

Phân tích và thiết kế HTTT theo UML

Chương 1: TỔNG QUAN VỀ PHÂN TÍCH THIẾT KẾ HỆ THỐNG

1- Dẫn nhập:

- ◆ 1.1- Tính trực quan:
- ◆ 1.2- Mô hình trừu tượng:
- ◆ 1.3- Mô hình hóa trực quan:

2- Mô tả chu trình phát triển phần mềm:

- ◆ 2.1- Software Development – một bài toán phức tạp:
- ◆ 2.2- Chu Trình Phát Triển Phần Mềm (Software Development Life Cycle):
- ◆ 2.3- Các giai đoạn của Chu Trình Phát Triển Phần Mềm:

3- Phương pháp hướng chức năng và phương pháp hướng đối tượng:

- ◆ 3.1- Phương pháp hướng chức năng:
- ◆ 3.2- Phương pháp hướng đối tượng:

4- Ưu điểm của mô hình hướng đối tượng:

- ◆ 4.1- Tính tái sử dụng (Reusable)
- ◆ 4.2- Các giai đoạn của chu trình phát triển phần mềm với mô hình hướng đối tượng:

Phần câu hỏi

Chương 2: NGÔN NGỮ MÔ HÌNH HOÁ THỐNG NHẤT LÀ GÌ

1- Giới thiệu UML:

- ◆ 1.1- Mô hình hóa hệ thống phần mềm.
- ◆ 1.2- Trước khi UML ra đời.
- ◆ 1.3- Sự ra đời của UML.

- ◆ 1.4- UML (Unified Modeling Language).
- ◆ 1.5- Phương pháp và các ngôn ngữ mô hình hoá.

2- UML trong phân tích thiết kế hệ thống:

3- UML và các giai đoạn phát triển hệ thống:

Phần câu hỏi

Chương 3: KHÁI QUÁT VỀ UML

1- UML và các giai đoạn của chu trình phát triển phần mềm

- ◆ 1.1- Giai đoạn nghiên cứu sơ bộ:
- ◆ 1.2- Giai đoạn phân tích:
- ◆ 1.3- Giai đoạn thiết kế:
- ◆ 1.4- Giai đoạn xây dựng:
- ◆ 1.5- Thử nghiệm:

2- Các thành phần của ngôn ngữ UML

3- Hướng nhìn (View)

- ◆ 3.1- Hướng nhìn Use case (Use case View):
- ◆ 3.2- Hướng nhìn logic (Logical View):
- ◆ 3.3- Hướng nhìn thành phần (Component View):
- ◆ 3.4- Hướng nhìn song song (Concurrency View):
- ◆ 3.5- Hướng nhìn triển khai (Deployment View):

4- Biểu đồ (diagram)

- ◆ 4.1- Biểu đồ Use case (Use Case Diagram):
- ◆ 4.2- Biểu đồ lớp (Class Diagram):
- ◆ 4.3- Biểu đồ đối tượng (Object Diagram):
- ◆ 4.4- Biểu đồ trạng thái (State Diagram):

- ◆ 4.5- Biểu đồ trình tự (Sequence Diagram):
- ◆ 4.6- Biểu đồ cộng tác (Collaboration Diagram):
- ◆ 4.7- Biểu đồ hoạt động (Activity Diagram):
- ◆ 4.8- Biểu đồ thành phần (Component Diagram):
- ◆ 4.9- Biểu đồ triển khai (Deployment Diagram):

5- Phần tử mô hình (model element)

6- Cơ chế chung (General Mechanism)

- ◆ 6.1- Trang trí (Adornment)
- ◆ 6.2- Ghi chú (Note)
- ◆ 6.3- Đặc tả (Specification)

7- Mở rộng UML

- ◆ 7.1- Khuôn mẫu (Stereotype)
- ◆ 7.2- Giá trị đính kèm (Tagged Value)
- ◆ 7.3- Hạn chế (Constraint)

8- Mô hình hóa với UML

9- Công cụ (Tool)

10- Tóm tắt về UML

Phần Câu hỏi

Chương 4: Mô hình hóa USE CASE

1- Giới thiệu Use Case

2- Một số ví dụ Use Case

3- Sự cần thiết phải có Use Case

4- Mô hình hóa Use Case

5- Biểu đồ Use Case

- ◆ 5.1- Hệ thống

- ◆ 5.2- Tác nhân
- ◆ 5.3- Tìm tác nhân
- ◆ 5.4- Biểu diễn tác nhân trong ngôn ngữ UML
- ◆ 5.5- Use Case
- ◆ 5.6- Tìm Use Case
- ◆ 5.7- Ví dụ tìm Use Case:

6- Các biến thể (Variations) trong một Use Case

7- Quan hệ giữa các Use Case

- ◆ 7.1- Quan hệ mở rộng
- ◆ 7.2- Quan hệ sử dụng
- ◆ 7.3- Quan hệ chung nhóm

8- Miêu tả Use Case

9- Thử Use Case

10- Thực hiện các Use Case

11- Tóm tắt về Use Case

Phần câu hỏi

Chương 6: MÔ HÌNH ĐỘNG

1- Sự cần thiết có mô hình động (Dynamic model)

