi chứa của công cụ CASE.

15.3.11 Khả năng chuyển đổi giữa CASE và các công cụ khác

Các mô hình UML có thể được mô tả bằng một ngôn ngữ văn bản chuẩn, đó là XMI, viết tắt của *XML Metadata Interchange.* Còn XML là viết tắt của *Extensible Markup Language*, đây là một chuẩn quốc tế trong việc mô tả tài liệu. Các công cụ CASE thường có tiện ích để tạo ra các phiên bản XMI cho các mô hình mà nó chứa và để import các mô hình XMI.

Điều này có nghĩa là những công cụ CASE có hỗ trợ XMI có thể hoán chuyển các mô hình của chúng với nhau. Có nhiều lý do để thực hiện điều này. Một số công cụ CASE được tích hợp chặt chẽ với một môi trường

phát triển đặc biệt, và XMI cho phép *import* các mô hình UML đã được phát triển ở một nơi khác. Ngày càng có nhiều các hệ thống lớn được xây dựng bằng cách tích hợp các hệ thống con được phát triển bởi nhiều ngôn ngữ khác nhau và bởi những nhà cung cấp khác nhau. Do đó việc truy cập đến nhiều loại mô hình trong các hệ thống con này là cần thiết. XMI cho phép lựa chọn công cụ tốt nhất và có chi phí chuyển đổi nhỏ nhất để thực hiện một công việc cụ thể.

15.3.12 Quan hệ giữa CASE và các công cụ khác

Các công cụ CASE chỉ hỗ trợ một phần nào đó trong một dự án và UML được thiết kế để bao hàm những khía cạnh về lập mô hình hệ thống. Các công cụ CASE thường liên kết với các công cụ khác. Ở mức đơn giản nhất, các tài liệu văn bản có thể được liên kết vào mô hình để hỗ trợ việc nắm bắt yêu cầu và ghi nhận các khía cạnh của việc lập mô hình không được hỗ trợ bởi công cụ CASE, chẳng hạn như mô tả nguyên văn nghiệp vụ. Các công cụ phân tích yêu cầu hiện có có thể được liên kết vào các công cụ CASE để cung cấp khả năng theo dõi giữa các phần tử trong mô hình và các phát biểu yêu cầu. các công cụ CASE thường được liên kết với các công cụ phát triển chẳng hạn như *Interactive Development Enviroments* (IDEs, môi trường phát triển tích hợp), là những môi trường hỗ trợ việc phát triển mã thi hành. Chúng cũng có thể liên kết tới các công cụ kiểm tra để phát sinh, theo dõi và thi hành các kế hoạch chạy thử chương trình.

15.3.13 Hỗ trợ và tôn trọng phương pháp

UML là một ký hiệu, không phải là một phương pháp. Hầu hết các công cụ đều có mục tiêu là hỗ trợ một phương pháp phát triển phần mềm. Phương pháp này có thể dựa trên UP hoặc một phương pháp khác. Việc hỗ trợ phương pháp được cung cấp theo nhiều cách khác nhau. Ở mức độ cơ bản nhất, công cụ sẽ kết hợp hướng dẫn trong dạng tài liệu giúp đỡ và hỗ trợ. Các nơi chứa có thể được cấu trúc theo các quy tắc của phương pháp (một số công cụ cho phép bạn sắp xếp lại cấu trúc của nơi chứa để phù hợp với các phương pháp khác nhau). Trong một số trường hợp công cụ CASE bắt buộc phải tuân theo các quy tắc của một phương pháp nào đó.

15.3.14 Framework (khung làm việc)

Một *framework* là một thiết kế có thể tái sử dụng và được dùng để hỗ trợ và cải tiến việc phát triển hệ thống và để cung cấp một cấu trúc chung cho các hệ thống được phát triển bởi một tổ chức. Bên cạnh việc thiết kế các *pattern* (mẫu), *framework* còn giúp cho việc chuẩn hóa và tối ưu hóa

các cách tiếp cận trong phát triển phần mềm. Có nhiều loại *framework* khác nhau. Các *framework* kỹ thuật như các lớp trong *Java Development Kit*, là nơi cung cấp các lớp và các thành phần được định nghĩa trước cho nhiều công việc lập trình khác nhau. Cũng có các *framework* ứng dụng như được phát triển bởi dự án IBM San Francisco, mà trọng tâm là phát triển các lớp tổng quát và các thành phần cho các hệ thống nghiệp vụ (chẳng hạn hệ thống kế toán). Cũng có các *framework* hướng đối tượng cấu trúc nơi chứa theo phương pháp phát triển chẳng hạn như UP. Một số công cụ CASE cho phép kết hợp và định nghĩa *framework*, được mô tả như một tập các mô hình được định nghĩa trước sẵn sàng khi bắt đầu một dự án.

15.3.15 OCL

Hiện nay, OCL thường được tích hợp vào các công cụ CASE để định nghĩa ràng buộc. Tuy nhiên, chưa thể tích hợp đầy đủ với việc phát sinh mã kiểm tra các *pre-condition*. Các công cụ CASE thường cho phép biểu diễn ràng buộc dưới dạng nguyên văn ban đầu, như được mô tả trong đặc tả UML.

15.3.16 Phát sinh tài liệu

Các mô hình UML có giá trị đối với các *stakeholder* trong dự án. Ví dụ, người dùng cuối muốn xem mô hình use case trong giai đoạn nắm bắt yêu cầu. Các công cụ CASE hỗ trợ việc tạo các tài liệu trên giấy, trộn các biểu đồ UML với văn bản mô tả các phần tử trong mô hình. Một số công cụ cho phép phát sinh các khung nhìn dựa trên web về nơi chứa của chúng để mọi người có thể truy cập đến các mô hình một cách rộng rãi thông qua web mà không cần truy cập vào công cụ CASE.

Câu hỏi ôn tập

15.1 CASE dựa trên cụm từ tiếng Anh nào?

15.2 Có phải CASE chỉ dành cho UML không?

15.3 UML hỗ trợ các công cụ CASE theo cách nào?

15.4 XMI là gì?

15.5 Một nơi chứa trong một công cụ CASE là gì?

15.6 Giá trị thẻ là gì?

15.7 Tại sao các giá trị thẻ lại quan trọng?

15.8 Ý niệm *round trip engineering* là gì?

Bài tập có lời giải

15.1 Trưởng dự án của hệ thống CarMatch muốn chọn một bộ công cụ CASE phù hợp, bao gồm các công cụ lập mô hình có hỗ trợ UML. Ông ta có hai tháng để chọn công cụ. Biết rằng ngoài thị trường không chắc chắn có công cụ hoàn hảo, ông ủy nhiệm cho một nhà cố vấn để thi hành một qui trình để cung cấp các công cụ CASE. Vậy những câu hỏi chính mà nhà cố vấn đưa vào tài liệu đề cập đến công cụ lập mô hình phần mềm?

Lời giải:

Nhà cố vấn cung cấp một số thông tin yêu cầu chung về giá cả, nền kỹ thuật và yêu cầu, ghi nhận thông tin về nhà cung cấp, khả năng thâm nhập thị trường, doanh thu và lợi nhuận, cộng thêm yêu cầu về địa điểm. Sau đó người này thêm các câu hỏi liên quan đến các công cụ CASE như sau:

1. Công cụ tuân theo chuẩn UML như thế nào? Tỉ lệ tương thích của công cụ với các biểu đồ: use case, lớp, trạng thái, hoạt động, tuần tự, cộng tác, thành phần, và triển khai.
2. Việc truy cập đến nơi chứa được quản lý như thế nào?
3. Các nơi chứa được trộn chung như thế nào?
4. Tính toàn vẹn của nơi chứa được quản lý như thế nào?
5. Kiểm soát phiên bản của nội dung nơi chứa được quản lý như thế nào?
6. Công cụ có thể tích hợp với các công cụ CASE khác nào?
7. Việc theo dõi các yêu cầu trong qui trình cài đặt được hỗ trợ như thế nào?
8. Ký hiệu nào khác với UML được công cụ này hỗ trợ?
9. Công cụ hỗ trợ phát sinh mã cho những ngôn ngữ nào? Có khả năng đảo ngược kỹ thuật không?
10. XMI có được hỗ trợ không?
11. Công cụ hỗ trợ những phương pháp nào?
12. Có hỗ trợ việc phát sinh tài liệu không? Có bao gồm các tài liệu dựa trên web không?

Bài tập bổ sung

15.2 Trưởng dự án *VolBank* không được cung cấp ngân sách đủ để mua các công cụ CASE hỗ trợ cho việc phát triển hệ thống. Hãy cung cấp một trang ghi nhớ để thuyết phục việc dùng công cụ CASE hỗ trợ cho việc lập mô hình phần mềm. Hãy nêu rõ các lợi ích cửa việc dùng một công cụ CASE lập mô hình phần mềm có hỗ trợ UML, và các rủi ro có thể xảy ra nếu không dùng một công cụ CASE.

Chương 16
MẪU THIẾT KẾ

16.1 Giới thiệu

Mẫu (pattern) không phải là một phần của UML cốt lõi, nhưng nó lại được sử dụng rộng rãi trong thiết kế hệ thống hướng đối tượng, và UML cung cấp các cơ chế để biểu diễn mẫu dưới dạng đồ họa. Với hai lý do trên, chúng tôi tạo ra chương ngắn này để trình bày về các mẫu thiết kế.

Một mẫu thiết kế là một giải pháp cho vấn đề phổ biến trong việc thiết kế các hệ thống máy tính. Đây là một giải pháp đã được công nhận là tài liệu có giá trị, những người phát triển có thể áp dụng giải pháp này để giải quyết các vấn đề tương tự. Giống với các yêu cầu của thiết kế và phân tích hướng đối tượng (nhằm đạt được khả năng tái sử dụng các thành phần và thư viện lớp), việc sử dụng các mẫu cũng cần phải đạt được khả năng tái sử dụng các giải pháp chuẩn đối với các vấn đề thường xuyên xảy ra.

Trong chương 8 chúng ta đã có một ví dụ về *mẫu* (xem hình 8.12). Mẫu *Façade* được sử dụng rộng rãi trong các hệ thống khi người phát triển muốn che dấu đi sự phức tạp của hệ thống con đằng sau một lớp, chỉ cung cấp giao diện với các chức năng của lớp. Đây là một vấn đề phổ biến trong thiết kế hệ thống. Nếu một hệ thống con có nhiều lớp cộng tác với nhau nhằm cung cấp các dịch vụ của nó, sẽ có rủi ro trong việc tạo ra một giao diện phức tạp đến hệ thống đó. Mỗi lớp có thể có nhiều thao tác, và các hệ thống con khác phải có khả năng gởi thông điệp cho thể hiện của mỗi lớp này. Điều này sẽ tạo ra một *coupling* (móc nối) giữa các hệ thống con: các thay đổi đến các lớp trong hệ thống con đang cung cấp các dịch vụ sẽ làm ảnh hưởng đến tất cả các hệ thống con có gọi đến các thao tác của lớp này.

Cách làm đơn giản tình huống này là tạo ra một lớp *façade*, lớp này cung cấp một giao diện đơn vào trong hệ thống con và phối hợp các hành động của các thể hiện trong hệ thống con này. Theo cách này, các thay đổi đến cài đặt của các lớp trong hệ thống con sẽ chỉ ảnh hưởng ở một mức độ nào đó đối với các hệ thống con khác: có thể giới hạn các thay đổi đối với chính hệ thống con đó, bởi vì các thông điệp đến các lớp đã thay đổi (bên trong hệ thống con này) sẽ đến từ lớp *façade*. Việc dò tìm ảnh hưởng của

các thay đổi cũng được thực hiện dễ dàng hơn, bởi chỉ cần tìm ra các điểm (trong các hệ thống con khác) nơi thông điệp được gởi đến các thể hiện của lớp *façade* – tức là chỉ một lớp thay vì nhiều lớp như trước đây.

Mẫu *Façade* chỉ là một trong nhiều mẫu đã được sưu liệu bởi các nhà phát triển có kinh nghiệm. Trước khi đưa ra các ví dụ chi tiết hơn và giải thích cách biểu diễn các mẫu trong UML, chúng ta hãy tìm hiểu một chút về nguồn gốc của chúng.

16.2 Nguồn gốc của mẫu

Ý tưởng dùng mẫu xuất phát từ ngành kiến trúc. *Alexander*, *Ishikawa*, *Silverstein*, *Jacobson*, *Fiksdahl-King* & *Angel* (1977) lần đầu tiên đề ra ý tưởng dùng các mẫu chuẩn trong thiết kế xây dựng và truyền thông. Họ đã xác định và lập sưu liệu các mẫu có liên quan để có thể dùng để giải quyết các vấn đề thường xảy ra trong thiết kế các cao ốc. Mỗi mẫu này là một cách thiết kế, chúng đã được phát triển hàng trăm năm như là các giải pháp cho các vấn đề mà người làm trong lĩnh vực xây dựng thường gặp. Các giải pháp tốt nhất có được ngày nay là qua một quá trình sàng lọc tự nhiên. Mặc dù ngành phát triển phần mềm không có lịch sử giống với ngành kiến trúc, nhưng mẫu được xem là giải pháp tốt để giải quyết vấn đề xây dựng hệ thống phần mềm.

Evitts (2000) có tổng kết về cách mẫu thâm nhập vào thế giới phát trển phần mềm (sách của ông nói về những mẫu có thể được sử dụng trong UML chứ không phải những mẫu tổng quát). Ông ta công nhận *Kent Beck* và *Ward Cunningham* là những người phát triển những mẫu đầu tiên với *Smalltalk* trong công việc của họ được báo cáo tại hội nghị *OOPSLA'87*. Có 5 mẫu mà *Kent Beck* và *Ward Cunningham* tìm ra trong việc kết hợp với các người dùng của một hệ thống mà họ đang thiết kế. Năm mẫu này đều được áp dụng để thiết kế giao diện người dùng trong môi trường Windows.

Suốt những năm đầu 1990, mẫu được thảo luận ở các hội thảo *workshop*, sau đó người ta nỗ lực thảo ra danh sách các mẫu và lập sưu liệu về chúng. Những người tham gia bị dồn vào việc cần thiết phải cung cấp một số kiểu cấu trúc ở một mức quan niệm cao hơn đối tượng và lớp để cấu trúc này có thể được dùng để tổ chức các lớp. Đây là kết quả của sự nhận thức được rằng việc dùng các kỹ thuật hướng đối tượng độc lập sẽ không mang lại những cải tiến đáng kể đối với của chất lượng cũng như hiệu quả của công việc phát triển phần mềm. Mẫu được xem là cách tổ chức việc phát triển hướng đối tượng, cách đóng gói các kinh nghiệm của những người đi trước và rất hiệu quả trong thực hành.

Năm 1994 hội nghị *PLoP* (Pattern Language of Program Design) đã được tổ chức. Cũng trong năm này quyển sách *Design Patterns: Element of Reaseable Object-Oriented Software* (Gamma, Johnson, Helm & Vhissdes, 1995) đã được xuất bản đúng vào thời điểm diễn ra hội nghị OOPSLA'94. Đây là một tài liệu còn phôi thai trong việc làm nổi bật ảnh hưởng của mẫu đối với việc phát triển phần mềm; sự đóng góp của nó là xây dựng các mẫu thành *catalogue* (danh mục) với định dạng chuẩn được dùng làm tài liệu cho mỗi mẫu. Nó nổi tiếng với tên *Gang of Four* (bộ tứ), và các mẫu của nó thường được gọi là các mẫu *Gang of Four*. Các cuốn sách khác xuất hiện trong hai năm sau, và các định dạng chuẩn khác được đưa ra.

Các mẫu đầu tiên là các mẫu thiết kế độc quyền, dự định giải quyết các vấn đề mà người thiết kế thường gặp. Từ đó các loại mẫu khác đã được phát triển nhanh, đáng lưu ý là các mẫu phân tích, biểu diễn các khái niệm được dùng trong việc lập mô hình thuộc phạm vi bài toán; và các mẫu tổ chức, đề xuất những giải pháp cho các vấn đề chung trong việc quản lý các qui trình nghiệp vụ, bao gồm cả việc phát triển phần mềm.

Framework và *idiom* có liên quan với mẫu, nhưng chúng ta cần phải phân biệt chúng. *Framework* thì tổng quát hơn và có thể áp dụng cho một *lĩnh vực* cụ thể. Ví dụ, một framework tài chính sẽ chứa một tập các lớp về tài chính trong các mối quan hệ được xác định bởi các mẫu, *framework* này có thể được phát triển để tạo ra các ứng dụng tài chính. (Trong UML, *framework* có một ý nghĩa riêng và được định nghĩa hoặc như các gói *stereotype* chứa mẫu chính hoặc như mẫu kiến trúc cung cấp một mẫu có thể mở rộng cho các ứng dụng trong một lĩnh vực riêng). Một *idiom* là một tập tác chỉ dẫn về cách cài đặt các khía cạnh của một hệ thống phần mềm viết bằng một ngôn ngữ cụ thể. *Coplien* (1992) lần đầu tiên đã xuất bản một tập các *idiom* cho việc dùng ngôn ngữ C++. Các *idiom* này ghi lại các kinh nghiệm của các lập trình viên chuyên nghiệp C++, để từ đó các lập trình viên không chuyên có thể giải quyết các vấn đề thường gặp khi viết chương trình bằng C++.

16.3 Sưu liệu cho mẫu

Mẫu được phân loại thành hai nhóm: *pattern catalogue* (danh mục mẫu) và *pattern language* (ngôn ngữ mẫu). Một *pattern catalogue* là một nhóm mẫu có quan hệ với nhau có thể được sử dụng cùng nhau hoặc độc lập. Một *pattern language* sẽ lập sưu liệu cho các mẫu làm cùng nhau và có thể được áp dụng để giải quyết các vấn đề trong một lĩnh vực nào đó.

Các mẫu được lập sưu liệu bằng cách dùng các *template*, các *template* cung cấp các *heading* bên dưới chứa chi tiết của mẫu và cách thức nó làm

việc cho phép người dùng biết mẫu đó có thích hợp với vấn đề của họ hay không, nếu có thì áp dụng mẫu này để giải quyết vấn đề. Có 4 lọai *template* khác nhau, hai trong số đó thường được sử dụng nhất là của *Coplien* và *Gamma*. Các *heading* được liệt kê dưới đây là *template* của *Copilen*.

- *Name* – tên của *mẫu*, mô tả ý tưởng giải pháp theo một số cách.
- *Problem* – vấn đề mà *mẫu* giúp giải quyết.
- *Context* – ngữ cảnh ứng dụng của mẫu (kiến trúc hoặc nghiệp vụ) và các yếu tố chính để mẫu làm việc thành công trong một tình huống nào đó.
- *Force* – các ràng buộc hoặc các vấn đề phải được giải quyết bởi mẫu; chúng tạo ra sự mất cân đối, mẫu sẽ giúp cân đối.
- *Solution* – giải pháp để cân đối các ràng buộc xung đột và làm cho hợp với ngữ cảnh.
- *Sketch* – bản phác thảo tượng trưng của các ràng buộc và cách giải quyết chúng.
- *Resulting context* – ngữ cảnh sau khi được thay đổi bởi giải pháp.
- *Rationale* – lý do và động cơ cho mẫu.