2- Các thành phần của mô hình động

3- Ưu điểm của mô hình động

4- Sự kiện và thông điệp (Event & Message)

- ◆ 4.1- Sự kiện (Event)
- ◆ 4.2- Thông điệp (Message)

5- Biểu đồ tuần tự (Sequence diagram)

6- Biểu đồ cộng tác (Collaboration Diagram)

7- Biểu đồ trạng thái (State Diagram)

- ◆ 7.1- Trạng thái và sự biến đổi trạng thái (State transition)
- ◆ 7.2- Biểu đồ trạng thái
- ◆ 7.3- Nhận biết trạng thái và sự kiện
- ◆ 7.4- Một số lời mách bảo cho việc tạo dựng biểu đồ trạng thái

8- Biểu đồ hoạt động (Activity Diagram)

9- Vòng đời đối tượng (Object lifecycle)

- ◆ 9.1- Vòng đời sinh ra và chết đi
- ◆ 9.2- Vòng đời lặp

10- Xem xét lại mô hình động

- ◆ 10.1- Thẩm vấn biểu đồ trạng thái
- ◆ 10.2- Phối hợp sự kiện
- ◆ 10.3- Bao giờ thì sử dụng biểu đồ nào
- ◆ 10.4- Lớp con và biểu đồ trạng thái

11- Phối hợp mô hình đối tượng và mô hình động

12- Tóm tắt về mô hình động

Phần câu hỏi

1- DẪN NHẬP

◆ 1.1- Tính trực quan:

Chúng ta có thể thấy rằng: "Một số tập hợp dữ liệu phức tạp nhất định khi được trình bày bằng đồ thị sẽ truyền tải đến người đọc nhiều thông tin hơn so với các dữ liệu thô". Với phần mềm cũng vậy, khi ngành Công nghiệp của chúng ta ngày càng phát triển, các hệ thống sẽ trở nên phức tạp hơn. Khả năng nắm bắt và kiểm soát sự phức tạp đó của chúng ta đi kèm với khả năng trình bày hệ thống một cách toàn diện - một sự trình bày vượt ra ngoài giới hạn của những dòng lệnh thô. Sự thành công trên thị trường của những ngôn ngữ như Visual Basic và phần giao diện trực quan của C++, Java đã cho thấy sự trình bày trực quan mang tính cốt yếu đối với quá trình phát triển các hệ thống phức tạp.

◆ 1.2- Mô hình trừu tượng:

Trước đây, có một thời gian dài, ngành công nghiệp chúng ta đã phải nói tới một "Cuộc khủng hoảng phần mềm". Các cuộc tranh luận đều dựa trên thực tế là chẳng những nhiều đồ án phần mềm không thể sản sinh ra những hệ thống thoả mãn đòi hỏi và nhu cầu của khách hàng, mà còn vượt quá ngân sách và thời hạn. Các công nghệ mới như lập trình hướng đối tượng, lập trình trực quan cũng như các môi trường phát triển tiên tiến có giúp chúng ta nâng cao năng suất lao động, nhưng trong nhiều trường hợp, chúng chỉ hướng tới tầng thấp nhất của việc phát triển phần mềm: phần viết lệnh (coding). Một trong những vấn đề chính của ngành phát triển phần mềm thời nay là có nhiều đồ án bắt tay vào lập trình quá sớm và tập trung quá nhiều vào việc viết code. Lý do một phần là do ban quản trị thiếu hiểu biết về quy trình phát triển phần mềm và họ nảy lo âu khi thấy đội quân lập trình của họ không viết code. Và bản thân các lập trình viên cũng cảm thấy an tâm hơn khi họ ngồi viết code - vốn là tác vụ mà họ quen thuộc! – hơn là khi xây dựng các mô hình trừu tượng cho hệ thống mà họ phải tạo nên.

◆ 1.3- Mô hình hóa trực quan:

Mô hình hoá trực quan là một phương thức tư duy về vấn đề sử dụng các mô hình được tổ chức xoay quanh các khái niệm đời thực. Mô hình giúp chúng ta hiểu vấn đề, giao tiếp với mọi người có liên quan đến dự án (khách hàng, chuyên gia lĩnh vực thuộc đề án, nhà phân tích, nhà thiết

kế, ...). Mô hình rất hữu dụng trong việc mô hình hoá doanh nghiệp, soạn thảo tài liệu, thiết kế chương trình cũng như ngân hàng dữ liệu. Mô hình giúp hiểu các đòi hỏi của hệ thống tốt hơn, tạo các thiết kế rõ ràng hơn và xây dựng nên các hệ thống dễ bảo trì hơn.