Sưu liệu của mẫu có thể gồm mã và các biểu đồ tiêu biểu. Các biểu đồ UML có thể được dùng để minh hoạ cho cách làm việc của từng mẫu. Việc chọn lựa kiểu biểu đồ phụ thuộc vào bản chất của mẫu. Các mẫu thiết kế đã được phân lớp thành *creational* (khởi tạo), s*tructural* (cấu trúc) và *behavioural* (hành vi).

- *Creational* – liên quan với việc tạo ra các thể hiện đối tượng, tách biệt với cách được thực hiện từ ứng dụng.
- *Structural* – liên quan đến các quan hệ cấu trúc giữa các thể hiện, dùng *generalization*, *aggregation* và *composition*.
- *Behavioural* – liên quan đến việc gán trách nhiệm để cung cấp chức năng giữa các đối tượng trong hệ thống.

Với các mẫu *structural* thì biểu đồ lớp sẽ cung cấp thông tin cần thiết về mẫu. Với các mẫu *behavioural* thì biểu đồ tương tác hoặc biểu đồ trạng thái sẽ giải thích cho cách chuyển giao chức năng. Đối với các mẫu *creational* thì việc chọn lựa các biểu đồ sẽ phụ thuộc vào bản chất của mẫu (mẫu thiên về hành vi hay cấu trúc).

Tuy nhiên, đặc tả UML đưa ra, một cách đặc biệt, các *cộng tác mẫu* (template collaboration) làm phương tiện để mô hình các mẫu, các cộng tác mẫu này được giải thích trong mục sau.

16.4 Các mẫu được biểu diễn như thế nào trong UML?

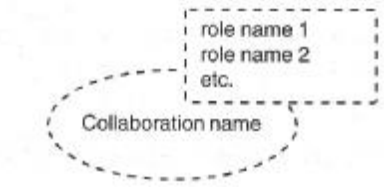
Một trong những mục tiêu của UML là "*hỗ trợ các khái niệm ở cấp cao, như thành phần, cộng tác, framework và mẫu*". Việc hỗ trợ này được thực hiện bằng cách cung cấp một cơ chế nhằm định nghĩa rõ ràng ngữ nghĩa của chúng, từ đó việc sử dụng các khái niệm được dễ dàng hơn nhằm đạt được khả năng tái sử dụng mà các phương pháp hướng đối tượng yêu cầu. Khía cạnh thuộc cấu trúc của mẫu được biểu diễn trong UML bằng cách dùng các cộng tác mẫu (xem hình 8.4).

Cộng tác mẫu được biểu diễn bằng một hình *elipse* đứt nét và một hình chữ nhật đứt nét nằm chồng lên phần cung phía trên bên phải của nó, như hình 16.1.

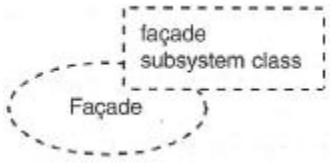
Tên của các vai trò mà các lớp tham gia đảm nhận được viết trong hình chữ nhật đứt nét như hình 16.2.

Ký hiệu này cũng được dùng để biểu diễn các lớp thực sự có liên kết với vai trò, như trong hình 16.3.

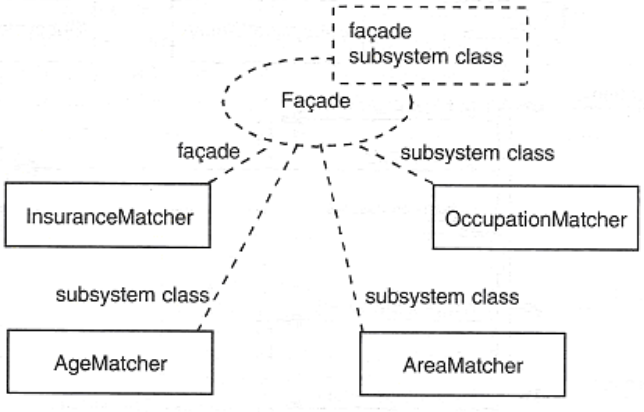
Cộng tác mẫu chỉ trình bày các vai trò tham gia và các lớp tham gia để thực hiện các vai trò này. Thay vì để hiển thị cấu trúc của cộng tác, chúng ta lại cần phải vẽ một biểu đồ lớp cho các lớp tham gia. Điều này được thực hiện dưới dạng các vai trò lớp, như trong hình 16.4, đây là một biểu đồ đơn giản hơn bởi vì chỉ có hai vai trò trong mẫu. Lưu ý đến hướng của kết hợp.



Hình 16.1: Ký hiệu cho cộng tác mẫu



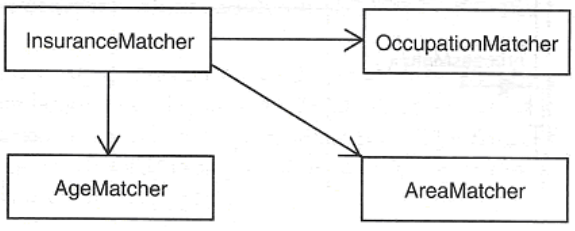
Hình 16.2: Các vai trò lớp liên quan trong mẫu Façade



Hình 16.3: Ví dụ một cộng tác lập sưu liệu sử dụng mẫu façade

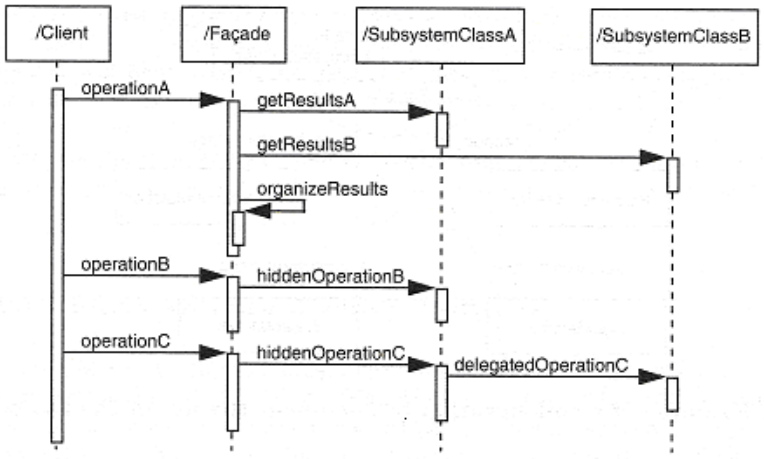
Hình 16.4: Các vai trò được biểu diễn bởi các lớp trong mẫu Façade

Khi một mẫu được áp dụng cho một bài toán nào đó trong thiết kế một hệ thống, các quan hệ có cấu trúc giữa các lớp tham gia vào cộng tác có thể được trình bày trong biểu đồ lớp. Trong hệ thống *CarMatch*, có một yêu cầu kết hợp các hợp đồng bảo hiểm với thành viên dựa trên ba đặc điểm của họ: *age* (tuổi), *home address* (địa chỉ nhà) và *occupation* (nghề nghiệp). Cách che dấu sự phức tạp của các lớp cài đặt ba loại tiến trình kết hợp là tạo ra một lớp *façade*, trong trường hợp này là *InsuranceMatcher*, để cung cấp một giao diện đơn giản cho các đối tượng client, như hình 16.5.

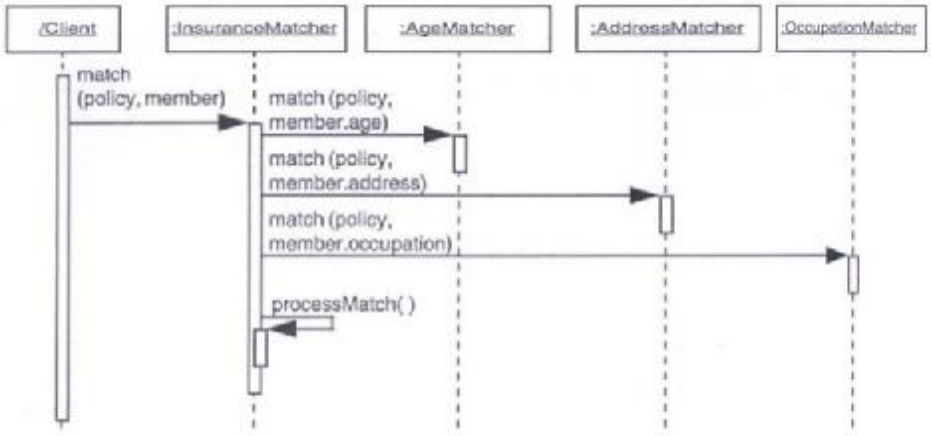


Hình 16.5: Ví dụ InsuranceMatcher *của mẫu* Façade

Tương tác liên quan trong mẫu có thể được mô hình bằng một biểu đồ tương tác. Hình 16.6 biểu diễn ba loại mẫu tiêu biểu về tương tác có thể được cung cấp bằng cách sử dụng mẫu thiết kế *façade*. Trong trường hợp đầu tiên (*operationA*), lớp façade gọi các thao tác của các lớp hệ thống con để giữ một vài kết quả và sau đó sắp xếp kết quả trước khi truyền trở lại client. Trường hợp thứ hai (*operationB*), lớp *façade* có nhiệm vụ đơn giản là dấu đi thao tác được gọi từ *client*. Trường hợp thứ ba (*operationC*), lớp *façade* một lần nữa dấu đi thao tác được gọi, ngoại trừ thao tác có liên quan đến tương tác giữa các vai trò trong hệ thống con.



Hình 16.6: Các tương tác tiêu biểu dùng mẫu Façade



Hình 16.7: Biểu đồ tuần tự cho mẫu Façade

Trong trường hợp của ví dụ *InsuranceMatcher*, tương tác thuộc về loại đầu tiên: *InsuranceMatcher* nhận một yêu cầu từ một đối tượng khác, gửi các yêu cầu này cho từng lớp hệ thống con một cách tuần tự, xử lý kết quả và trả về một kết quả đơn cho *client*, như hình 16.7.

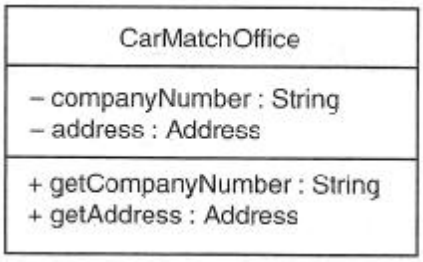
Các khía cạnh phi cấu trúc khác của mẫu được ghi trong văn bản hoặc nhãn bằng cách dùng một khuôn dạng như khuôn dạng *Coplien* được mô tả trong mục 16.3.

16.5 Áp dụng mẫu

Mẫu được áp dụng ở nhiều giai đoạn khác nhau trong qui trình phát triển hệ thống. *Các mẫu tổ chức* (organizational pattern) được dùng trong việc lập kế hoạch cho dự án; *các mẫu phân tích* (analysis pattern) được dùng trong việc lập cấu trúc cho mô hình thuộc phạm vi bài toán; *các mẫu kiến trúc* (architectural apttern) được dùng để xác định toàn bộ kiến trúc của hệ thống; *các mẫu thiết kế* (design pattern) được dùng để thiết kế các quan hệ và tương tác giữa các lớp thiết kế. Ở đây chúng ta tập trung vào *mẫu thiết kế*.

Như trên đã nói, mẫu được dự định để cho phép những người phát triển chưa có kinh nghiệm sử dụng các kinh nghiệm tích lũy của các chuyên gia. Tuy nhiên, để dùng các mẫu này, người phát triển chưa có kinh nghiệm phải có ít nhất một số kiến thức về mẫu và các vấn đề mà người này cần giải quyết. Khi đối mặt với một bài toán trong phát triển hệ thống, chúng ta không thể dễ dàng tìm được một mẫu thích hợp từ một tập các mẫu phức tạp. Trong phần này, chúng ta sẽ khảo sát một vấn đề thiết kế trong hệ thống CarMatch và đề ra một giải pháp cho vấn đề này dựa trên các mẫu.

Ví dụ 16.1

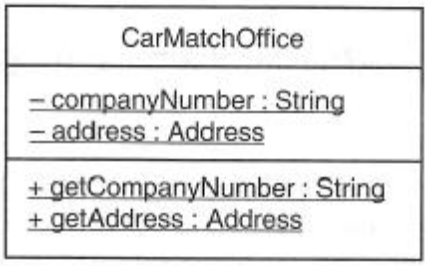


Hình 16.8: Lớp CarMacthOffice

Trong mỗi văn phòng chi nhánh *CarMatch* sẽ có một hệ thống máy tính được dùng để chạy các nghiệp vụ của chi nhánh đó. Chúng ta cần phải giữ thông tin cục bộ về chi nhánh, chẳng hạn như địa chỉ và số mà công

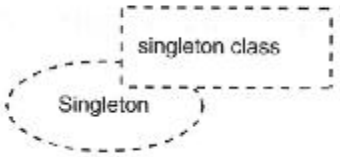
ty đã đăng ký. Vì vậy biểu đồ lớp nên có một lớp *CarMatchOffice* như trong hình 16.8. Nếu không dự định dùng để xây dựng một hệ thống phân tán kết nối tất cả các văn phòng chi nhánh lại với nhau, thì nên có một thể hiện của lớp này trong bất kỳ hệ thống nào. Làm thế nào để chúng ta đảm bảo rằng có một thể hiện luôn được tạo ra?

Một hướng tiếp cận là không tạo ra bất kỳ thể hiện nào ngoại trừ việc dùng chính lớp này và tạo ra tất cả các thuộc tính và thao tác phạm vi lớp như trong hình 16.9. Hướng tiếp cận này đã được giải thích trong mục 7.4.



Hình 16.9: Lớp CarMatchOffice *với các thao tác và thuộc tính phạm vi lớp*

Tuy nhiên hướng tiếp cận này có một số thiếu sót, bởi sau đó chúng ta muốn lớp này trở thành lớp con, và các lớp con không thể định nghĩa lại các thao tác của lớp này. Vì vậy chúng ta phải tạo ra một thể hiện đối tượng, nhưng phải bảo đảm rằng chỉ có một thể hiện luôn được tạo ra. Mẫu *Singleton* là một cách khác thực hiện điều này. Singleton có thể được dùng như một giải pháp cho câu hỏi "*Làm thế nào để xây dựng được một lớp chỉ có một thể hiện có phạm vi truy xuất public bên trong một ứng dụng?*". *Singleton* là một *mẫu creational*.



Hình 16.10: Cộng tác mẫu cho mẫu Singleton

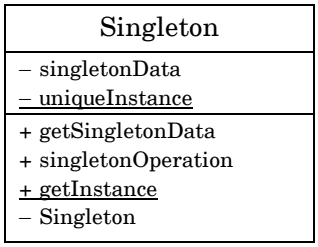
Trong mẫu *Singleton*, một thao tác phạm vi lớp *getInstance*() được thêm vào phần định nghĩa lớp. Thao tác này trả về một tham chiếu cho một thể hiện đơn của lớp hoặc sẽ tạo ra một thể hiện đơn nếu chưa có. Tham chiếu đến thể hiện được giữ như một thuộc tính phạm vi lớp. Để ngăn

chặn các đối tượng khác gọi trực tiếp lớp *Singleton* này, thao tác khởi tạo được khai báo là *private*.

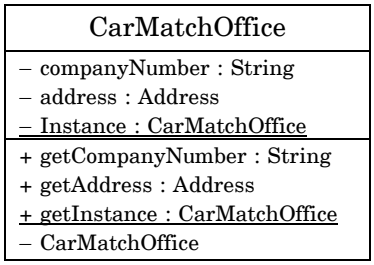
Mẫu *Singleton* rất đơn giản, chỉ liên quan đến một lớp, nên chỉ cần vẽ một cộng tác mẫu như trong hình 16.10.

Một lớp *Singleton* có thể như trong hình 16.11.

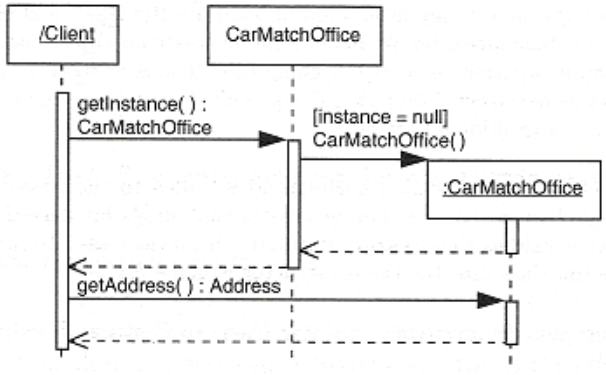
Mẫu này được áp dụng cho lớp *CarMatchOffice*, và hình 16.12 là kết quả.



Hình 16.11: Mẫu lớp Singleton

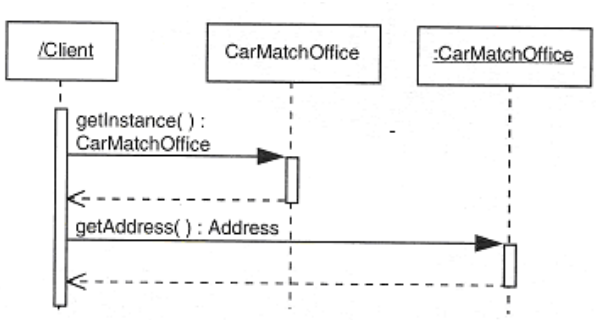


Hình 16.12: CarMatchOffice *là một lớp Singleton*



Hình 16.13: Thao tác getInstance *của* CarMatch *tạo ra một thể hiện*

Chúng ta có thể mô hình thao tác *getInstance*() trong một biểu đồ tuần tự. Có hai *scenario* có thể xảy ra: thứ nhất, thể hiện duy nhất chưa tồn tại và phải được tạo và trả về một tham chiếu đến nó; thứ hai, thể hiện duy nhất đã có và tham chiếu được trả về ngay lập tức. Đối tượng *client* sau đó sẽ gọi một thao tác trên thể hiện này. Điều này được biểu diễn trong hình 16.13 và 16.14. Hãy lưu ý sự khác biệt giữa lớp *CarMatchOffice* và thể hiện *:CarMatchOffice*.



Hình 16.14: Thao tác getInstance *của* CarMatchOffice *trả về một thể hiện tồn tại*

16.6 Cách sử dụng các mẫu

Mẫu không phải luôn là câu trả lời cho mọi vấn đề trong thiết kế hệ thống. Tuy nhiên, chúng có thể cung cấp các lợi ích có ý nghĩa bằng cách rút gọn qui trình tìm ra các giải pháp cho các vấn đề thiết kế trong các hệ thống phức tạp.

Khi gặp vấn đề trong thiết kế hệ thống, hãy xem xét các điểm sau:

- Mẫu đang tồn tại có áp dụng cho vấn đề này hay vấn đề tương tự?
- Tài liệu của mẫu này có gợi ý rằng bất kỳ một giải pháp nào khác có được chấp nhận hơn không?
- Có giải pháp đơn giản hơn không? (Không dùng mẫu cho mục đích này).
- Ngữ cảnh của mẫu và vấn đề có nhất quán với nhau hay không?
- Các kết quả của việc dùng mẫu có thể chấp nhận được hay không?
- Các ràng buộc của phần mềm đang dùng có tranh chấp khi sử dụng mẫu hay không?

Gamma và các cộng sự (1995) gợi ý bảy bước ứng dụng của một mẫu để đảm bảo sử dụng thành công mẫu.

1. Đọc mẫu để có một cái nhìn toàn cảnh.
2. Học chi tiết cấu trúc, các đối tượng tham gia và các cộng tác của mẫu.
3. Kiểm tra mã nguồn mẫu để xem ví dụ về cách dùng mẫu.
4. Chọn tên cho các đối tượng tham gia của mẫu (ví dụ như lớp) sao cho có ý nghĩa đối với ứng dụng.
5. Định nghĩa các lớp.
6. Chọn tên cho các thao tác.
7. Cài đặt các thao tác thực hiện các trách nhiệm và cộng tác trong mẫu.

Các mẫu có thể rất có ích, nhưng phải áp dụng chúng một cách cẩn thận. Ưu điểm của mẫu là cho phép tái sử dụng các kinh nghiệm của những nhà pháp triển đi trước và học tập dạng thực hành từ các ví dụ. Tuy nhiên, việc dùng mẫu có thể làm cho hệ thống cần thiết kế phức tạp hơn mức cần thiết.

Câu hỏi ôn tập

16.1 *Pattern* (mẫu) là gì?

16.2 Framework là gì?

16.3 Idiom là gì?

16.4 Mẫu được lập sưu liệu như thế nào?

16.5 Hãy liệt kê và giải thích từng heading trong khuôn dạng của Coplien.

16.6 Ba loại khác nhau của mẫu thiết kế là gì?

16.7 Hãy giải thích ý nghĩa của mỗi loại mẫu thiết kế trên.

16.8 Ký hiệu cho cộng tác template là gì?

16.9 Bốn loại biểu đồ UML có thể được dùng để mô hình một mẫu là gì?

16.10 Hãy liệt kê các bước mà Gamma và các cộng sự (1995) gợi ý nên thực hiện để kiểm tra rằng một mẫu là phù hợp?

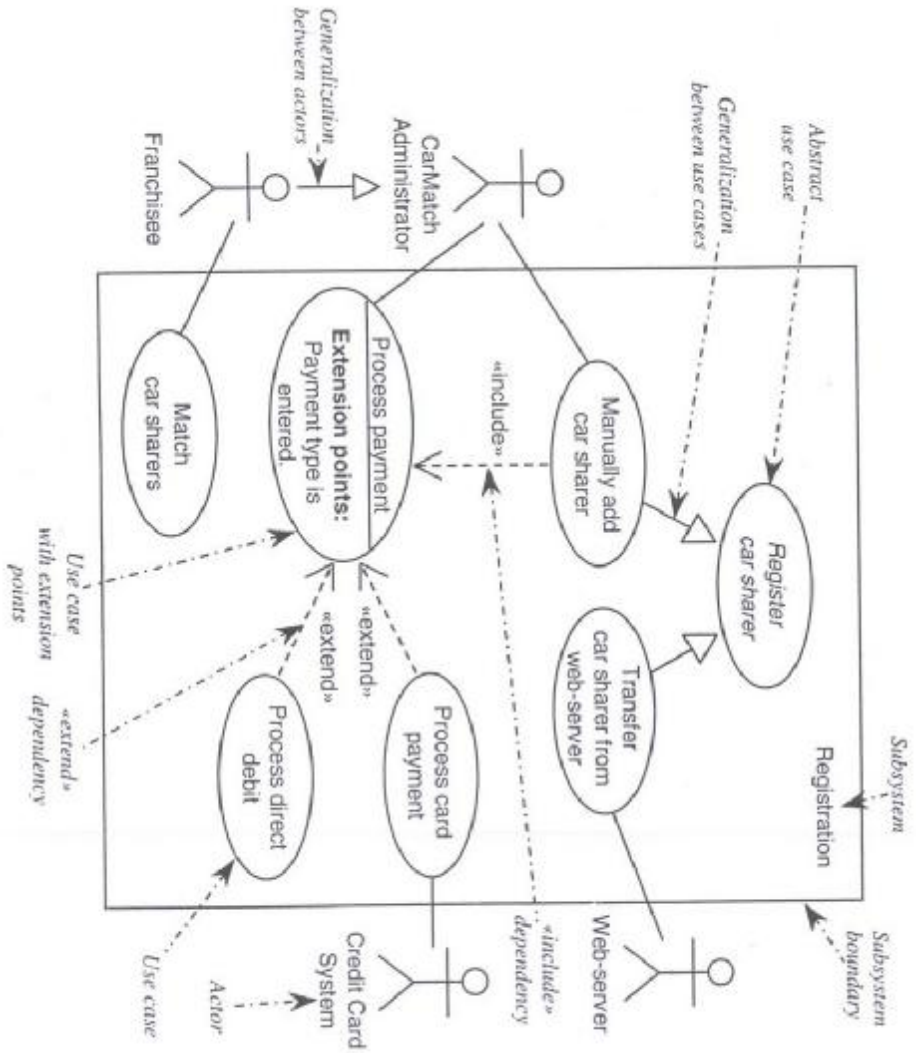
Bài tập bổ sung

16.1 Hãy tìm ra các mẫu thiết kế khác (bằng cách đọc sách hoặc tìm kiếm trên Internet) và xem có mẫu nào áp dụng được cho case study *CarMacth* và *VolBank* hay không.

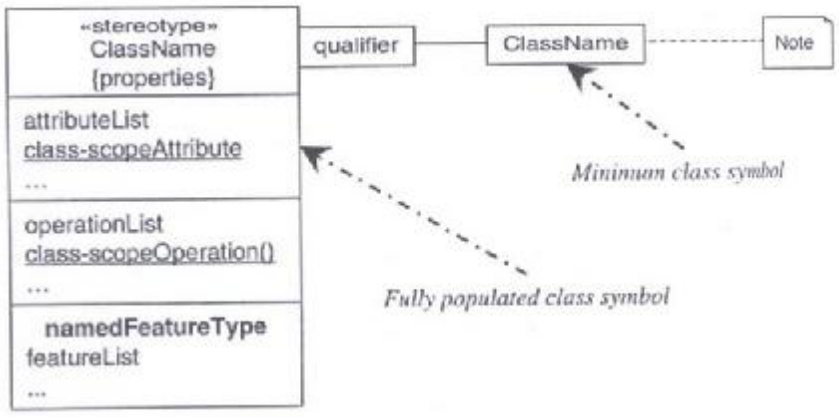
16.2 Chọn một mẫu thiết kế từ các mẫu vừa tìm được. Hãy vẽ các biểu đồ UML để mô hình cấu trúc và hành vi của mẫu này.

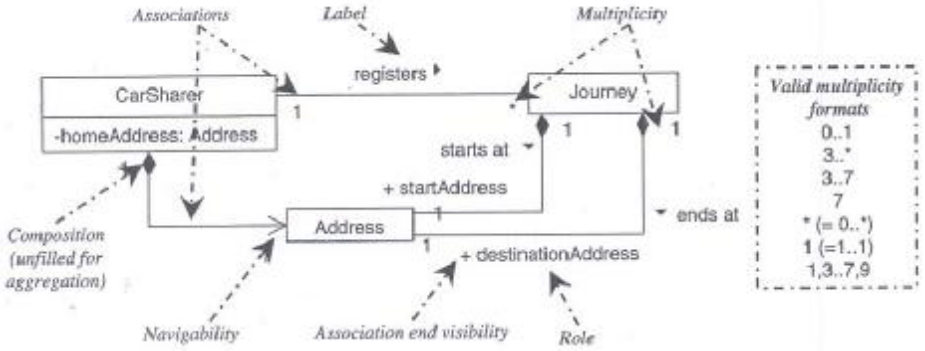
PHỤ LỤC A
TÓM TẮT KÝ HIỆU

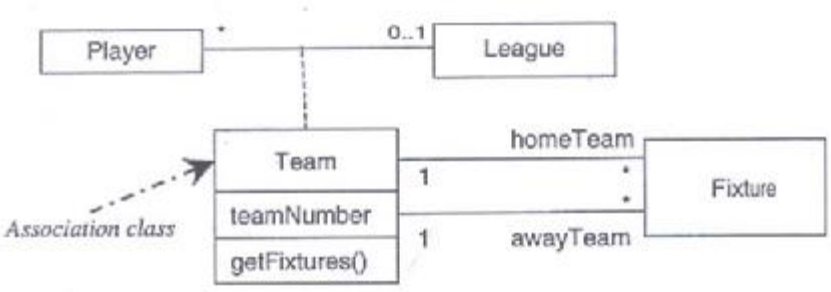
Biểu đồ Use Case

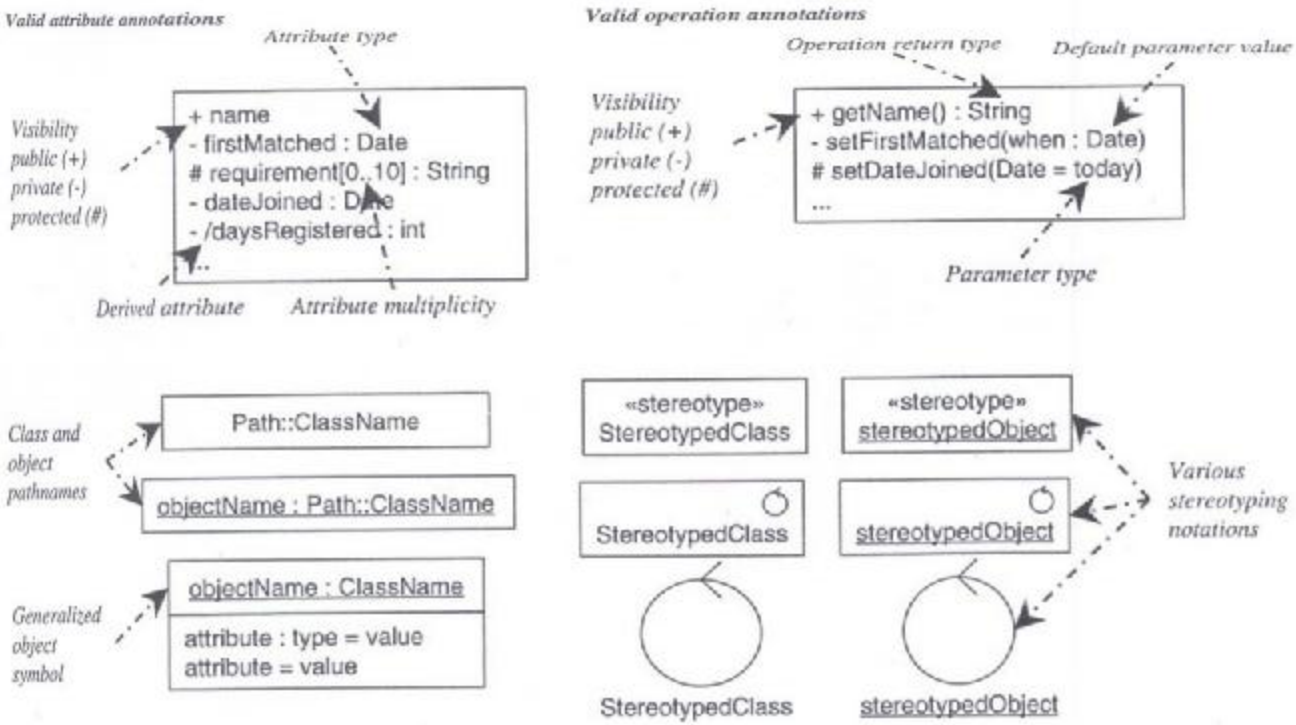


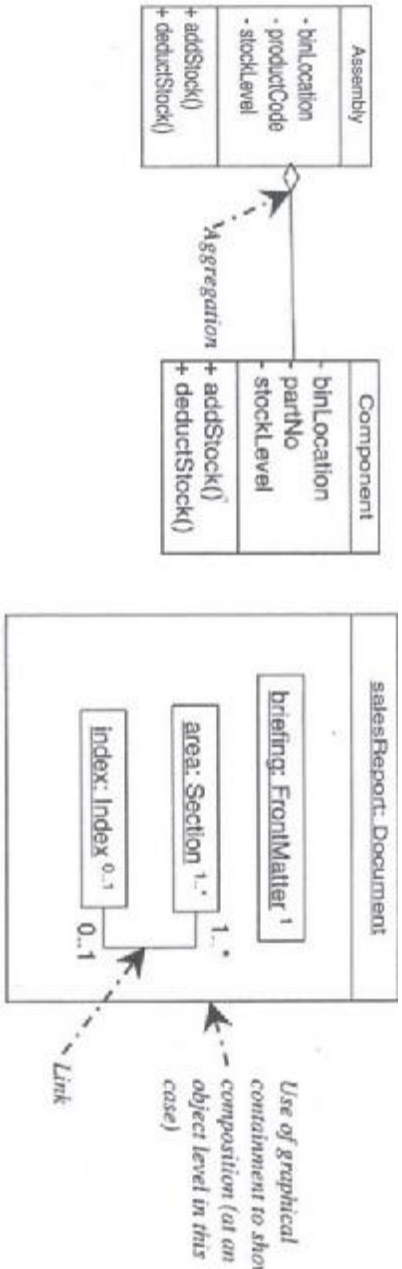
Biểu đồ lớp

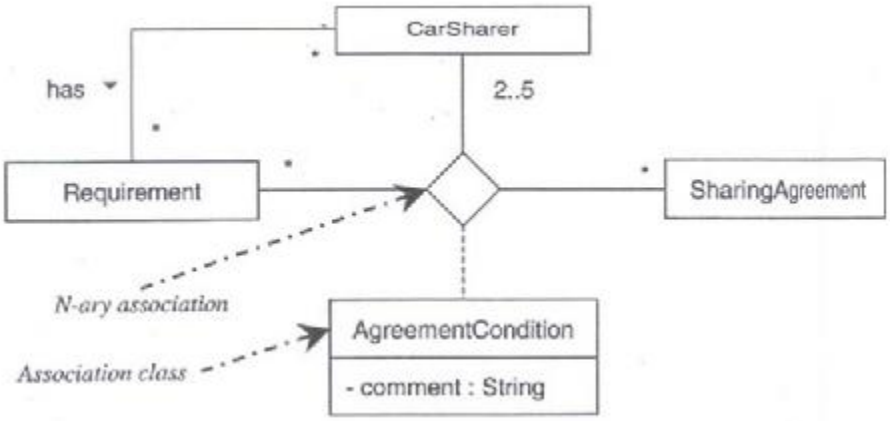

«stereotype»
ClassName
{properties}
attributeList
class-scopeAttribute
...
operationList
class-scopeOperation()
...
namedFeatureType
featureList
...
qualifier
ClassName
Note
Minimum class symbol
Fully populated class symbol

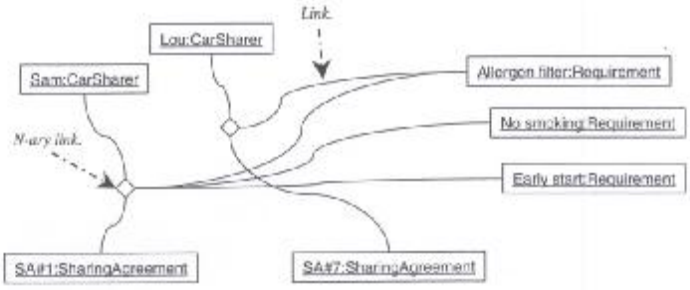

Associations
Label
Multiplicity
CarSharer
-homeAddress: Address
registers
Journey
1
1
1
starts at
+ startAddress
Address
1
ends at
1
+ destinationAddress
Composition
(unfilled for
aggregation)
Navigability
Association end visibility
Role
Valid multiplicity
formats
0..1
3..*
3..7
7
* (= 0..*)
1 (=1..1)
1,3..7,9

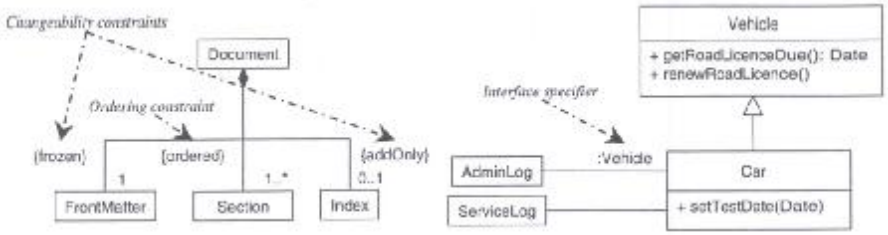

Player
*
0..1
League
Team
teamNumber
getFixtures()
homeTeam
1
*
*
1
awayTeam
Fixture
Association class

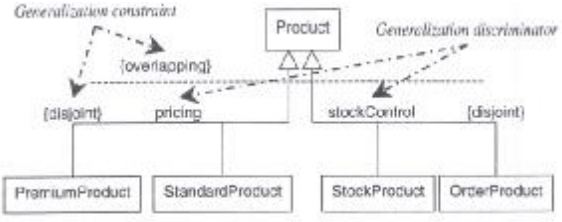

Valid attribute annotations
Attribute type
Visibility
public (+)
private (-)
protected (#)
+ name
- firstMatched : Date
requirement[0..10] : String
- dateJoined : Date
- /daysRegistered : int
...
Derived attribute
Attribute multiplicity
Valid operation annotations
Operation return type
Default parameter value
Visibility
public (+)
private (-)
protected (#)
+ getName() : String
- setFirstMatched(when : Date)
setDateJoined(Date = today)
...
Parameter type
Class and
object
pathnames
Path::ClassName
objectName : Path::ClassName
Generalized
object
symbol
objectName : ClassName
attribute : type = value
attribute = value
«stereotype»
StereotypedClass
«stereotype»
stereotypedObject
Various
stereotyping
notations
StereotypedClass
stereotypedObject
StereotypedClass
stereotypedObject