Mô hình là kết quả của sự trừu tượng hóa nhằm miêu tả các thành phần cốt yếu của một vấn đề hay một cấu trúc phức tạp qua việc lọc bớt các chi tiết không quan trọng và làm cho vấn đề trở thành dễ hiểu hơn. Trừu tượng hóa là một năng lực căn bản của con người, cho phép chúng ta giải quyết các vấn đề phức tạp. Các kỹ sư, nghệ sĩ và thợ thủ công đã xây dựng mô hình từ hàng ngàn năm nay để thử nghiệm thiết kế trước khi thực hiện. Phát triển phần mềm cũng không là ngoại lệ. Để xây dựng các hệ thống phức tạp, nhà phát triển phải trừu tượng hóa nhiều hướng nhìn khác nhau của hệ thống, sử dụng ký hiệu chính xác để xây dựng mô hình, kiểm tra xem mô hình có thỏa mãn các đòi hỏi của hệ thống, và dần dần bổ sung thêm chi tiết để chuyển các mô hình thành thực hiện.

Chúng ta xây dựng mô hình cho các hệ thống phức tạp bởi chúng ta không thể hiểu thấu đáo những hệ thống như thế trong trạng thái toàn vẹn của chúng. Khả năng thấu hiểu và nắm bắt tính phức tạp của con người là có hạn. Điều này ta có thể thấy rõ trong ví dụ của ngành xây dựng. Nếu bạn muốn tạo một túp lều ở góc vườn, bạn có thể bắt tay vào xây ngay. Nếu bạn xây một ngôi nhà, có lẽ bạn sẽ cần tới bản vẽ, nhưng nếu bạn muốn xây một toà nhà chọc trời thì chắc chắn bạn không thể không cần bản vẽ. Thế giới phần mềm của chúng ta cũng thế. Chỉ tập trung vào các dòng code hay thậm chí cả phân tích Forms trong Visual Basic chẳng cung cấp một cái nhìn toàn cục về việc phát triển đồ án. Xây dựng mô hình cho phép nhà thiết kế tập trung vào bức tranh lớn về sự tương tác giữa các thành phần trong đồ án, tránh bị sa lầy vào những chi tiết riêng biệt của từng thành phần.

Một môi trường kinh doanh mang tính cạnh tranh gay gắt và luôn luôn thay đổi dẫn đến tính phức tạp ngày càng tăng cao, và tính phức tạp này đặt ra những thách thức đặc trưng cho các nhà phát triển hệ thống. Mô hình giúp chúng ta tổ chức, trình bày trực quan, thấu hiểu và tạo nên các hệ thống phức tạp. Chúng giúp chúng ta đáp ứng các thách thức của việc phát triển phần mềm, hôm nay cũng như ngày mai.

2- MÔ TẢ CHU TRÌNH PHÁT TRIỂN PHẦN MỀM:

◆ 2.1- Software Development – một bài toán phức tạp:

Kinh nghiệm của nhiều nhà thiết kế và phát triển cho thấy phát triển phần mềm là một bài toán phức tạp. Xin nêu một số các lý do thường được kể đến:

- Những người phát triển phần mềm rất khó hiểu cho đúng những gì người dùng cần
- Yêu cầu của người dùng thường thay đổi trong thời gian phát triển.
- Yêu cầu thường được miêu tả bằng văn bản, dài dòng, khó hiểu, nhiều khi thậm chí mâu thuẫn.
- Đội quân phát triển phần mềm, vốn là người "ngoài cuộc", rất khó nhận thức thấu đáo các mối quan hệ tiềm ẩn và phức tạp cần được thể hiện chính xác trong các ứng dụng lớn.
- Khả năng nắm bắt các dữ liệu phức tạp của con người (tại cùng một thời điểm) là có hạn.
- Khó định lượng chính xác hiệu suất của thành phẩm và thỏa mãn chính xác sự mong chờ từ phía người dùng.
- Chọn lựa phần cứng và phần mềm thích hợp cho giải pháp là một trong những thách thức lớn đối với Designer.

Phần mềm ngoài ra cần có khả năng **thích ứng** và **mở rộng**. Phần mềm được thiết kế tốt là phần mềm đứng vững trước những biến đổi trong môi trường, dù từ phía cộng đồng người dùng hay từ phía công nghệ. Ví dụ phần mềm đã được phát triển cho một nhà băng cần có khả năng tái sử dụng cho một nhà băng khác với rất ít sửa đổi hoặc hoàn toàn không cần sửa đổi. Phần mềm thỏa mãn các yêu cầu đó được coi là phần mềm có khả năng thích ứng.

Một phần mềm có khả năng mở rộng là phần mềm được thiết kế sao cho dễ phát triển theo yêu cầu của người dùng mà không cần sửa chữa nhiều.

Chính vì vậy, một số các khiếm khuyết thường gặp trong phát triển phần mềm là:

- Hiểu không đúng những gì người dùng cần
- Không thể thích ứng cho phù hợp với những thay đổi về yêu cầu đối với hệ thống

- Các Module không khớp với nhau
- Phần mềm khó bảo trì và nâng cấp, mở rộng
- Phát hiện trễ các lỗi hổng của dự án
- Chất lượng phần mềm kém
- Hiệu năng của phần mềm thấp
- Các thành viên trong nhóm không biết được ai đã thay đổi cái gì, khi nào, ở đâu, tại sao phải thay đổi.

♦ 2.2- Chu Trình Phát Triển Phần Mềm (Software Development Life Cycle):

Vì phát triển phần mềm là một bài toán khó, nên có lẽ trước hết ta cần điểm qua một số các công việc căn bản của quá trình này. Thường người ta hay tập hợp chúng theo tiến trình thời gian một cách tương đối, xoay quanh chu trình của một phần mềm, dẫn tới kết quả khái niệm Chu Trình Phát Triển Phần Mềm (Software Development Life Cycle - SDLC) như sau:

Chu Trình Phát Triển Phần Mềm là một chuỗi các hoạt động của nhà phân tích (Analyst), nhà thiết kế (Designer), người phát triển (Developer) và người dùng (User) để phát triển và thực hiện một hệ thống thông tin. Những hoạt động này được thực hiện trong nhiều giai đoạn khác nhau.

🔗 **Nhà phân tích (Analyst):** là người nghiên cứu yêu cầu của khách hàng/người dùng để định nghĩa một phạm vi bài toán, nhận dạng nhu cầu của một tổ chức, xác định xem nhân lực, phương pháp và công nghệ máy tính có thể làm sao để cải thiện một cách tốt nhất công tác của tổ chức này.

🔗 **Nhà thiết kế (Designer):** thiết kế hệ thống theo hướng cấu trúc của database, screens, forms và reports – quyết định các yêu cầu về phần cứng và phần mềm cho hệ thống cần được phát triển.

🔗 **Chuyên gia lĩnh vực (Domain Experts):** là những người hiểu thực chất vấn đề cùng tất cả những sự phức tạp của hệ thống cần tin học hoá. Họ không nhất thiết phải là nhà lập trình, nhưng họ có thể giúp nhà lập trình hiểu yêu cầu đặt ra đối với hệ thống cần phát triển. Quá trình phát triển phần mềm sẽ có rất nhiều thuận lợi nếu đội ngũ làm phần mềm có được sự trợ giúp của họ.

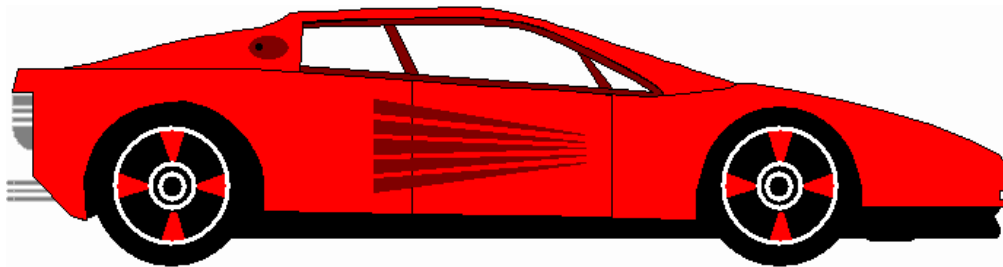
🔗 **Lập trình viên (Programmer):** là những người dựa trên các phân tích và thiết kế để viết chương trình (coding) cho hệ thống bằng ngôn ngữ lập trình đã được thống nhất.

🔗 **Người dùng (User):** là đối tượng phục vụ của hệ thống cần được phát triển.

Để cho rõ hơn, xin lấy ví dụ về một vấn đề đơn giản sau:

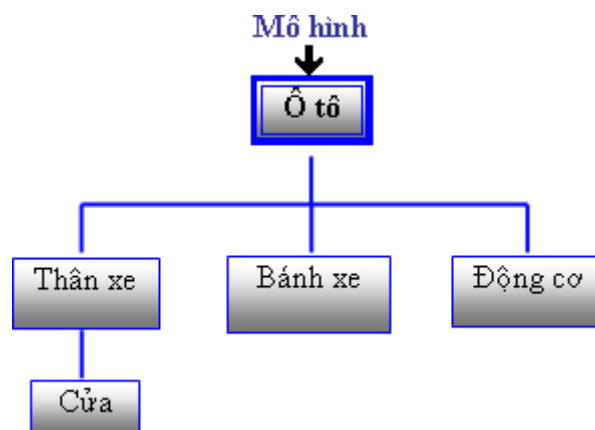
Người bình thường chúng ta khi nhìn một chiếc xe ô tô thường sẽ có một bức tranh từ bên ngoài như sau:

Vấn đề



Hình 1.1: Nhìn vấn đề ô tô của người bình thường

Chuyên gia lĩnh vực sẽ giúp nhà phân tích "trình bày lại" vấn đề như sau:



Hình 1.2: Nhìn vấn đề ô tô của chuyên gia phân tích

Chính vì sự trợ giúp của chuyên gia lĩnh vực có thể đóng vai trò rất quan trọng nên trong những giai đoạn đầu của quá trình phát triển phần mềm, kết quả phân tích nên được thể hiện sao cho dễ hiểu đối với các chuyên gia lĩnh vực. Đây cũng là một trong rất nhiều lý do khiến cho phương pháp hướng đối tượng được nhiều người hưởng ứng.