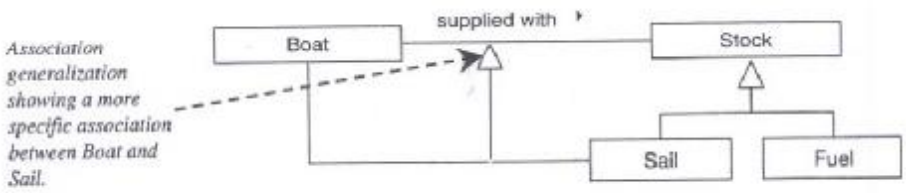

Assembly
- binLocation
- productCode
- stockLevel
+ addStock()
+ deductStock()
Aggregation
Component
- binLocation
- partNo
- stockLevel
+ addStock()
+ deductStock()
salesReport: Document
briefing: FrontMatter 1
area: Section 1..*
index: Index 0..1
1..*
0..1
Link
Use of graphical containment to show composition (at an object level in this case)

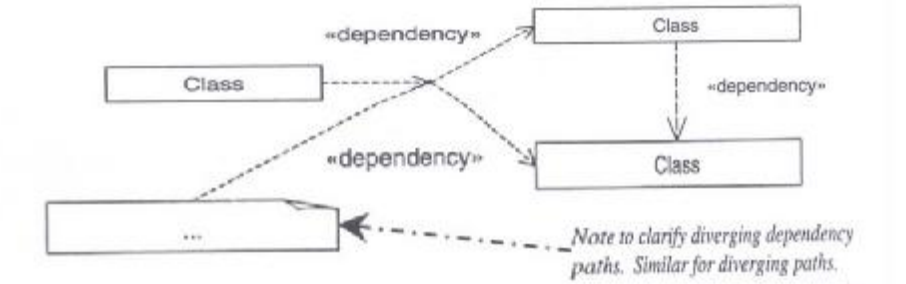

CarSharer
has
2..5
Requirement
SharingAgreement
N-ary association
Association class
AgreementCondition
- comment : String

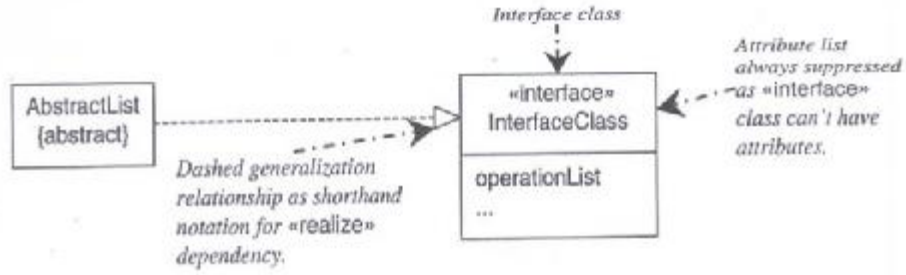

Link
Lou:CarSharer
Sam:CarSharer
Allergen filter:Requirement
No smoking:Requirement
Early start:Requirement
N-ary link
SA#1:SharingAgreement
SA#7:SharingAgreement

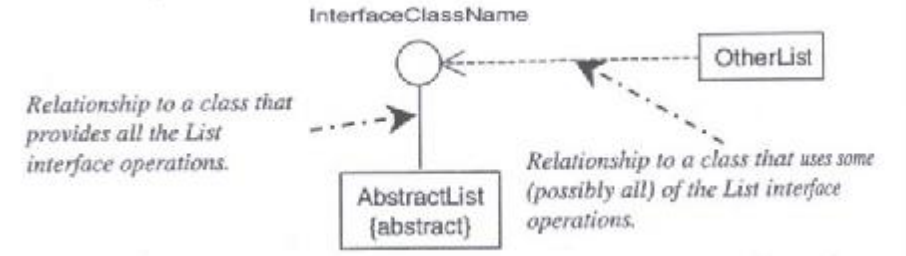

Changeability constraints
Document
Ordering constraint
{frozen}
{ordered}
{addOnly}
1
1..*
0..1
FrontMatter
Section
Index
Vehicle
+ getRoadLicenceDue(): Date
+ renewRoadLicence()
Interface specifier
:Vehicle
AdminLog
ServiceLog
Car
+ setTestDate(Date)

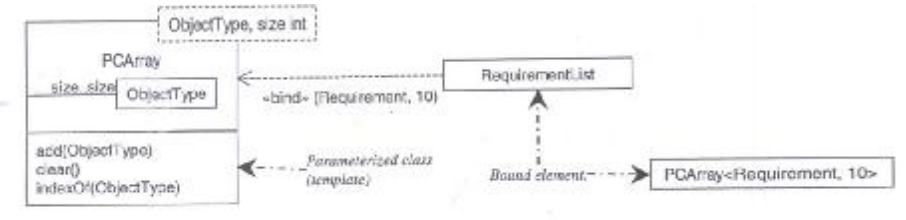

Generalization constraint
Product
Generalization discriminator
{overlapping}
{disjoint}
pricing
stockControl
{disjoint}
PremiumProduct
StandardProduct
StockProduct
OrderProduct


Association generalization showing a more specific association between Boat and Sail.
Boat
supplied with
Stock
Sail
Fuel

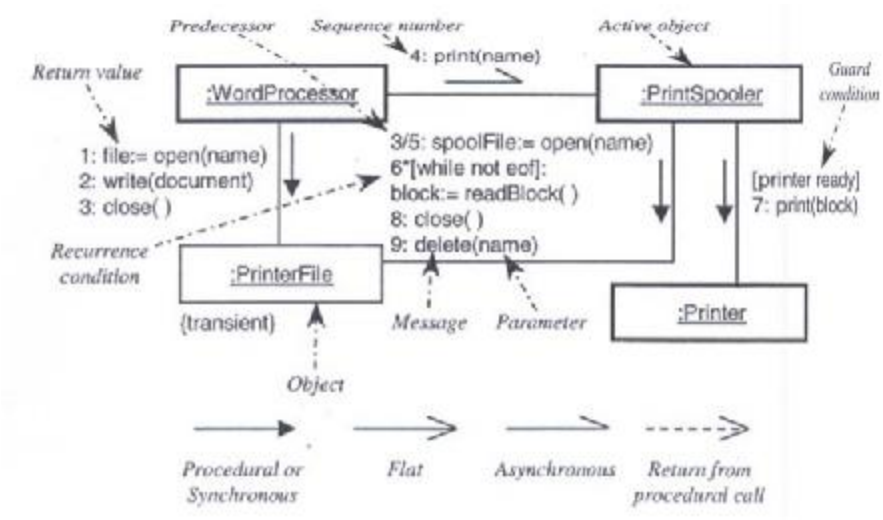

«dependency»
Class
Class
«dependency»
«dependency»
Class
...
Note to clarify diverging dependency paths. Similar for diverging paths.


Interface class
Attribute list always suppressed as «interface» class can't have attributes.
AbstractList {abstract}
«interface» InterfaceClass
operationList
...
Dashed generalization relationship as shorthand notation for «realize» dependency.

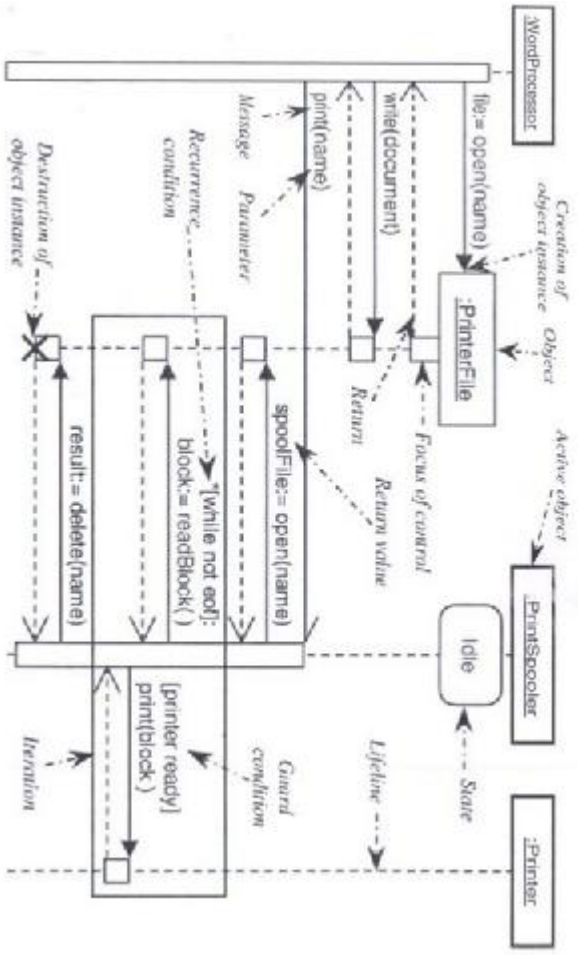

InterfaceClassName
OtherList
Relationship to a class that provides all the List interface operations.
Relationship to a class that uses some (possibly all) of the List interface operations.
AbstractList {abstract}


ObjectType, size int
PCArray
size, size
ObjectType
RequirementList
«bind» (Requirement, 10)
add(ObjectType)
clear()
indexOf(ObjectType)
Parameterized class (template)
Bound element
PCArray<Requirement, 10>

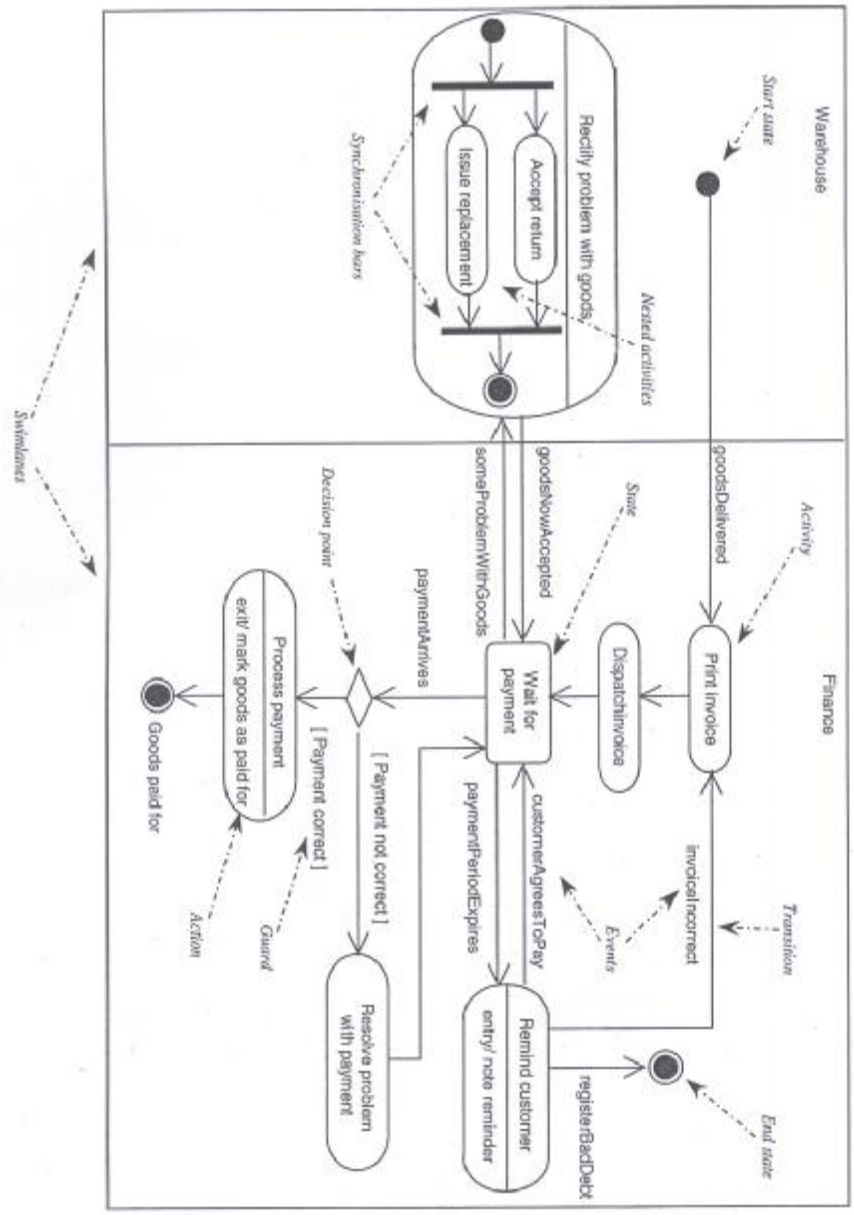
Biểu đồ cộng tác



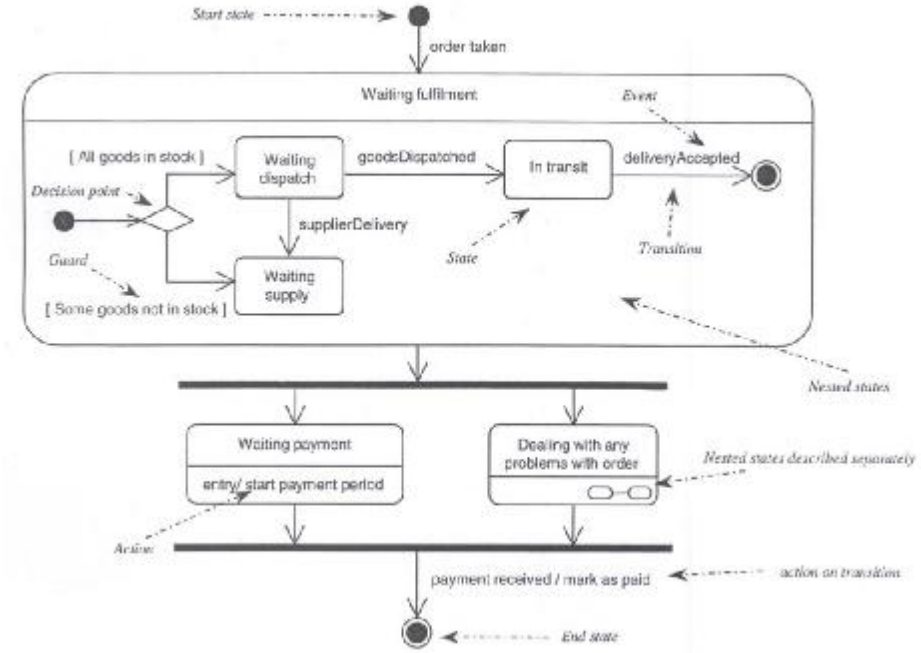
Biểu đồ tuần tự



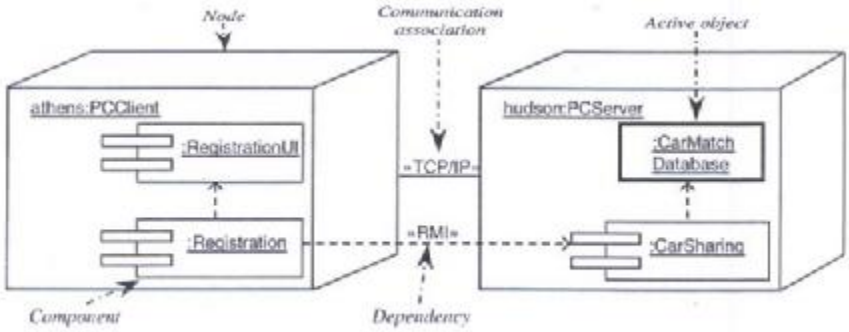
Biểu đồ hoạt động



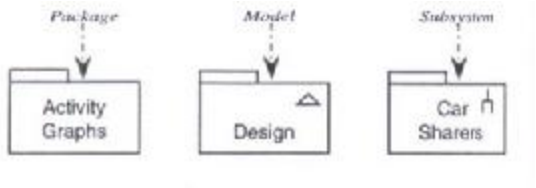
Biểu đồ trạng thái



Biểu đồ cài đặt



Quản lý mô hình



PHỤ LỤC B
TRẢ LỜI CÁC CÂU HỎI ÔN TẬP

Chương 2: CƠ BẢN VỀ UML

2.1 James Rumbaugh, Grady Booch và Ivar Jacobson.

2.2 Object Management Group (OMG).

2.3 Cung cấp đặc tả làm thế nào để các mô hình UML có thể được chuyển đổi giữa các áp dụng.

2.4 Cú pháp trừu tượng, luật hình thức và ngữ nghĩa.

2.5 Đối tượng người dùng, mô hình, siêu mô hình và siêu-siêu mô hình.

2.6 Trong ngôn ngữ ràng buộc đối tượng OCL.

2.7 Để nhóm các biểu đồ cùng nhau tạo nên UML dựa trên mức liên kết giữa chúng.

2.8 Revision Task Force (RTF).

2.9 Chúng được dùng để cung cấp các kiểu mở rộng UML cho một lĩnh vực riêng như mô hình nghiệp vụ hoặc phát triển hệ thống thời gian thực.

2.10 Ngoại trừ (c), những cái còn đều là sự vật trừu tượng?

2.11 Ngoại trừ (d), những cái còn đều là mô hình?

2.12 Chọn ba lý do trong số sau:

a Nó trở thành chuẩn công nghiệp phổ biến.

b Nó hợp nhất các tiếp cận của các tác giả khác nhau thành một ký hiệu duy nhất.

c Nó có thể mở rộng để áp dụng trong các lĩnh vực riêng.

d Nó được hỗ trợ bởi qui trình UP, cung cấp một mô hình xử lý cho cách thức áp dụng nó.

2.13 Inception, Elaboration, Construction và Transition.

2.14 *Dòng công việc* xác định các *hoạt động* trong dự án và vai trò của các *worker*, những người thực hiện *hoạt động*. *Dòng công việc* nhóm các hoạt động lại thành các tác vụ dự án chủ yếu, như tác vụ xác định các yêu cầu chẳng hạn. *Hoạt động* xác định công việc phải được

thực hiện, và được định nghĩa thành dãy các *bước* nhằm mô tả chi tiết cách thực hiên công việc đó.

Chương 3: USE CASE

3.1 Một chuỗi các hành động được thực hiện bởi hệ thống đưa ra một kết quả có ý nghĩa đối với một người dùng hoặc hệ thống ngoài.

3.2 Một hình oval với tên use case được đặt ở bên trong hoặc bên dưới.

3.3 Một người dùng hoặc hệ thống ngoài tương tác với hệ thống hoặc hệ thống con đang được mô hình.

3.4 Một hình người với tên actor đặt ở bên dưới.

3.5 Với một usecase, hệ thống có thể có một số cách thực hiện khác nhau phụ thuộc vào các tình huống khác nhau, vào trạng thái của hệ thống và vào đáp ứng của actor.

3.6 Mô hình nghiệp vụ hoặc xác định yêu cầu.

3.7 Mục đích:

a Mô hình chuỗi hành động được thực hiện bởi hệ thống đưa ra một kết quả đáng chú ý đến người và vật bên ngoài hệ thống, được hiểu là actor.

b Cung cấp một khung nhìn mức cao về những gì hệ thống làm và ai dùng nó.

c Cung cấp cơ sở để xác định giao diện *người-máy* cho hệ thống.

d Mô hình các scenario.

e Dùng các ký hiệu đơn giản dễ hiểu đối với người dùng cuối và truyền thông với họ về khung nhìn ở mức cao của hệ thống.

f Dùng làm cơ sở để phác thảo các đặc tả kiểm tra.