◆ 2.3- Các giai đoạn của Chu Trình Phát Triển Phần Mềm:

Chu trình của một phần mềm có thể được chia thành các giai đoạn như sau:

- Nghiên cứu sơ bộ (Preliminary Investigation hay còn gọi là Feasibility Study)
- Phân tích yêu cầu (Analysis)
- Thiết kế hệ thống (Design of the System)
- Xây dựng phần mềm (Software Construction)
- Thử nghiệm hệ thống (System Testing)
- Thực hiện, triển khai (System Implementation)
- Bảo trì, nâng cấp (System Maintenance)

a - Nghiên cứu sơ bộ:

Câu hỏi quan trọng nhất khi phát triển một hệ thống hoàn toàn không phải câu hỏi mang tính phương pháp luận. Mà cũng chẳng phải câu hỏi về kỹ thuật. Nó là một câu hỏi dường như có vẻ đơn giản, nhưng thật ra đặc biệt khó trả lời: "Đây có đúng là một hệ thống để thực hiện không?" Đáng buồn là chính câu hỏi này trong thực tế thường chẳng hề được đặt ra và lại càng không được trả lời. Mặc dù việc lăm lăm về phương pháp hay quyết định sai lầm về kỹ thuật cũng có thể dẫn tới thất bại, nhưng thường thì dự án có thể được cứu vãn nếu có đầy đủ tài nguyên cùng sự cố gắng quên mình của các nhân viên tài giỏi. Nhưng sẽ chẳng một ai và một điều gì cứu vãn cho một hệ thống phần mềm hoàn toàn chẳng được cần tới hoặc cố gắng tự động hóa một quy trình lăm lăm.

Trước khi bắt tay vào một dự án, bạn phải có một ý tưởng cho nó. Ý tưởng này đi song song với việc nắm bắt các yêu cầu và xuất hiện trong giai đoạn khởi đầu. Nó hoàn tất một phát biểu: "Hệ thống mà chúng ta mong muốn sẽ làm được những việc như sau....". Trong suốt giai đoạn này, chúng ta tạo nên một bức tranh về ý tưởng đó, rất nhiều giả thuyết sẽ được công nhận hay loại bỏ. Các hoạt động trong thời gian này thường bao gồm thu thập các ý tưởng, nhận biết rủi ro, nhận biết các giao diện bên ngoài, nhận biết các chức năng chính mà hệ thống cần cung cấp, và có thể tạo một vài nguyên mẫu dùng để "minh chứng các khái niệm của hệ thống". Ý tưởng có thể đến từ nhiều nguồn khác nhau: khách hàng, chuyên

gia lĩnh vực, các nhà phát triển khác, chuyên gia về kỹ nghệ, các bản nghiên cứu tính khả thi cũng như việc xem xét các hệ thống khác đang tồn tại. Một khía cạnh cần nhắc tới là code viết trong thời kỳ này thường sẽ bị "bỏ đi", bởi chúng được viết nhằm mục đích thẩm tra hay trợ giúp các giả thuyết khác nhau, chứ chưa phải thứ code được viết theo kết quả phân tích và thiết kế thấu đáo.

Trong giai đoạn nghiên cứu sơ bộ, nhóm phát triển hệ thống cần xem xét các yêu cầu của doanh nghiệp (cần dùng hệ thống), những nguồn tài nguyên có thể sử dụng, công nghệ cũng như cộng đồng người dùng cùng các ý tưởng của họ đối với hệ thống mới. Có thể thực hiện thảo luận, nghiên cứu, xem xét khía cạnh thương mại, phân tích khả năng lời-lỗ, phân tích các trường hợp sử dụng và tạo các nguyên mẫu để xây dựng nên một khái niệm cho hệ thống đích cùng với các mục đích, quyền ưu tiên và phạm vi của nó.

Thường trong giai đoạn này người ta cũng tiến hành tạo một phiên bản thô của lịch trình và kế hoạch sử dụng tài nguyên.

Một giai đoạn nghiên cứu sơ bộ thích đáng sẽ lập nên tập hợp các yêu cầu (dù ở mức độ khái quát cao) đối với một hệ thống khả thi và được mong muốn, kể cả về phương diện kỹ thuật lẫn xã hội. Một giai đoạn nghiên cứu sơ bộ không được thực hiện thoả đáng sẽ dẫn tới các hệ thống không được mong muốn, đắt tiền, bất khả thi và được định nghĩa lằm lặc – những hệ thống thường chẳng được hoàn tất hay sử dụng.

Kết quả của giai đoạn nghiên cứu sơ bộ là Báo Cáo Kết Quả Nghiên Cứu Tính Khả Thi. Khi hệ thống tương lai được chấp nhận dựa trên bản báo cáo này cũng là lúc giai đoạn Phân tích bắt đầu.

b- Phân tích yêu cầu

Sau khi đã xem xét về tính khả thi của hệ thống cũng như tạo lập một bức tranh sơ bộ của dự án, chúng ta bước sang giai đoạn thường được coi là quan trọng nhất trong các công việc lập trình: hiểu hệ thống cần xây dựng. Người thực hiện công việc này là nhà phân tích.

Quá trình phân tích nhìn chung là hệ quả của việc trả lời câu hỏi "Hệ thống cần phải làm gì?". Quá trình phân tích bao gồm việc nghiên cứu chi tiết hệ thống doanh nghiệp hiện thời, tìm cho ra nguyên lý hoạt động của nó và những vị trí có thể được nâng cao, cải thiện. Bên cạnh đó là việc nghiên cứu xem xét các chức năng mà hệ thống

cần cung cấp và các mối quan hệ của chúng, bên trong cũng như với phía ngoài hệ thống. Trong toàn bộ giai đoạn này, nhà phân tích và người dùng cần cộng tác mật thiết với nhau để xác định các yêu cầu đối với hệ thống, tức là các tính năng mới cần phải được đưa vào hệ thống.

Những mục tiêu cụ thể của giai đoạn phân tích là:

- ❑ Xác định hệ thống cần phải làm gì.
- ❑ Nghiên cứu thấu đáo tất cả các chức năng cần cung cấp và những yếu tố liên quan
- ❑ Xây dựng một mô hình nêu bật bản chất vấn đề từ một hướng nhìn có thực (trong đời sống thực).
- ❑ Trao định nghĩa vấn đề cho chuyên gia lĩnh vực để nhận sự đánh giá, góp ý.
- ❑ Kết quả của giai đoạn phân tích là bản Đặc Tả Yêu Cầu (Requirements Specifications).

c - Thiết kế hệ thống

Sau giai đoạn phân tích, khi các yêu cầu cụ thể đối với hệ thống đã được xác định, giai đoạn tiếp theo là thiết kế cho các yêu cầu mới. Công tác thiết kế xoay quanh câu hỏi chính: Hệ thống làm cách nào để thỏa mãn các yêu cầu đã được nêu trong Đặc Tả Yêu Cầu?

Một số các công việc thường được thực hiện trong giai đoạn thiết kế:

- ❑ Nhận biết form nhập liệu tùy theo các thành phần dữ liệu cần nhập.
- ❑ Nhận biết reports và những output mà hệ thống mới phải sản sinh
- ❑ Thiết kế forms (vẽ trên giấy hay máy tính, sử dụng công cụ thiết kế)
- ❑ Nhận biết các thành phần dữ liệu và bảng để tạo database
- ❑ Ước tính các thủ tục giải thích quá trình xử lý từ input đến output.

Kết quả giai đoạn thiết kế là Đặc Tả Thiết Kế (Design Specifications). Bản Đặc Tả Thiết Kế Chi Tiết sẽ được chuyển sang cho các lập trình viên để thực hiện giai đoạn xây dựng phần mềm.

d - Xây dựng phần mềm

Đây là giai đoạn viết lệnh (code) thực sự, tạo hệ thống. Từng người viết code thực hiện những yêu cầu đã được nhà thiết kế định sẵn. Cũng chính người viết code chịu trách nhiệm viết tài liệu liên quan đến chương trình, giải thích thủ tục (procedure) mà anh ta tạo nên được viết như thế nào và lý do cho việc này.

Để đảm bảo chương trình được viết nên phải thoả mãn mọi yêu cầu có ghi trước trong bản Đặc Tả Thiết Kế Chi Tiết, người viết code cũng đồng thời phải tiến hành thử nghiệm phần chương trình của mình. Phần thử nghiệm trong giai đoạn này có thể được chia thành hai bước chính:

Thử nghiệm đơn vị:

Người viết code chạy thử các phần chương trình của mình với dữ liệu giả (test/dummy data). Việc này được thực hiện theo một kế hoạch thử, cũng do chính người viết code soạn ra. Mục đích chính trong giai đoạn thử này là xem chương trình có cho ra những kết quả mong đợi. Giai đoạn thử nghiệm đơn vị nhiều khi được gọi là "Thử hộp trắng" (White Box Testing)

Thử nghiệm đơn vị độc lập:

Công việc này do một thành viên khác trong nhóm đảm trách. Cần chọn người không có liên quan trực tiếp đến việc viết code của đơn vị chương trình cần thử nghiệm để đảm bảo tính "độc lập". Công việc thử đợt này cũng được thực hiện dựa trên kế hoạch thử do người viết code soạn nên.

e- Thử nghiệm hệ thống

Sau khi các thủ tục đã được thử nghiệm riêng, cần phải thử nghiệm toàn bộ hệ thống. Mọi thủ tục được tích hợp và chạy thử, kiểm tra xem mọi chi tiết ghi trong Đặc Tả Yêu Cầu và những mong chờ của người dùng có được thoả mãn. Dữ liệu thử cần được chọn lọc đặc biệt, kết quả cần được phân tích để phát hiện mọi lệch lạc so với mong chờ.