3.8 Tổng quát hoá.

3.9 Kết hợp tổng quát hoá và các quan hệ <<*include*>>, <<*extend*>>.

3.10 Với quan hệ <<*include*>>, use case *được include* luôn xảy ra như là một bộ phận của use case *include* nó; còn với quan hệ <<*extend*>>, use case *extend* không bắt buộc phải xảy ra.

3.11 Một đường nối hai phần tử với biểu tượng hình tam giác hướng đến kiểu cha.

3.12 Một mũi tên mở đứt nét, đầu mũi tên nối với use case *được include* cùng với khuôn dạng <<*include*>> đặt dọc theo mũi tên.

3.13 Một mũi tên mở đứt nét, đầu mũi tên nối với use case *được extend* cùng với khuôn dạng << *extend*>> đặt dọc theo mũi tên.

3.14 Điểm trong quá trình thi hành use case, tại đó có một use case khác mở rộng chức năng của nó, xảy ra hay không phụ thuộc vào một số điều kiện.

3.15 Dòng công việc *Requirements*.

3.16 *Find Actors and Use Cases* (tìm actor và use case), *Prioritize Use Cases* (ưu tiên hoá use case), *Detail a Use Cases* (chi tiết hoá một use case), *Structure the Use Cases Model* (lập cấu trúc cho mô hình use case), *Prototype User Interface* (phác thảo giao diện người dùng).

Chương 4: BIỂU ĐỒ LỚP, LỚP VÀ MỐI KẾT HỢP

4.1 Biểu đồ lớp chứa chủ yếu các lớp và các kết hợp. Các ký hiệu khác là các ghi chú và các đối tượng. Một biểu đồ lớp chỉ gồm có đối tượng được xem là biểu đồ đối tượng. Xem thêm chương 7.

4.2 Biểu đồ lớp được dùng để:

a Làm tài liệu cho các lớp cấu thành hệ thống hoặc hệ thống con.

b Mô tả kết hợp, tổng quát hoá và các quan hệ kết tập giữa các lớp trong biểu đồ.

c Chỉ rõ các đặc trưng của lớp, các thuộc tính và thao tác chính của mỗi lớp.

d Biểu đồ lớp được dùng khắp nơi trong chu trình phát triển, từ các lớp đặc thù trong lĩnh vực bài toán đến mô hình cài đặt, để chỉ ra cấu trúc lớp cho hệ thống.

e Sưu liệu về cách tương tác với các thư viện lớp có trước.

f Chỉ rõ các thể hiện đối tượng cụ thể bên trong cấu trúc lớp.

g Chỉ rõ các giao diện được các lớp hỗ trợ.

4.3 Lớp dùng để mô tả các đặc trưng và tính chất chung của một tập các đối tượng trong lĩnh vực bài toán hoặc trong giải pháp cài đặt. Lớp có các tính chất được đặt trong cặp dấu ngoặc móc, {...}, và các đặc trưng là thuộc tính và thao tác.

4.4 Ký hiệu cơ bản nhất của lớp là một hình chữ nhật với tên lớp được đặt ở bên trong.

4.5 Tên đường dẫn của lớp phản ánh cấu trúc quản lý mô hình và còn được gọi là *namespace* của lớp.

4.6 Một lớp trong gói *Accounts* nằm trong gói *Finance* sẽ có tên đường dẫn là *Finance*::*Accounts*::*Class*.

4.7 Thuộc tính là một mục dữ liệu mà một lớp chịu trách nhiệm bảo tồn.

4.8 Ký hiệu cơ bản cho một thuộc tính chỉ gồm tên. Tên thuộc tính được bắt đầu với một chữ thường và không chứa khoảng trắng. Mỗi từ lập thành tên được bắt đầu với chữ in.

4.9 Chúng gồm: kiểu dữ liệu, các tính chất riêng, tính khả kiến, dẫn xuất, giá trị khởi tạo và bản số.

4.10 Ký hiệu cho các tính chất nói trên như sau:

a Kiểu dữ liệu có dạng *attributeName* : *type*.

b Các tính chất riêng được đặt trong cặp dấu móc nhọn, {...}.

c Tính khả kiến được chỉ rõ với ký hiệu -, +, hoặc # (hoặc do người dùng định nghĩa)

d Dẫn xuất được biểu thị bằng ký hiệu '/' đặt trước tên.

e Giá trị khởi tạo có dạng *attributeName* : *type* = *initialValue*.

f Bản số có dạng *attributeName* [*lower..upper*].

4.11 Một thao tác là một trách nhiệm thuộc về chức năng mà một lớp phải có. Các thao tác trên lớp cộng tác với nhau để cung cấp một chức năng có thể nhìn thấy từ bên ngoài.

4.12 Tham số là một mục dữ liệu được truyền cho thao tác. Kiểu dữ liệu có thể là kiểu thư viện chuẩn (như *String*, *int*, *Date*) hoặc kiểu lớp trong lĩnh vực bài toán (như *CarSharer*)

4.13 *Truyền bằng tham chiếu* là truyền địa chỉ ô nhớ. Bất cứ một thay đổi nào trên tham số đều làm thay đổi thuộc tính được truyền. *Truyền bằng tham trị* là truyền một bản sao của thuộc tính. Sự thay đổi trên tham số không làm thay đổi thuộc tính được truyền.

4.14 Ký hiệu cơ bản của thao tác có dạng *operationName*: *type*, có thể kèm thêm ký hiệu về tính khả kiến.

4.15 Ký hiệu tham số của thao tác là một danh sách các tham số được đặt trong cặp dấu ngoặc và được đặt sau tên của thao tác. Mỗi tham số có dạng *parameterName*: *type* = *defaultValue*. Kiểu và giá trị khởi tạo có thể được bỏ qua; cũng được phép mô tả tham số chỉ gồm kiểu.

4.16 Ở mức quan niệm, các lớp phản ánh lĩnh vực bài toán. Chúng phản ánh nhiều hơn về các yêu cầu xử lý dữ liệu. Đến mức đặc tả và cài đặt, mô hình nhấn mạnh đến trách nhiệm về chức năng của các lớp. Lúc này các lớp liên quan đến cài đặt hệ thống được đưa ra và các thao tác của các cộng tác lớn hơn được thêm vào để bổ sung cho các thao tác get và set đơn giản ban đầu.

4.17 Giá trị khởi tạo có dạng *attributeName*: *type* = *initialValue*.

4.18 Giá trị mặc định có dạng *parameterName*: *type* = *defaultValue* hoặc *parameterName* = *defaultValue*.

4.19 Một kết hợp cho biết hai lớp sẽ truyền thông điệp cho nhau.

4.20 Ký hiệu cơ bản cho một kết hợp là một đường thẳng liền nét. Nhãn (tên) của kết hợp được gắn kèm một đầu *mũi tên đặc*, ở trước hoặc sau cho biết hướng kết hợp được xác định từ ngữ nghĩa của nhãn.

4.21 Vai trò của một lớp trong kết hợp được chỉ rõ bằng cách thêm một *tên vai trò* ở đầu kết hợp gần với lớp liên quan. Hình thức của *tên vai trò* giống với tên thuộc tính.

4.22 Trong ngữ cảnh thuộc tính, multiplicity đặc tả số giá trị riêng biệt mà thuộc tính có thể lưu trữ. Trong ngữ cảnh kết hợp, multiplicity đặc tả số thể hiện tham gia vào kết hợp.

4.23 Ký hiệu của multiplicity có dạng *lowerBound..upperBound*. Với thuộc tính multiplicity được đặt giữa cặp dấu ngoặc vuông. Cận dưới *lowerBound* là một số nguyên không âm. Cận trên *upperBound* là một số nguyên không âm bằng hoặc lớn hơn cận dưới. Nếu có nhiều multiplicity thì chúng được đặc tả thành một danh sách phân cách bởi dấu phẩy

4.24 Thuộc tính và kết hợp.

4.25 Chúng gồm định hướng quan niệm, định hướng đặc tả và định hướng cài đặt.

4.26 Hai cách chính để vẽ biểu đồ lớp là tiếp cận hướng dữ liệu và tiếp cận hướng chức năng. Tiếp cận hướng dữ liệu là thích hợp trong điều tra ban đầu và xác định các trách nhiệm về dữ liệu. Tiếp cận hướng chức năng xây dựng biểu đồ lớp từ các cộng tác nhằm thực hiện đầy đủ các chức năng được thấy từ bên ngoài.

Chương 5: BIỂU ĐỒ LỚP KẾT TẬP, HỢP THÀNH VÀ TỔNG QUÁT HOÁ

5.1 Kết tập ngụ ý rằng đầu *whole* (toàn thể) của kết hợp nhóm các thể hiện ở đầu *part* (bộ phận) lại với nhau, hơn nữa nó còn mang ý nghĩa bao hàm.

5.2 Về ngữ nghĩa, kết tập ngụ ý sự liên kết là mạnh hơn so với kết hợp bình thường.

5.3 Hợp thành là kết hợp chặt chẽ hơn so với kết tập khi thể hiện *part* có chu kỳ sống gắn kết với *whole*. *Part* không được tạo ra trước *whole* và bị huỷ khi *whole* bị huỷ.

5.4 Trong kết tập thể hiện *part* không có chu kỳ sống gắn kết với *whole*, chúng có thể tồn tại độc lập trong khi vẫn tham gia vào kết tập.

5.5 Ký hiệu của kết tập là một hình thoi không tô đặt ở đầu *whole*.

5.6 Hoặc là một hình thoi tô đặc được đặt ở đầu *whole*, hoặc ký hiệu đối tượng chứa dạng đồ họa.

5.7 Kết hợp chuẩn cho biết thao tác giữa hai lớp có quan hệ tương tác, còn kết tập và hợp thành làm rõ tính tự nhiên của mối quan hệ *whole-part* (toàn thể - bộ phận).

5.8 Tổng quát hoá ngụ ý các đối tượng của một lớp là một loại (*kind of*) đối tượng của một lớp khác, như chuối là một loại trái cây. Trong một biểu đồ lớp của một ứng dụng ngân hàng phải đặc tả *Nợ* và *Có* là hai loại giao dịch.

5.9 Ký hiệu của tổng quát hoá là một đường liền nét từ lớp con đến đầu mũi tên hình tam giác đặt ở lớp cha.

5.10 Chúng là các đường liền nét rời nhau hoặc hội tụ.

5.11 Ràng buộc tổng quát hoá đặc tả hai điều. Thứ nhất là quan hệ giữa các cấu trúc tổng quát hoá chia sẻ cùng một lớp cha (gồm *overlapping* hoặc *disjoint*). Thứ hai là qui mô (gồm *incomplete* hoặc *complete*). Như vậy bốn ràng buộc tổng quát hoá được UML định nghĩa trước là *overlapping*, *disjoint*, *incomplete* và *complete*. Ví dụ một cấu trúc về tập con các lớp mô tả các kiểu nhân viên hoặc khách hàng khác nhau phải có ràng buộc *incomplete* ; hoặc kiểu cha của phương tiện vận chuyển phải có ràng buộc *disjoint* trên hai cấu trúc tổng quát hoá một cho nhiên liệu và một cho động cơ.

5.12 Ví dụ *mapReferenceType* (kiểu tham chiếu bản đồ) là phân biệt giữa các tham chiếu bản đồ của *OS*, *Tiger* và *Latitude-Longitude* trong gói *GeoLocation*. Xem bài tập 5.3.

5.13 Dùng ngôn ngữ ràng buộc đối tượng.

5.14 Nếu dùng không thích hợp sẽ làm cho cài đặt không được mềm dẽo. Nên cẩn thận khi dùng chúng.

5.15 Trong tiếp cận top-down, các lớp cha có thể có nên được xác định bởi các nhà phân tích rồi cấu trúc tổng quát hoá sẽ được khảo sát và xem xét. Tiếp cận này dựa chủ yếu vào kinh nghiệm của nhà phân tích.

5.16 Tiếp cận bottom-up tìm kiếm các trách nhiệm được chia sẻ. Các thao tác và các thuộc tính tương tự nhau phải gợi ý tốt các cấu trúc tổng quát hoá chưa được phát hiện.

Chương 6: BIỂU ĐỒ LỚP, TÌM HIỂU THÊM VỀ KẾT HỢP

6.1 Ký hiệu *visibility* ở đầu mút của kết hợp (-, +, # hoặc do người dùng định nghĩa) được đặt trước tên vai trò.

6.2 Chúng là {*changeable*}, {*addOnly*} và {*frozen*}.

6.3 Ở mức quan niệm là {*ordered*}, còn ở mức đặc tả và cài đặt là {*sorted*}.

6.4 *Navigability* được trình bày với một đầu mũi tên chỉ rõ hướng thông điệp được gởi. Khi nào cần cài đặt các quan hệ thể hiện sự cộng tác hai phía, mối kết hợp phải được đặc tả tường minh để nó có thể được điều hướng theo hai chiều.

6.5 Kết hợp *interface specifier* có dạng :*Class* ở đầu cuối kết hợp cho biết đầu nguồn của kết hợp chỉ yêu cầu các thao tác public hơn là giao diện của chính lớp kết hợp.

6.6 *Qualifier* cho phép đặc tả các thuộc tính dùng để định danh sự xuất hiện của thể hiện liên quan đến kết hợp.

6.7 Có dấu / trước tên của dẫn xuất.

6.8 UML cung cấp một kết hợp n-ngôi, một UML ký hiệu hình thoi được liên kết với các đường liền nét nối đến các lớp tham gia kết hợp.

6.9 Nếu một kết hợp được tìm thấy có thuộc tính hoặc thao tác, thì chúng được mô hình thành lớp kết hợp, có ký hiệu giống lớp, được nối với kết hợp (hai ngôi hoặc n-ngôi) bằng đường đứt nét.

Chương 7: BIỂU ĐỒ LỚP, CÁC KÝ HIỆU KHÁC

7.1 Biểu đồ đối tượng cung cấp một cách thức hữu ích để minh họa các quan điểm thảo luận và để minh họa tính hiệu quả của các lớp theo một cách nào đó.

7.2 Quan hệ *instanceOf* chỉ rõ sự phụ thuộc của đối tượng vào lớp, còn quan hệ liên kết cho biết có sự xuất hiện của một mối kết hợp giữa hai đối tượng.

7.3 Được phép: nhãn kết hợp, tên vai trò, đặc tả giao diện (interface specifier), tính khả kiến, điều hướng, kết tập, hợp thành, định danh

(qualifier). Không được phép là bản số. Liên kết là thể hiện của kết hợp và chỉ liên quan đến các đối tượng nhìn thấy trong kết nối.

7.4 Các khuôn dạng <<*become*>> và <<*copy*>> chỉ rõ ảnh hưởng của hành vi động lên mô hình đối tượng tĩnh. Quan hệ *become* dùng để chỉ rõ một thay đổi trạng thái của một đối tượng, còn quan hệ *copy* cho thấy bản sao của một thể hiện đối tượng. Cách dùng khác của *copy* là minh hoạ việc truyền tham số đối tượng bằng tham trị.

7.5 Ký hiệu của quan hệ phụ thuộc *realize* được dẫn xuất từ ký hiệu của quan hệ tổng quát hoá (đầu mũi tên hình tam giác) và ký hiệu của quan hệ phụ thuộc chuẩn (đường đứt nét).

7.6 Đặc trưng phạm vi lớp được gạch chân. Điều này gây nên sự lẫn lộn vì ký hiệu UML cho thể hiện đối tượng cũng được gạch chân.

7.7 Đặc trưng phạm vi thể hiện liên quan đến chính đối tượng. Các thuộc tính phạm vi lớp lưu cùng một giá trị đối với tất cả các thể hiện của lớp, các thao tác phạm vi lớp làm việc trên các thuộc tính phạm vi lớp lưu hoặc xử lý các thể hiện của lớp (ví dụ thao tác constructor).

7.8 Một là tên khuôn dạng trong cặp dấu << và >>, hai là một biểu tượng được đặt ở góc trên bên phải của ngăn chứa tên, hoặc một biểu tượng với tên được đặt bên dưới và không thêm một đặc trưng nào khác.

7.9 Lớp *enumeration* cung cấp một danh sách các giá trị được định trước.

7.10 Lớp *utility* cung cấp cách thuận lợi để đặc tả một họ các biến và các thao tác toàn cục trong một nơi được xác định rõ ràng.

7.11 Một là lớp giao diện với khuôn dạng <<*interface*>> và một là biểu tượng hình tròn nhỏ với tên được đặt ở bên trên hoặc bên dưới. Khi dùng biểu tượng sẽ không có một thuộc tính hoặc thao tác nào được hiển thị liệt kê.

7.12 Lớp *type* có thao tác và có thể có các thuộc tính cũng như các kết hợp hướng ra ngoài. Các thuộc tính và các kết hợp chỉ để hỗ trợ cho việc đặc tả thao tác. Các thao tác của lớp *type* không có phương thức, chúng chỉ có nhiệm vụ giữ chỗ và chắc rằng có lớp khác cài đặt. Lớp *implementationClass* là lớp hiện thực một lớp *type* bằng cách cài đặt các phương thức của lớp *type*.

7.13 Ký hiệu phụ thuộc <<*realize*>> từ lớp *implementationClass* đến lớp *type* của nó.

7.14 Ký hiệu cơ bản cho một lớp *parameterized* (lớp mẫu) giống như ký hiệu lớp thêm một hộp đứt nét phủ lên góc trên bên phải của ký hiệu lớp. Trong hộp danh sách tham số, các tham số được liệt kê cách nhau bởi dấu phẩy,. Bản số và kiểu của thuộc tính cũng như giá trị trả về của thao tác trong lớp mẫu có thể được định nghĩa dùng các tham số trong hộp.

7.15 Một là phụ thuộc <<*bind*>> và một là tên lớp dạng *templateName* <*arguments*>. Nếu dùng phụ thuộc <<*bind*>> phải kèm <*arguments*> phía sau thể hiện các tham số của lớp mẫu.

7.16 Các khác biệt chính bao gồm:

- **Lớp <<*type*>>** có các thao tác, các thuộc tính và các kết hợp hướng ra ngoài. Các thuộc tính và các kết hợp này nhằm hỗ trợ việc đặc tả các thao tác của lớp. Các thao tác của lớp *type* không có phương thức - chúng là các thao tác giữ chỗ nhằm đảm bảo rằng một lớp khác cài đặt lớp này sẽ cung cấp tất cả các thao tác cần thiết. Với định hướng đặc tả và cài đặt, một lớp *type* không thực sự làm điều gì nên nó không có các thể hiện trực tiếp.
- **Lớp {*abstract*}** có thể có thuộc tính, thao tác và kết hợp. Các thao tác được cài đặt. Một lớp trừu tượng có thể hỗ trợ các thể hiện đối tượng trực tiếp bởi vì nó có các thuộc tính, các thao tác có phương thức, và các kết hợp. Tuy nhiên, các lớp trừu tượng không có các thể hiện đối tượng trực tiếp đơn giản vì chúng là trừu tượng.
- **Lớp <<*interface*>>** không có các thuộc tính hoặc các kết hợp hướng ra bên ngoài. Các thao tác của nó không có phương thức; chúng là các thao tác giữ chỗ để bảo đảm rằng một lớp nào khác hiện thực nó sẽ cung cấp tất cả các thao tác cần thiết. Với định hướng đặc tả và cài đặt, do lớp *interface* không thực sự làm điều gì, nên nó không thể có các thể hiện đối tượng trực tiếp.

Chương 8: BIỂU ĐỒ CỘNG TÁC

8.1 Một tập các thành viên, các đối tượng và các vai trò, làm việc cùng nhau để đạt được một kết quả có nghĩa trong ngữ cảnh của hệ thống.

8.2 Một dãy các thông điệp được gởi giữa các đối tượng hoặc vai trò trong ngữ cảnh một cộng tác để đạt được chức năng của cộng tác.

8.3 Kết quả là nhận ra được nhu cầu phát sinh các lớp mới, thuộc tính mới và cả thao tác mới.

8.4 Biểu đồ cộng tác được dùng để:

- Mô hình cộng tác giữa các đối tượng hoặc các vai trò nhằm thực hiện chức năng của một use case.
- Mô hình cộng tác giữa các đối tượng hoặc các vai trò nhằm thực hiện chức năng của một thao tác.
- Mô hình các cơ chế bên trong *thiết kế kiến trúc* (architectural design).
- Biểu đồ cộng tác được chú thích với các tương tác. Các tương tác này trình bày các thông điệp giữa các đối tượng hoặc vai trò bên trong cộng tác.
- Biểu đồ cộng tác được dùng để mô hình các *scenario* bên trong một use case hoặc một thao tác liên quan đến sự cộng tác của các đối tượng khác nhau và các tương tác khác nhau.
- Biểu đồ cộng tác được dùng trong các giai đoạn đầu của dự án để xác định các đối tượng (do đó các lớp) tham gia vào một use case.
- Biểu đồ cộng tác đươc dùng để mô tả các đối tượng tham gia trong một mẫu thiết kế.

8.5 Một hình oval đứt nét với tên cộng tác được đặt ở bên trong hoặc bên dưới.

8.6 Là những phụ thuộc <<*trace*>> và <<*realize*>>.

8.7 Bằng cách mô tả các vai trò của các thành viên trong cộng tác như là các vai trò tham số sẽ được thay thế bởi các thành viên thực sự.

8.8 Biểu đồ cộng tác mức đặc tả chỉ ra các vai trò chung và tất cả các tuỳ chọn có thể. Biểu đồ cộng tác mức thể hiện chỉ ra các thể hiện cụ thể của cộng tác trong đó các vai trò được thay thế bởi các thể hiện đối tượng và một nhánh đơn được đặt ngang qua cộng tác.

8.9 Chỉ ra khía cạnh cấu trúc của ngữ cảnh một cộng tác dưới dạng các vai trò lớp và các vai trò kết hợp giữa chúng.

8.10 Tên của các thể hiện đối tượng được gạch chân.

8.11 Dấu '/'

8.12 (a) vai trò. (b) đối tượng. (c) đối tượng. (d) vai trò.

8.13 Là <<*local*>> và <<*parameter*>>.

8.14 Một truyền thông giữa hai đối tượng chuyển thông tin với mong muốn một hành động sẽ xảy ra.

8.15 Thông điệp là đặc tả của *stimulus*, và *stimulus* biểu diễn một thể hiện nào đó của việc gởi thông điệp với các tham số cụ thể.

8.16 Một biến cố, một lời gọi thao tác và việc tạo hoặc huỷ đối tượng.

8.17 Gồm: kiểu thủ tục (đồng bộ), kiểu phẳng, kiểu không đồng bộ và kiểu trả về nơi gọi.

8.18 Biến cố là không đồng bộ.

8.19 Các thao tác đồng bộ có thể có một *giá trị trả về* được gởi trở lại cho đối tượng gởi thông điệp.

8.20 Sequence-term trong một thông điệp gồm một số nguyên hoặc tên với *recurrence* (số lần thực hiện) tuỳ chọn

8.21 Tên được dùng trong các sequence-expression để biểu diễn use case trong đó thông điệp được gởi, để biểu diễn tên đối tượng gởi thông điệp và để đặt nhãn cho cách nhánh rẽ sau một phân nhánh.

8.22 Một phát biểu điều kiện phải được lượng giá là đúng trước khi thông điệp được gởi.

8.23 Là đối tượng chạy trên chính tiểu trình điều khiển của nó.

8.24 Là một hình chữ nhất với biên dày hoặc từ khoá {*active*}.

8.25 Gồm hai hình chữ nhật đặt chồng lên nhau và hơi lệch.

8.26 Chúng là {*new*}, {*destroyed*} và {*transient*}.

8.27 Chúng là <<*become*>> và <<*copy*>>.

8.28 Các bước tạo ra một biểu đồ cộng tác bao gồm:

- Quyết định ngữ cảnh của tương tác: hệ thống, hệ thống con, use case hoặc thao tác.
- Xác định các phần tử cấu trúc (vai trò lớp, đối tượng, hệ thống con) cần thiết để thực hiện chức năng của cộng tác.
- Mô hình các quan hệ cấu trúc giữa các phần tử này để tạo ra một biểu đồ nhằm biểu diễn ngữ cảnh của tương tác.
- Xem xét một số *scenario* thay thế cần thiết
- Vẽ các biểu đồ cộng tác mức thể hiện nếu cần.
- Vẽ biểu đồ cộng tác mức đặc tả để tóm tắt các *scenario* trong biểu đồ tuần tự mức thể hiện, đây là bước tuỳ chọn.

8.29 Chúng là *Actor*, *Worker*, *Case Worker*, *Internal Worker* và *Entity*.

8.30 Dòng công việc phân tích (*Analysis Workflow*) và dòng công việc thiết kế (*Design Workflow*).

8.31 Ba bước trong hoạt động *Analysis a Use Case* (phân tích một use case) là *Identitying Analysis Classes* (xác định lớp phân tích), *Describing Analysis Object Interactions* (mô tả tương tác đối tượng phân tích) và *Capturing Special Requirements* (nắm bắt các yêu cầu đặc biệt)

8.32 Khác với hoạt động *Use Case Description* (mô tả use case) mô tả tương tác giữa hệ thống với actor ngoài, hoạt động *Flow of Events Analysis* (phân tích dòng biến cố) mô tả tương tác giữa các lớp bên trong hệ thống.

8.33 Là cộng tác định nghĩa mẫu tương tác chung giữa các vai trò có thể được áp dụng hoặc đặc tả trong các cộng tác khác.

8.34 Chúng là *xác định các lớp thiết kế tham gia* (Indentifying the Participating Design Classes), *mô tả các tương tác của đối tượng thiết kế* (Describing Design Object Interactions), *xác định các giao diện và các hệ thống con tham gia* (Identifying the Participating Subsystems and Interfaces), *mô tả các tương tác của hệ thống con* (Describing Subsystem Interactions) và *nắm bắt các yêu cầu cài đặt* (Capturing Inplementation Requirments).

Chương 9: BIỂU ĐỒ TUẦN TỰ

9.1 Một dãy các thông điệp được gởi giữa các đối tượng hoặc vai trò trong ngữ cảnh một cộng tác để đạt được chức năng của cộng tác.

9.2 Biểu đồ tuần tự mô hình một tương tác dưới dạng dãy các thông điệp được sắp xếp theo thời gian gởi, còn biểu đồ cộng tác trình bày các thông điệp trong ngữ cảnh quan hệ cấu trúc giữa các vai trò lớp và các thể hiện đối tượng liên quan.

9.3 Thời gian được biểu diễn theo trục thẳng đứng từ trên xuống, khi ấy các thông điệp ở vị trí thấp hơn được gởi muộn hơn (hoặc có thể chọn biểu diễn thời gian hướng từ trái sang phải).

9.4 Biểu đồ tuần tự được dùng để:

- Lập mô hình tương tác ở mức cao giữa các đối tượng hoạt động.
- Lập mô hình tương tác giữa các thể hiện đối tượng bên trong một cộng tác nhằm hiện thực một use case.
- Lập mô hình tương tác giữa các đối tượng bên trong một cộng tác nhằm hiện thực một thao tác.
- Lập mô hình các tương tác tổng thể (biểu diễn tất cả các đường đi có thể có thông qua tương tác) hoặc dùng để xác định các thể hiện của một tương tác (chỉ biểu diễn một đường đi thông qua tương tác).

9.5 Biểu đồ tuần tự thể hiện trình bày một thể hiện cụ thể của tương tác, tương đương với một *scenario* trong một use case. Có nhiều biểu đồ tuần tự thể hiện, mỗi biểu đồ là một nhánh xuyên suốt tương tác.

Biểu đồ tuần tự tổng quát phối hợp các nhánh khác nhau ngang qua tương tác trong một biểu đồ duy nhất.

9.6 Một đường đứt nét treo từ một hình chữ nhật biểu diễn đối tượng.

9.7 Một hình chữ nhật trắng trên đường sinh tồn của đối tượng.

9.8 Bằng cách tô đen vùng *tiêu điểm kiểm soát* (focus of control) khi đối tượng đang thực hiện hoạt động.

9.9 Một mũi tên xuất phát từ đường sinh tồn hoặc vùng kiểm soát của đối tượng đến đối tượng khác (hoặc chính nó), với *chữ ký* của thông điệp được đặt trên mũi tên.

9.10 Một đường đứt nét với đầu mũi tên mở đi từ đối tượng bị gọi trả ngược về đối tượng gọi.

9.11 Là (b) và (c).

9.12 Gồm: kiểu thủ tục (đồng bộ), kiểu phẳng, kiểu không đồng bộ và kiểu trả về nơi gọi.

9.13 Bằng một mũi tên thông điệp đến chính hình chữ nhật biểu diễn đối tượng và đường sinh tồn của đối tượng được bắt đầu ngay lập tức bên dưới hình chữ nhật này.

9.14 Bằng một dấu chữ thập hình chữ x được tô đậm đặt ở cuối đường sinh tồn ngay tại vị trí đối tượng bị huỷ.

9.15 Tên được dùng để biểu diễn use case trong đó thông điệp được gởi, để biểu diễn tên đối tượng gởi thông điệp và để đặt nhãn cho cách nhánh rẽ sau một phân nhánh

9.16 Hoặc một dấu hoa thị, hoặc dấu hoa thị theo sau là một điều kiện được đặt trong dấu ngoặc vuông cho biết thông điệp tiếp tục được lặp lại khi điều kiện vẫn còn đúng.

9.17 Là đối tượng chạy trên chính tiểu trình điều khiển của nó.

9.18 Là một hình chữ nhật với biên dày hoặc từ khoá {*active*}.

9.19 Dùng ký hiệu recurrence (dấu hoa thị và điều kiện) lên thông điệp, hoặc đặt một hình chữ nhật quanh các thông điệp với một điều kiện ở dưới đáy của hình chữ nhật.

9.20 Để cho biết có các ràng buộc về khoảng thời gian cho phép giữa việc gởi và nhận thông điệp, hoặc giữa việc gởi một thông điệp và nhận một thông điệp khác, thường là một đáp ứng.

9.21 Các bước tạo ra một biểu đồ tuần tự bao gồm:

- Quyết định ngữ cảnh của tương tác: hệ thống, hệ thống con hoặc thao tác.

- Xác định các thành phần có cấu trúc (lớp hoặc đối tượng) cần thiết để thực hiện chức năng của use case hoặc thao tác.
- Xem xét các *scenario* thay thế.
- Vẽ các biểu đồ thể hiện:
  - Đặt các đối tượng từ trái sang phải.
  - Bắt đầu bằng thông điệp khởi đầu sự tương tác, đặt các thông điệp trên trang theo chiều từ trên xuống dưới. Biểu diễn các đặc tính của các thông điệp cần thiết để giải thích ngữ nghĩa của tương tác.
  - Thêm *tiêu điểm kiểm soát* nếu cần thiết để trực quan hoá các hành động lồng nhau hoặc thời điểm hoạt động đang diễn ra.
  - Thêm các ràng buộc thời gian nếu cần.
  - Thêm các giải thích vào biểu đồ nếu cần, ví dụ điều kiện đầu và điều kiện cuối.
- Nếu cần hãy vẽ một biểu đồ tổng quát để tóm tắt tất cả các trường hợp trong các biểu đồ thể hiện.

9.22 Là dòng công việc thiết kế (*Design Workflow*) và dòng công việc kiểm tra (*Test Workflow*).

9.23 Chúng là: *xác định các lớp thiết kế tham gia* (Indentifying the Participating Design Classes), *mô tả các tương tác của đối tượng thiết kế* (Describing Design Object Interactions), *xác định các giao diện và các hệ thống con tham gia* (Identifying the Participating Subsystems and Interfaces), *mô tả các tương tác của hệ thống con* (Describing Subsystem Interactions) và *nắm bắt các yêu cầu cài đặt* (Capturing Inplementation Requirments).

9.24 Bốn bước trong hoạt động *thiết kế kiểm tra* bao gồm: *thiết kế các trường hợp kiểm tra tích hợp* (Designing Integration Test Cases), *thiết kế các trường hợp kiểm tra hệ thống* (Designing System Test Cases), *thiết kế các trường hợp kiểm tra hồi qui* (Designing Regression Test Cases) và *xác định và lập cấu trúc các thủ tục kiểm tra* (Identifying and Structuring Test Procedures).

Chương 10: BIỂU ĐỒ HOẠT ĐỘNG

10.1 Biểu đồ hoạt động thường được dùng để mô tả các dòng nghiệp vụ (qui trình nghiệp vụ), các dòng bên trong use case, các dòng giữa các use case và các dòng bên trong các thao tác phức tạp.

10.2 Biểu đồ hoạt động có thể được dùng đầu tiên trong phân tích nghiệp vụ, như là một cách tạo ra các use case nghiệp vụ, hoặc để hỗ trợ một phương thức của dòng nghiệp vụ khác.

10.3 Biểu đồ hoạt động là một đồ thị hai chiều minh hoạ các dòng phức tạp được thực hiện bên trong một hệ thống, là hệ thống nghiệp vụ hoặc hệ thống máy tính. Chúng được dùng để diễn tả các hệ thống theo chuyên môn của những người nắm các quy tắc quản lý hệ thống (các *stakeholder*).

10.4 Một hoạt động là một đơn vị công việc.

10.5 Một hành động là một phần tử công việc mà hoạt động thực hiện.

10.6 Có thể xảy ra ngay lập tức hoặc mất một khoảng thời gian dài.

10.7 Hành động xảy ra khi bắt đầu hoạt động, trong suốt thời gian thực hiện hoạt động, khi đáp ứng một biến cố, hoặc khi kết thúc hoạt động.

10.8 Chuyển tiếp là sự di chuyển giữa các hoạt động và/hoặc các trạng thái.

10.9 Một chuyển tiếp xảy ra khi hoạt động hoàn tất. Chuyển tiếp có thể phát sinh khi một biến cố xảy ra.

10.10 Khi hoạt động kết thúc.

10.11 Một trạng thái tương tự một hoạt động, nhưng nó được dùng để chỉ định dòng công việc đang đợi biến cố kích hoạt. Một hoạt động kết thúc khi công việc hoàn tất mà không cần kích hoạt, trong lúc trạng thái chỉ kết thúc khi có biến cố kích hoạt.

10.12 Cú pháp: *event(arguments) [condition] / action ^target.sendEvent (arguments)*

10.13 Điểm quyết định là một điểm trên dòng công việc, tại đó sự chuyển tiếp ra khỏi hoạt động hoặc trạng thái có thể đi theo một số hướng tùy chọn phụ thuộc vào một điều kiện.

10.14 Là một điều kiện trên một chuyển tiếp. Chuyển tiếp thực hiện chỉ khi điều kiện này thoả mãn.

10.15 Một hoặc nhiều.

10.16 Hai hoặc nhiều. Mỗi chuyển tiếp đi vào phải có điều kiện, các điều kiện trên tất cả các chuyển tiếp đi ra phải không được chồng chéo lên nhau.

10.17 Swimlane là một cột trong biểu đồ hoạt động, dùng để chỉ cho biết đơn vị tổ chức hoặc kỹ thuật thực hiện hoạt động.

10.18 Swimlane có ích trong việc định vị cho một hoạt động tại một đơn vị tổ chức hoặc một vùng kỹ thuật.

10.19 Thanh đồng bộ hóa được dùng để phân chia một chuyển tiếp thành các dòng song song, hoặc trộn nhiều chuyển tiếp song song thành một chuyển tiếp đơn để đồng bộ hoá sự kết thúc các dòng song song.

10.20 Một chuyển tiếp đi vào và nhiều chuyển tiếp đi ra, hoặc nhiều chuyển tiếp đi vào và một chuyển tiếp đi ra.

10.21 Các đối tượng có thể được đặt trong biểu đồ hoạt động, dùng để chỉ định hoạt động ảnh hưởng đến trạng thái của đối tượng như thế nào. Điều này được gọi là dòng đối tượng.

10.22 Thông tin về trạng thái của đối tượng.

10.23 Một số hoạt động có các dòng công việc phức tạp nên cần chi tiết chúng với một biểu đồ hoạt động riêng.

10.24 Biểu đồ hoạt động có thể gắn với use case nghiệp vụ, use case hệ thống, hoạt động, đối tượng và thao tác.

10.25 Các biểu tượng điều khiển là các biểu tượng trực quan chỉ định vị trí trên biểu đồ, ở đó các biến cố sự phát sinh hoặc đáp ứng. Chúng là ký hiệu bổ sung, không nhất thiết cần, để việc kiểm soát các biến cố được tường minh hơn.

Chương 11: BIỂU ĐỒ TRẠNG THÁI

11.1 Biểu đồ trạng thái dùng để mô hình hoá các trạng thái mà các thực thể trong hệ thống có thể nhận và mô hình hoá sự thay đổi trạng thái của chúng, được định nghĩa bằng cách chuyển tiếp từ trạng thái này sang trạng thái khác.

11.2 Trạng thái là một điều kiện mà một đối tượng có thể bước vào và ở đó trong một quãng thời gian.

11.3 Không xác định, nhưng có thể rất ngắn.

11.4 Biến cố có thể phát sinh một hành động trong một trạng thái hoặc phát sinh việc kết thúc trạng thái.

11.5 Biểu đồ trạng thái thường được dùng cho đối tượng, cũng có thể dùng cho actor, use case và đôi khi cho thao tác.

11.6 Không.

11.7 Năm loại hành động trên một trạng thái bao gồm:

- *On Entry*: các hành động được phát sinh ngay khi trạng thái bắt đầu.
- *Do*: các hành động được phát sinh trong suốt thời gian tồn tại của trạng thái.