f - Thực hiện, triển khai

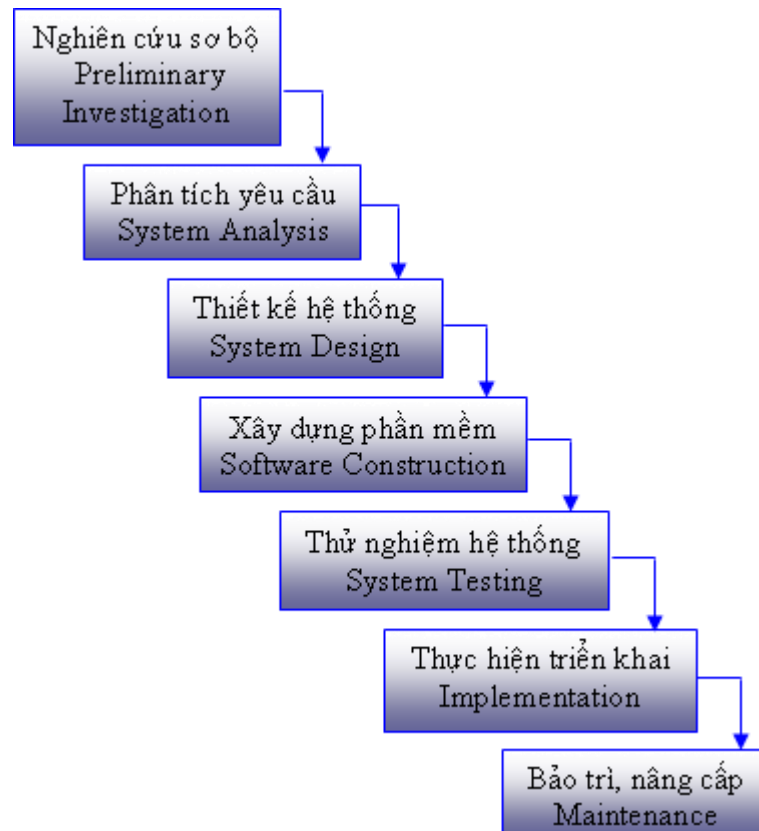
Trong giai đoạn này, hệ thống vừa phát triển sẽ được triển khai sao cho phía người dùng. Trước khi để người dùng thật sự bắt tay vào sử dụng hệ thống, nhóm các nhà phát triển cần tạo các file dữ liệu cần thiết cũng như huấn luyện cho người dùng, để đảm bảo hệ thống được sử dụng hữu hiệu nhất.

g - Bảo trì, nâng cấp

Tùy theo các biến đổi trong môi trường sử dụng, hệ thống có thể trở nên lỗi thời hay cần phải được sửa đổi nâng cấp để sử dụng có hiệu quả. Hoạt

động bảo trì hệ thống có thể rất khác biệt tùy theo mức độ sửa đổi và nâng cấp cần thiết.

Sơ đồ tổng quát các giai đoạn của Chu Trình Phát Triển Phần Mềm:



Hình 1.3: Sơ đồ tổng quát các giai đoạn của Chu Trình Phát Triển Phần Mềm

3- Phương pháp hướng chức năng và phương pháp hướng đối tượng:

◆ 3.1- Phương pháp hướng chức năng:

Đây là lối tiếp cận truyền thống của ngành Công nghệ phần mềm. Theo lối tiếp cận này, chúng ta quan tâm chủ yếu tới những thông tin mà hệ thống sẽ giữ gìn. Chúng ta hỏi người dùng xem họ sẽ cần những thông tin nào, rồi chúng ta thiết kế ngân hàng dữ liệu để chứa những thông tin đó, cung cấp Forms để nhập thông tin và in báo cáo để trình bày các thông tin. Nói một cách khác, chúng ta tập trung vào thông tin và không mấy để ý đến những gì có thể xảy ra với những hệ thống đó và cách hoạt động (ứng xử) của hệ thống là ra sao. Đây là lối tiệm cận xoay quanh dữ liệu và đã được áp dụng để tạo nên hàng ngàn hệ thống trong suốt nhiều năm trời.

Lỗi tiếp cận xoay quanh dữ liệu là phương pháp tốt cho việc thiết kế ngân hàng dữ liệu và nắm bắt thông tin, nhưng nếu áp dụng cho việc thiết kế ứng dụng lại có thể khiến phát sinh nhiều khó khăn. Một trong những thách thức lớn là yêu cầu đối với các hệ thống thường xuyên thay đổi. Một hệ thống xoay quanh dữ liệu có thể dễ dàng xử lý việc thay đổi ngân hàng dữ liệu, nhưng lại khó thực thi những thay đổi trong nguyên tắc nghiệp vụ hay cách hoạt động của hệ thống.

Phương pháp hướng đối tượng đã được phát triển để trả lời cho vấn đề đó. Với lỗi tiếp cận hướng đối tượng, chúng ta tập trung vào cả hai mặt của vấn đề: thông tin và cách hoạt động.

◆ 3.2- Phương pháp hướng đối tượng:

Hướng đối tượng là thuật ngữ thông dụng hiện thời của ngành công nghiệp phần mềm. Các công ty đang nhanh chóng tìm cách áp dụng và tích hợp công nghệ mới này vào các ứng dụng của họ. Thật sự là đa phần các ứng dụng hiện thời đều mang tính hướng đối tượng. Nhưng hướng đối tượng có nghĩa là gì?