- *On Event*: các hành động được phát sinh để đáp ứng một biến cố.
- *On Exit*: các hành động được phát sinh trước khi trạng thái kết thúc.
- *Include*: gọi một *máy trạng thái con*, được biểu diễn bởi một biểu đồ trạng thái khác.

11.8 Cú pháp của một hành động có dạng *action-label* / *action*, trong đó *action* là một trong số *entry*, *do*, *exit*, *include* hoặc tên biến cố.

11.9 Có dạng *event-name (parameters) [guard-condition]* / *action*, trong đó *parameters* là danh sách tham số cách nhau bởi dấu phẩy, *guard-condition* là điều kiện, điều kiện này phải thoả để biến cố phát sinh hành động.

11.10 Chia trạng thái thành hai ngăn, liệt kê các hành động trong ngăn thứ hai.

11.11 Một hoặc không.

11.12 Trạng thái phức là trạng thái được thấy có các trạng thái con.

11.13 Hoặc dùng các biểu đồ riêng, hoặc vẽ các dòng con lồng bên trong trạng thái.

11.14 Một trạng thái phức có thể có các dòng song song. Như vậy một trạng thái phức có thể ở trong nhiều trạng thái con trong cùng một thời điểm.

11.15 Thanh đồng bộ hóa cho phép phân chia một biểu đồ để biểu diễn các dòng song song giữa các trạng thái và rồi nối lại các dòng song song này.

11.16 Khi ấy một hành động *entry* (bước vào trạng thái) được phát sinh. Dòng con được bắt đầu ngay tại trạng thái bắt đầu của nó, hoặc nếu có các dòng đồng hành thì tất cả chúng đều được bắt đầu trạng thái bắt đầu của chúng. Ngoại trừ khi có một trạng thái lịch sử (history) thì các dòng con bắt đầu trở lại tại điểm chúng thoát ra lần trước.

11.17 Tất cả các trạng thái con đều kết thúc bằng cách thực hiện hành động *exit*. Trạng thái phức cũng kết thúc và cũng thực hiện hành động *exit* của nó.

11.18 Dòng con được bắt đầu tại trạng thái con do chuyển tiếp chỉ định. Hành động *entry* của trạng thái phức và trạng thái con nói trên được thực hiện.

11.19 Một chuyển tiếp được phát sinh bởi một biến cố có thể đi theo một số tuyến phụ thuộc một điều kiện. Một điểm quyết định được đưa ra để chỉ định các tuyến có thể chọn.

11.20 Trạng thái *history* chứa H nghĩa là một trạng thái phức bắt đầu lại tại dòng ở mức ngoài nhất tại chỗ nó kết thúc trước đó, còn *H cho biết trạng thái phức bắt đầu lại tại chỗ nó kết thúc trước đó, dù điểm này nằm ở mức lồng nhau phía sâu bên trong.

11.21 Trạng thái đồng bộ là trạng thái được chia sẻ giữa các dòng song song trong một trạng thái phức. Nó cho phép các dòng song song xảy ra đồng thời.

Chương 12: NGÔN NGỮ RÀNG BUỘC ĐỐI TƯỢNG

12.5 Ràng buộc là một luật về các giá trị trong mô hình, được biểu diễn như một giới hạn trên các giá trị này.

12.6 Chúng được dùng để định nghĩa hành vi của mô hình và quyền sử dụng hợp pháp.

12.7 OCL là ngôn ngữ ràng buộc đối tượng, một ngôn ngữ định nghĩa chính xác các ràng buộc.

12.8 Ngôn ngữ tự nhiên thường kém rõ ràng, trong lúc các các hành vi của mô hình cần được mô tả rõ ràng.

12.9 Pre-condition (điều kiện đầu) là một phát biểu định nghĩa điều kiện hợp pháp để một thao tác hoặc một use case có thể thực hiện chức năng của nó.

12.10 Post-condition (điều kiện đầu) là một phát biểu về mô hình, nó định nghĩa trạng thái của mô hình sau khi một thao tác hoặc một use case hoàn tất, phù hợp với điều kiện đầu.

12.11 Invariant là tính chất của đối tượng hoặc thao tác phải luôn đúng trong bất kỳ thời điểm nào.

12.12 Thiết kế gọn bao gồm các đặc tả hành vi của hệ thống dưới dạng điều kiện đầu, điều kiện cuối.

12.13 Ngữ cảnh của một ràng buộc là lớp hoặc thao tác.

12.14 Các kiểu cơ bản trong một biểu thức OCL bao gồm *boolean*, *integer*, *real* và *string*.

12.15 Ngoài ra còn có các kiểu tập và các kiểu được định nghĩa trong mô hình.

12.16 Trị là 1.

12.17 Gồm *tập* (set), *dãy* (sequence)và *gói* (bag).

12.18 *a->first*

12.19 Là *set* { 2, 4, 7, 3, 8, 9}

12.20 Kết quả là *bag* {2, 2, 3, 3, 4} -> *select*(x | x<5).

12.21 Kết quả là *true*.

12.22 Hiếm khi. Tại thời điểm định nghĩa use case, các đối tượng không được định nghĩa một cách thích đáng.

Chương 13: BIỂU ĐỒ CÀI ĐẶT

13.1 *Thành phần* là một phần tử vật lý của hệ thống, thông thường là một trong một số loại tập tin. *Thành phần* có thể là một tệp nguồn, được dùng để đưa ra sản phẩm phần mềm cho hệ thống, hoặc một phần tử của hệ thống thực thi.

13.2 Nút là một bộ xử lý trong cài đặt một hệ thống, xuất hiện trong biểu đồ triển khai.

13.3 Liên kết truyền thông là một kết hợp giữa các nút trong biểu đồ triển khai, thường được định khuôn với cơ chế hoặc nghi thức truyền thông.

13.4 Ký hiệu cho một thành phần gồm một hình chữ nhật với hai hình chữ nhật hẹp phủ lên cạnh trái, tên của thành phần được đặt trong hình chữ nhật lớn.

13.5 Ký hiệu một nút là một hình lập phương với tên nút ở góc trên bên trái.

13.6 Biểu đồ thành phần được dùng để:

- Mô hình vật lý các thành phần phần mềm và các mối quan hệ giữa chúng.
- Mô hình các tập tin mã nguồn và mối quan hệ giữa chúng.
- Mô hình cấu trúc của phiên bản phần mềm.
- Xác định các tập tin được biên dịch thành tập tin thi hành.

13.7 Biểu đồ triển khai được dùng để:

- Mô hình vật lý các thành phần phần cứng và các kênh liên lạc giữa chúng.
- Lập kế hoạch kiến trúc của một hệ thống.
- Lập tài liệu việc triển khai các thành phần phần mềm trên các *nút* phần cứng.

13.8 Các thành phần được mô tả trong biểu đồ triển khai để mô hình việc triển khai các *thành phần thực thi* trên các bộ xử lý trong hệ thống dự kiến. Chúng là các thể hiện thông thường của các thành

phần. Trong lúc các thành phần trong biểu đồ thành phần thường là các tập tin nguồn hoặc các phần tử khác của hệ thống phần mềm.

13.9 Một mũi tên đứt nét với đầu mũi tên mở.

13.10 Một là các icon được định khuôn có thể được dùng thay cho các icon chuẩn và hai là khuôn dạng được đặt trong cặp dấu <<...>> và được đặt ở bên trên tên của thành phần.

13.11 Một là các icon được định khuôn có thể được dùng thay cho các icon chuẩn dành cho nút và hai là các liên kết truyền thông có thể được định khuôn với nghi thức truyền thông và được đặt trong cặp dấu <<...>>.

13.12 Hiển thị thông tin về mô hình, như số phiên bản của các thành phần.

13.13 Các bước chính để tạo một biểu đồ thành phần là:

1. Xác định mục đích của biểu đồ.
2. Thêm các thành phần vào biểu đồ, nhóm chúng lại trong các thành phần khác nếu thích hợp.
3. Thêm các phần tử khác vào biểu đồ, như lớp, đối tượng hoặc giao diện.
4. Thêm các phụ thuộc giữa các phần tử của biểu đồ.

13.14 Nêu những bước chính để tạo một biểu đồ triển khai.

1. Quyết định mục đích của biểu đồ.
2. Thêm các nút vào biểu đồ.
3. Thêm các kết hợp truyền thông vào biểu đồ.
4. Thêm các phần tử khác vào biểu đồ, chẳng hạn như các thành phần hoặc các đối tượng hoạt động, nếu cần.
5. Thêm các phụ thuộc giữa các thành phần và đối tượng, nếu cần.

13.15 Là dòng công việc cài đặt (*Implementation Workflow*).

13.16 Là dòng công việc thiết kế (*Design Workflow*) và dòng công việc cài đặt (*Implementation Workflow*).

Chương 14: CÁC QUI ƯỚC KÝ HIỆU CHUNG CỦA UML

14.1 Là *nút* (node) và *đường* (path). Các *đường* nối các *nút* lại với nhau.

14.2 Ví dụ biểu đồ use case, biểu đồ lớp, biểu đồ cộng tác, biểu đồ thành phần, biểu đồ triển khai.

14.3 Chúng là quan hệ *connection* (kết nối), quan hệ *containment* (vật chứa) và quan hệ *visual attachment* (đính kèm).

14.4 (a) Đó là *biểu tượng* (icon), *biểu tượng hai chiều* (two-dimensional symbol), *đường* (path) và *chuỗi* (string)

(b) + Một *biểu tượng* là một phần tử đồ hoạ có hình dạng và kích thước cố định.

+ Một *biểu tượng hai chiều* được sử dụng để chứa các biểu tượng khác và sẽ thay đổi kích thước một cách thích hợp. Chúng có thể được chia thành các ngăn riêng.

+ Một *đường* dùng để nối các *biểu tượng hai chiều* và các *biểu tượng* lại với nhau.

+ Một *chuỗi* dùng để mô tả một dãy thông tin dạng văn bản.

(c) *Biểu tượng* ví dụ như *actor*; *biểu tượng hai chiều* ví dụ như *class*; *đường* ví dụ như *kết hợp* và *chuỗi* ví dụ như *tên của kết hợp*.

14.5 Tên định danh phần tử mô hình còn nhãn được gắn kèm để cung cấp thêm thông tin.

14.6 Như khuôn dạng trong cặp dấu <<...>>.

14.7 Ví dụ <<*include*>>, <<*realize*>>.

14.8 Các chi chú được hiển thị trong một hình chữ nhật bẻ góc và được gắn với phần tử cần chú thích bằng đường đứt nét.

14.9 *Classifier* là một thuật ngữ dùng cho các phần tử mô hình mô tả đặc trưng về *hành vi* và *cấu trúc* trong một mô hình.

14.10 Các *classifier* trong UML bao gồm *lớp*, *actor*, *use case*, *kiểu dữ liệu*, *thành phần*, *giao diên*, *tính hiệu*, *nút xử lý* và *hệ thống con*.

14.11 Là trạng thái, thuộc tính và thao tác.

14.12 Ký hiệu của

(a) Tên lớp là *Classname*.

(b) Tên của *thể hiện vô danh* của lớp là *:Classname*.

(c) Tên của *thể hiện* của lớp là *objectname*:*Classname*.

(d) Tên của *thể hiện* mà không biết lớp của nó là *objectname*.

(e) Tên của vai trò là /*Rolename*:*Classname*.

(f) Tên của *thể hiện* có vai trò là *objectname*/*Rolename*:*Classname*.

14.13 *Stereotype* (khuôn dạng) là phương tiện đặc tả một phần tử phù hợp với hành vi được hiểu rõ ràng, hoặc của một thực thể có thật. *Stereotype* được đặt trong cặp <<...>> hoặc dưới dạng một icon.

14.14 Một *stereotype* là một lớp *siêu mô hình* được đưa ra cho phép mô hình một loại lớp đặc biệt trong hệ thống.

14.15 Một là icon dứng riêng; hai là đặt kèm tên khuôn dạng trong cặp dấu <<....>> phía trên tên lớp; và cuối cùng là kèm icon với tên lớp.

14.16 Các lớp trong các mở rộng cho web của Bruce Conallen, chẳng hạn lớp <<*ServerPage*>>.

14.17 Mũi tên thông điệp được định khuôn dùng trong biểu đồ tuần tự.

14.18 Giá trị thẻ gồm một cặp có dạng {*tag* = *value*}, được liên kết với một phần tử mô hình (ví dụ *class*). Ở đây *tag* là tên của một tính chất và *value* là một giá trị nào đó được biểu diễn như xâu ký tự.

14.19 Các *tagged value* được biểu diễn thành cặp {*tag* = *value*} đặt dọc theo phần tử mô hình được áp dụng.

14.20 Ràng buộc định nghĩa một quan hệ giữa các phần tử mô hình hoặc trên một phần tử mô hình phải thoả, nếu không hệ thống buộc phải dừng.

14.21 Các ràng buộc, thường là các biểu thức OCL, được đặt trong cặp móc nhọn {...} và được đặt dọc theo phần tử mô hình. Trường hợp ràng buộc áp dụng trên quan hệ giữa hai đối tượng nối với nhau bởi một phụ thuộc thì ràng buộc được đặt dọc theo phụ thuộc này. Các ràng buộc cũng được đặc tả bên trong một nút (node).

14.22 Ví dụ {*age* <= 21 *years*}.

14.23 Là ngôn ngữ ràng buộc đối tượng OCL.

Chương 15: CÔNG CỤ CASE

15.1 CASE là viết tắt của *Computer Assisted Software Engineering* hoặc *Computer Aided System Engineering*.

15.2 Không. Các công cụ CASE có thể hỗ trợ cho tất cả các khía cạnh của qui trình công nghệ phần mềm, chứ không phải chỉ dành riêng cho UML. Các công cụ CASE cũng hợp nhất các ký hiệu mô hình khác.

15.3 UML cung cấp ký hiệu và các ngữ nghĩa chung cho các mô hình được phát triển trong các công cụ CASE khác nhau, và cung cấp một chuẩn biến đổi để các mô hình có thể được chuyển đổi giữa các công cụ.

15.4 XMI có nghĩa là *XML Metadata Interchange* (bộ chuyển đổi siêu dữ liệu XML), còn XML là *Extensible Markup Language* (ngôn ngữ đánh dấu mở rộng). XMI là một chuẩn cho ngôn ngữ văn bản để

mô tả các mô hình UML. Nó được định nghĩa để có thể chuyển đổi mô hình giữa các công cụ CASE khác nhau.

15.5 Một nơi chứa (*repository*) là nơi lưu tất cả các phần tử mô hình.

15.6 Giá trị thẻ gồm một cặp có dạng {*tag* = *value*}, được liên kết với một phần tử mô hình (ví dụ *class*). Ở đây *tag* là tên của một tính chất và *value* là một giá trị nào đó được biểu diễn như xâu ký tự.

15.7 Các giá trị thẻ là quan trọng bởi vì nó cho phép mở rộng UML.

15.8 *Round trip engineering* là nơi công cụ CASE có thể phát sinh mã nguồn và chuyển ngược mã nguồn, cũng như quản lý các cập nhật lên mô hình và lên mã nguồn chương trình.

Chương 16: MẪU THIẾT KẾ

16.1 Mẫu thiết kế là giải pháp cho bài toán chung trong thiết kế hệ thống máy tính. Nó là một tài liệu có giá trị để các nhà phát triển áp dụng để giải quyết vấn đề cùng loại.

16.2 Framework thì toàn diện hơn mẫu và là một áp dụng riêng phần có thể được sử dụng cho một lĩnh vực riêng.

16.3 Một *idiom* là một tập các chỉ dẫn về cách cài đặt các khía cạnh của một hệ thống phần mềm viết bằng một ngôn ngữ cụ thể.

16.4 Trong một *catalogue* mẫu.

16.5 Chúng gồm

- *Name* – tên của *mẫu*, mô tả ý tưởng giải pháp theo một số cách.
- *Problem* – vấn đề mà *mẫu* giúp giải quyết.
- *Context* – ngữ cảnh ứng dụng của mẫu (kiến trúc hoặc nghiệp vụ) và các yếu tố chính để mẫu làm việc thành công trong một tình huống nào đó.
- *Force* – các ràng buộc, các vấn đề phải được giải quyết bởi mẫu.
- *Solution* – giải pháp để cân đối các ràng buộc xung đột và làm cho hợp với ngữ cảnh.
- *Sketch* – bản phác thảo tượng trưng của các ràng buộc và cách giải quyết chúng.
- *Resulting context* – ngữ cảnh sau khi được thay đổi bởi giải pháp.
- *Rationale* – lý do và động cơ cho mẫu.

16.6 Chúng là các loại mẫu *creational*, *structural* và *behavioural*.

16.7 Ý nghĩa của các loại mẫu thiết kế như sau:

- *Creational* – liên quan với việc tạo ra các thể hiện đối tượng, tách biệt với cách được thực hiện từ ứng dụng.
- *Structural* – liên quan đến các quan hệ cấu trúc giữa các thể hiện, dùng *generalization*, *aggregation* và *composition*.
- *Behavioural* – liên quan đến việc gán trách nhiệm để cung cấp chức năng giữa các đối tượng trong hệ thống.

16.8 Một hình oval đứt nét với một hình chữ nhật đứt nét phủ lên góc phần tư phía trên bên phải hình oval. Tên của cộng tác mẫu được đặt trong hình oval còn tên các tham số được đặt trong hình chữ nhật.

16.9 Bốn loại biểu đồ UML có thể được dùng để mô hình một mẫu là cộng tác mẫu, biểu đồ cộng tác, biểu đồ lớp và biểu đồ tuần tự.

16.10 Các bước do Gamma và các cộng sự gợi ý bao gồm:

1. Đọc mẫu để có một cái nhìn toàn cảnh.
2. Học chi tiết cấu trúc, các đối tượng tham gia và các cộng tác của mẫu.
3. Kiểm tra mã nguồn mẫu để xem ví dụ về cách dùng mẫu.
4. Chọn tên cho các đối tượng tham gia của mẫu (ví dụ như lớp) sao cho có ý nghĩa đối với ứng dụng.
5. Định nghĩa các lớp.
6. Chọn tên cho các thao tác.
7. Cài đặt các thao tác thực hiện các trách nhiệm và cộng tác trong mẫu.

PHỤ LỤC C
TỪ ĐIỂN THUẬT NGỮ

Action (*hành động*): Một phát biểu có thể thi hành được, thường được liên kết với một hoạt động, trạng thái hoặc chuyển tiếp.

Activation (*vùng hoạt động*): Vùng hình chữ nhật trên đường sinh tồn của đối tượng trong biểu đồ tuần tự dùng để mô tả khi nào đối tượng đang thi hành một thao tác.

Activity (*hoạt động*): Một bước trong dòng công việc dùng để biểu diễn nơi công việc đang được thực hiện bên trong nghiệp vụ hoặc hệ thống. Hoạt động tiếp tục cho đến khi công việc hoàn tất, hoặc một biến cố làm nó kết thúc.

Activity Diagram (*biểu đồ hoạt động*): Phương tiện mô tả dòng công việc, liên kết các hoạt động và trạng thái, được dùng để mô tả các dòng công việc của nghiệp vụ hoặc hệ thống.

Actor (*tác nhân*): Người dùng hoặc hệ thống ngoài giao tiếp với hệ thống hoặc hệ thống con đang mô hình và được trình bày trong biểu đồ use case.

Aggregation (*kết tập*): Đặc tả tính tự nhiên của kết hợp *whole-part*, một lớp là bộ phận của một lớp khác.

Ancestor (*tiền bối*): Một lớp *cha* của một lớp bất chấp cấp tổng quát hoá.

Association (*kết hợp*): Một quan hệ giữa hai hoặc nhiều lớp, đặc tả các lớp liên quan cộng tác đến các lớp khác.

Association Class (*lớp kết hợp*): Biểu diễn kết hợp như một lớp. Điều này cho phép kết hợp có trách nhiệm với dữ liệu và chức năng (nghĩa là nó có thuộc tính và thao tác).

Attribute (*thuộc tính*): Mục dữ liệu mà một lớp có trách nhiệm.

Behaviour Specification (*đặc tả hành vi*): Mô tả hành vi được cung cấp bởi một use case.

Bound Element (*phần tử bị buộc*): Một lớp được đặc tả là thể hiện của một lớp tham số.

Class (*lớp*): Kiểu chung biểu diễn một họ các đối tượng có trách nhiệm với các mục dữ liệu và chức năng chung.

Class Diagram (*biểu đồ lớp*): Mô tả cấu trúc tĩnh của một tập các lớp. Cấu trúc này gồm (không nhất thiết tất cả) các lớp, thuộc tính, thao tác, kết hợp và tổng quát hoá.

Class-scope [*feature*] (*phạm vi lớp*): Một thuộc tính hoặc thao tác tồn tại chỉ trên lớp, không phải trên thể hiện của lớp. Một thuộc tính phạm vi lớp giữ cùng một giá trị đối với tất cả các thể hiện của lớp.

Classifier: Một thuật ngữ dùng cho các phần tử mô hình mô tả đặc trưng về hành vi và cấu trúc trong một mô hình. Chúng bao gồm lớp, tác nhân, use case, kiểu dữ liệu, thành phần, giao diện, tín hiệu, nút và hệ thống con.

Collaboration (*cộng tác*): Một tập các thành viên, các đối tượng và các vai trò, làm việc cùng nhau để đạt được một kết quả có nghĩa trong ngữ cảnh của hệ thống.

Collaboration Diagram (*biểu đồ cộng tác*): Biểu đồ trình bày các thành viên trong cộng tác, cùng với tương tác giữa chúng dưới dạng các thông điệp, các kích thích được gởi giữa chúng.

Communication Association (*kết hợp truyền thông*): Kết hợp giữa các nút trong biểu đồ triển khai, thường được định khuôn với cơ chế hoặc nghi thức truyền thông.

Component (*thành phần*): Một phần tử vật lý của hệ thống, thông thường là một số loại tập tin. *Thành phần* có thể là một tệp nguồn, được dùng để đưa ra sản phẩm phần mềm cho hệ thống, hoặc một phần tử của hệ thống thực thi.

Component Diagram (*biểu đồ thành phần*): Biểu đồ mô tả các thành phần và các quan hệ giữa chúng.

Composition (*hợp thành*): Quan hệ *whole-part* chặt chẽ hơn so với kết tập, ở đây lớp bộ phận (part) cùng gắn kết với lớp toàn thể (whole).

Constraint (*ràng buộc*): Một hạn chế lên mô hình được biểu diễn như một điều kiện, được dùng cho điều kiện đầu, điều kiện cuối và các bất biến.

Control Icon (*biểu tượng điều khiển*): Phương tiện làm cho việc gởi biến cố trên một chuyển tiếp được tường minh hơn.

Data Token (*dấu hiệu dữ liệu*): Biểu tượng biểu diễn dữ liệu gởi theo thông điệp trong biểu đồ tương tác.

Decision Point (*điểm quyết định*): Một điểm trên dòng công việc, tại đó một chuyển tiếp có thể rẽ nhánh tuỳ điều kiện.

Dependency (*phụ thuộc*): Một quan hệ kèm khuôn dạng giữa hai phần tử. Khuôn dạng mô tả tính tự nhiên của quan hệ, thường thì chúng phản ánh hướng xử lý, ví dụ <<*realize*>>, <<*refine*>>.

Deployment Diagram (*biểu đồ triển khai*): Biểu đồ trình bày cài đặt của hệ thống thực hiện dưới dạng các nút (xử lý), mối kết hợp truyền thông giữa các nút, và các thành phần được triển khai trên các nút.

Design by Contract (*thiết kế gọn*): Thiết kế dùng các điều kiện đầu và cuối để đặc tả hành vi của các bộ phận của một hệ thống.

Enumeration Class (*lớp liệt kê*): Một lớp định nghĩa một tập các phần tử.

Event (*biến cố*): Một sự xuất hiện đáng chú ý tại một thời điểm nào đó. Trong các biểu đồ hoạt động và trạng thái, biến cố được dùng cho các phát sinh chuyển tiếp.

Expression (*biểu thức*): Một xâu ký tự có thể được lượng giá để cho ra một kết quả.

[*Class*] *Feature* (*đặc trưng*): Một thuộc tính, thao tác hay khái niệm khác do người dùng định nghĩa (vd biến cố) mà một lớp phải chịu trách nhiệm về nó.

Generalization (*tổng quát hoá*): Sự trừu tượng các thuộc tính chung của các phần tử (thường là lớp) thành một kiểu *cha* thích hợp.

Guard (*bảo vệ*): Một điều kiện phải thoả để một chuyển tiếp xảy ra.

Icon (*biểu tượng*): Biểu diễn đồ họa của một phần tử được định khuôn.

Instance (*thể hiện*): Một xuất hiện của một lớp.

Instance-scope [*feature*] (*phạm vi thể hiện*): Phạm vi thông thường của một thuộc tính, thao tác. Một thuộc tính mà giá trị của nó chỉ liên quan đến một thể hiện hoặc một thao tác mà phương thức của nó làm việc trên các thuộc tính cùng thể hiện.

Interaction Diagram (*biểu đồ tương tác*): Thuật ngữ chung của biểu đồ cộng tác và biểu đồ tuần tự, được dùng để mô tả tương tác giữa các đối tượng hoặc vai trò cộng tác.

Interface (*giao diện*): Cách tổ chức một tập các *chữ ký thao tác* và được cài đặt bởi một phần tử mô hình như lớp chẳng hạn.

Invariant (*bất biến*): Một điều kiện trên thao tác hoặc use case luôn luôn đúng.

Lifeline (*đường sinh tồn*): Đường trong biểu đồ tuần tự để chỉ cho biết sự có mặt của một đối tượng hoặc vai trò tham gia vào tương tác.

Link (*liên kết*): Một xuất hiện (thể hiện) của kết hợp.

Message (*thông điệp*): Đặc tả sự liên lạc giữa các đối tượng hoặc vai trò trong một cộng tác. Được xác định qua tên, kiểu của các tham số và kiểu của giá trị trả về.

Method (*phương thức*): Cài đặt của thao tác.

Model (*mô hình*): Sự trừu tượng hoá của một hệ thống vật lý cho mục đích đặc tả. *Mô hình* được dùng trong nhiều khung nhìn hệ thống, nó được phát triển ở các giai đoạn khác nhau của dự án.

Multiplicity (*bản số*): số các thể hiện có thể tìm thấy trong phần tử mô hình. Một ví dụ thường thấy là bản số ở đầu mút kết hợp của một lớp, cho biết có bao nhiêu thể hiện của một lớp có thể kết hợp với lớp này.

N-ary Association (*kết hợp n-ngôi*): Một kết hợp liên kết nhiều hơn hai lớp lại với nhau.

Node (*nút*): Bộ xử lý trong cài đặt một hệ thống, xuất hiện trong biểu đồ triển khai.

Note (*ghi chú*): Một xâu văn bản gắn với phần tử mô hình cung cấp thêm thông tin về nó.

Object (*đối tượng*): Thể hiện của lớp.

OCL (*Object Constraint Language – ngôn ngữ ràng buộc đối tượng*): Một ngôn ngữ hình thức biểu diễn ràng buộc.

Operation (*thao tác*): Chức năng hoặc hành vi của một lớp.

Operation Signature (*chữ ký của thao tác*): Tham chiếu chung đến tên, danh sách tham số và kiểu trả về của một thao tác.

Package (*gói*): Cơ chế tổ chức các khung nhình khác nhau của một dự án hoặc hệ thống.

Parameter (*tham số*): Một thuộc tính hoặc đối tượng được truyền như đối số cho thao tác.

Parameterized Class (*lớp tham số*): Một định nghĩa lớp chứa các thuộc tính và thao tác được mô tả làm tham số phải được truyền để tạo thể hiện. Còn được biết với thuật ngữ *template*. Xem thêm *Bound Element*.

Parent Class (*lớp cha*): Là lớp tổng quát hoá trực tiếp.

Pattern (*mẫu*): Một giải pháp được sưu liệu cho vấn đề chung trong thiết kế hệ thống máy tính.

Post-condition (*điều kiện cuối*): Một điều kiện buộc phải đúng nếu một thao tác hoặc một use case được áp dụng hợp lệ.

Pre-condition (*điều kiện đầu*): Một điều kiện cần phải thoả để một thao tác được thực thi hợp lệ, hoặc để một use case được áp dụng hợp lệ.

Relationship (*mối quan hệ*): Tên chung của các loại kết nối giữa hai phần tử mô hình bao gồm kết hợp, tổng quát hoá và phụ thuộc.

Repository (*kho chứa*): Một phương tiện lưu các phần tử UML, các biểu đồ và các tài liệu liên quan.

Role (*vai trò*): Một tên hành vi của một phần tử trong một ngữ cảnh nào đó, thường là phần được giao cho đối tượng phải đóng vai trong một kết hợp hoặc cộng tác.

Sequence Diagram (*biểu đồ tuần tự*): Là biểu đồ tương tác mô tả tính tuần tự của thao tác theo thời gian.

Signal (*tín hiệu*): Một thông điệp không đồng bộ được gởi giữa các thể hiện trong một cộng tác.

Specialization (*chuyên biệt hoá*): Là quá trình ngược với tổng quát hoá. Một lớp chuyên biệt hoá khác với lớp cha và các lớp anh em về thuộc tính, thao tác cũng như phương thức.

State (*trạng thái*): Thuật ngữ chung dành cho các giá trị thuộc tính của đối tượng và liên kết của nó với các đối tượng khác trong thời điểm cụ thể. Đối tượng có thể có nhiều trạng thái tại một thời điểm. Các trạng thái được kết thúc bởi biến cố.

Statechart Diagram (*biểu đồ trạng thái*): Phương tiện mô tả cách một phần tử chuyển từ trạng thái này sang trạng thái khác. Thay đổi trạng thái là kết quả của chuyển tiếp.

Stereotype (*khuôn dạng*): Trong UML, *khuôn dạng* là phương tiện đặc tả một phần tử phù hợp với hành vi được hiểu rõ ràng, hoặc của một thực thể có thật. *Stereotype* được đặt trong cặp <<...>> hoặc dưới dạng một icon.

Stimulus (*kích thích*): Một thể hiện của thông điệp.

Subclass (*lớp con*): Một lớp chuyên biệt hoá trong cấu trúc tổng quát hoá.

Subflow (*dòng con*): Một hoạt động hoặc trạng thái có thể được ngắt thành các *dòng con*.

Subsystem (*hệ thống con*): Trong cơ chế trừu tượng hoá, môt hệ thống có thể được phân cấp bằng cách chia thành các *hệ thống con.*

Subtype: Xem *Subclass.*

Superclass (*lớp cha*): Một lớp tổng quát hoá trong cấu trúc tổng quát hoá.

Supertype: Xem *Superclass.*

Swimlane (*đường phân dòng nghiệp vụ*): Một cột trong biểu đồ hoạt động dùng để biểu thị vùng trách nhiệm của các hoạt động.

Synchronization Bar (*thanh đồng bộ hoá*): Một điểm trong biểu đồ hoạt động hoặc trong biểu đồ trạng thái, tại đó hoặc một chuyển tiếp có thể chia làm nhiều dòng song song, hoặc nhiều dòng song song được đồng bộ hoá để gặp nhau và đưa ra một chuyển tiếp đơn.

Tagged Value (*giá trị thẻ*): Cơ chế gắn cặp (tên, giá trị) vào các phần tử của mô hình; là một trong các cách mở rộng của UML.

Template (*mẫu*): Xem lớp tham số.

Transition (*chuyển tiếp*): Một mối quan hệ giữa các trạng thái, giữa các hoạt động hoặc giữa trạng thái và hoạt động. Khi trạng thái hoặc hoạt động kết thúc, *chuyển tiếp* cho biết trạng thái hoặc hoạt động nào được bắt đầu tiếp theo.

Use Case (*tình huống sử dụng*): Một dãy hành động được thực hiện bởi hệ thống để đạt được một mục đích có ý nghĩa đối với người dùng (hoặc hệ thống) ngoài.

Use Case Diagram (*biểu đồ use case*): Biểu đồ cung cấp một mô hình mức cao các chưc năng của hệ thống, gồm các use case, các actor và các kết hợp giữa chúng.

Utility Class (*lớp tiện ích*): Một tập các thuộc tính và thao tác toàn cục. Khuôn dạng của *lớp tiện ích* là <<*utility*>>

Workflow (*dòng công việc*): Trong UML, *dòng công việc* là thuật ngữ chung để mô hình các tác vụ dưới dạng dãy các hoạt động. Trong UP, *dòng công việc* là một nhóm các hoạt động nhằm thực hiện một tác vụ phát triển hệ thống nào đó.

Visibility (*tính khả kiến*): Đặc tả khả năng truy xuất của một phần tử đến các phần tử kết hợp với nó; thường được áp dụng cho thuộc tính và thao tác, cũng được phép dùng cho đầu mút của kết hợp. UML cung cấp ba lựa chọn cho *tính khả kiến* là public (+), private (-) và protected (#).

CHỈ MỤC

LỜI NÓI ĐẦU

Khi thiết kế các loại sản phẩm chúng ta đều dùng hình ảnh hoặc biểu đồ để trợ giúp. Cũng vậy, các nhà phân tích và thiết kế dùng biểu đồ để mô hình hệ thống phần mềm. Cách làm như vậy cho phép trừu tượng hoá các thuộc tính của bản thiết kế, chỉ ra các mối quan hệ giữa các thành phần của bản thiết kế. Nhờ đó chúng ta có khả năng nhìn một sự vật phức tạp trở nên đơn giản hơn, và không bỏ sót các đặc điểm hệ thống của đối tượng.

Khi đưa ra một mô hình, nhà phân tích và thiết kế đang dùng một ngôn ngữ riêng để họ, cùng với những người khác, hiểu thật chính xác tình trạng hiện tại cũng như yêu cầu trong tương lai của hệ thống. Điều chúng ta cần ở đây là một tiếp cận với các chuẩn mô hình được thừa nhận; từ đó phát triển phương pháp và kỹ thuật cũng như các công cụ hỗ trợ thiết kế.

Ngày nay, tiếp cận định hướng đối tượng dần trở nên phổ biến. Trong các dự án phát triển hệ thống hướng đối tượng, UML là ngôn ngữ mô hình được ưu tiên cho việc phân tích và thiết kế. Lý do mạnh mẽ nhất là bởi vì nó đang tiến triển như là một chuẩn, và trở thành một chuẩn quốc tế được *tổ chức tiêu chuẩn quốc tế ISO* (International Standard Organization) chấp nhận.

Mục tiêu của cuốn sách này nhằm cung cấp:

- Các khái niệm cơ sở của UML thông qua các biểu đồ.
- Các cơ chế mở rộng.
- Các kỹ thuật lập mô hình.
- Các qui trình phát triển phần mềm và qui trình lập mô hình nghiệp vụ.
- Nâng cao kỹ năng và tích lũy kinh nghiệm qua các bài tập được sắp xếp cẩn thận có chủ đích.

Ngoài ra, do cung cấp được các yêu cầu mà, cuốn sách cũng gián tiếp giúp lựa chọn công cụ cho phù hợp.

Cuốn sách bao gồm 16 chương. Hai chương đầu giới thiệu các case study được dùng làm ví dụ để chuyển tải các khái niệm cơ sở của UML – các *case study* là các ví dụ thực tế, chúng ta mượn chúng để trình bày các

khái niệm lý thuyết và trực tiếp triển khai thực hành với chúng như là một cách học hiệu quả. Mười một chương kế tiếp, từ chương 2 đến chương 12, trình bày các biểu đồ UML, giải thích các ký hiệu và trình bày kỹ thuật xây dựng chúng. Với mỗi biểu đồ, ngoài ký hiệu và kỹ thuật xây dựng, chúng tôi còn chỉ rõ mối quan hệ với các biểu đồ khác và cách áp dụng trong mô hình hoá nghiệp vụ, trong qui trình phát triển phần mềm. Các ví dụ và bài tập được phát triển từ dễ đến khó dựa trên hai case study đã cho. Cuối mỗi chương đều có các câu hỏi ôn tập, các bài tập có lời giải và các bài tập bổ sung để bạn đọc tự làm. Chương 14 tóm tắt các thuộc tính chung của tất cả các biểu đồ. Hai chương cuối cùng, chương 15 và 16, cung cấp các thông tin liên quan đến việc lựa chọn công cụ và sử dụng các mẫu thiết kế.

Trong cuốn sách có rất nhiều thuật ngữ chuyên môn bằng tiếng Anh, chúng tôi vẫn để nguyên văn các thuật ngữ này, có mở ngoặc thuật ngữ tiếng Việt tương đương. Để tiện tra cứu, chúng tôi cũng cung cấp một từ điển thuật ngữ và một bản chỉ mục nằm ở cuối cuốn sách. Ngoài ra trong các ví dụ có nhiều ký hiệu biểu đồ dạng văn bản là các từ tiếng Anh; bạn đọc có thể lĩnh hội nội dung của chúng khi đọc các đoạn văn liên quan, trong đó chúng đều được giải thích tỉ mỉ.

Cuốn sách được viết ra không khỏi có những thiếu sót. Chúng tôi rất mong được bạn đọc đóng góp để lần tái bản sau được hoàn thiện hơn. Mọi ý kiến liên quan đến cuốn sách, xin vui lòng gởi về *nhà sách Minh Khai*:

- 249 Nguyễn Thị Minh Khai, Quận I, TP Hồ Chí Minh.
- E-mail: mk.pub@cinet.vnnews.com
- Website: www.minhkhai.com.vn

Xin chân thành cám ơn!

Tp HCM, tháng 08 năm 2003

MK.PUB

MỤC LỤC