Lỗi tiếp cận hướng đối tượng là một lối tư duy về vấn đề theo lối ánh xạ các thành phần trong bài toán vào các đối tượng ngoài đời thực. Với lỗi tiếp cận này, chúng ta chia ứng dụng thành các thành phần nhỏ, gọi là các đối tượng, chúng tương đối độc lập với nhau. Sau đó ta có thể xây dựng ứng dụng bằng cách chắp các đối tượng đó lại với nhau. Hãy nghĩ đến trò chơi xây lâu đài bằng các mẫu gỗ. Bước đầu tiên là tạo hay mua một vài loại mẫu gỗ căn bản, từ đó tạo nên các khối xây dựng căn bản của mình. Một khi đã có các khối xây dựng đó, bạn có thể chắp ráp chúng lại với nhau để tạo lâu đài. Tương tự như vậy một khi đã xây dựng một số đối tượng căn bản trong thế giới máy tính, bạn có thể chắp chúng lại với nhau để tạo ứng dụng của mình.

Xin lấy một ví dụ đơn giản: vấn đề rút tiền mặt tại nhà băng. Các "mẫu gỗ" thành phần ở đây sẽ là ánh xạ của các đối tượng ngoài đời thực như tài khoản, nhân viên, khách hàng, ... Và ứng dụng sẽ được nhận diện cũng như giải đáp xoay quanh các đối tượng đó.

4- ƯU ĐIỂM CỦA MÔ HÌNH HƯỚNG ĐỐI TƯỢNG:

◆ 4.1- Tính tái sử dụng (Reusable)

Phương pháp phân tích và thiết kế hướng đối tượng thực hiện theo các thuật ngữ và khái niệm của phạm vi lĩnh vực ứng dụng (tức là của doanh nghiệp hay đơn vị

mà hệ thống tương lai cần phục vụ), nên nó tạo sự tiếp cận tương ứng giữa hệ thống và vấn đề thực ngoài đời. Trong ví dụ bán xe ô tô, mọi giai đoạn phân tích thiết kế và thực hiện đều xoay quanh các khái niệm như khách hàng, nhân viên bán hàng, xe ô tô, ... Vì quá trình phát triển phần mềm đồng thời là quá trình cộng tác của khách hàng/người dùng, nhà phân tích, nhà thiết kế, nhà phát triển, chuyên gia lĩnh vực, chuyên gia kỹ thuật,... nên lối tiếp cận này khiến cho việc giao tiếp giữa họ với nhau được dễ dàng hơn.

Một trong những ưu điểm quan trọng bậc nhất của phương pháp phân tích và thiết kế hướng đối tượng là tính tái sử dụng: bạn có thể tạo các thành phần (đối tượng) một lần và dùng chúng nhiều lần sau đó. Giống như việc bạn có thể tái sử dụng các khối xây dựng (hay bản sao của nó) trong một tòa lâu đài, một ngôi nhà ở, một con tàu vũ trụ, bạn cũng có thể tái sử dụng các thành phần (đối tượng) căn bản trong các thiết kế hướng đối tượng cũng như code của một hệ thống kế toán, hệ thống kiểm kê, hoặc một hệ thống đặt hàng.

Vì các đối tượng đã được thử nghiệm kỹ càng trong lần dùng trước đó, nên khả năng tái sử dụng đối tượng có tác dụng giảm thiểu lỗi và các khó khăn trong việc bảo trì, giúp tăng tốc độ thiết kế và phát triển phần mềm.

Phương pháp hướng đối tượng giúp chúng ta xử lý các vấn đề phức tạp trong phát triển phần mềm và tạo ra các thể hệ phần mềm có khả năng thích ứng và bền chắc.

4.2- Các giai đoạn của chu trình phát triển phần mềm với mô hình hướng đối tượng:

Phân tích hướng đối tượng (Object Oriented Analysis - OOA):

Là giai đoạn phát triển một mô hình chính xác và súc tích của vấn đề, có thành phần là các đối tượng và khái niệm đời thực, dễ hiểu đối với người sử dụng.

Trong giai đoạn OOA, vấn đề được trình bày bằng các thuật ngữ tương ứng với các đối tượng có thực. Thêm vào đó, hệ thống cần phải được định nghĩa sao cho người không chuyên Tin học có thể dễ dàng hiểu được.

Dựa trên một vấn đề có sẵn, nhà phân tích cần ánh xạ các đối tượng hay thực thể có thực như khách hàng, ô tô, người bán hàng, ... vào thiết kế để tạo ra được bản thiết kế gần cận với tình huống thực. Mô hình thiết kế sẽ chứa các thực thể trong một vấn đề có thực và giữ nguyên các mẫu hình về cấu trúc, quan hệ cũng như hành vi của chúng. Nói một cách khác, sử dụng phương pháp hướng đối tượng chúng ta có thể mô hình hóa các thực thể thuộc một vấn đề có thực mà vẫn giữ được cấu trúc, quan hệ cũng như hành vi của chúng.

Đối với ví dụ một phòng bán ô tô, giai đoạn OOA sẽ nhận biết được các thực thể như:

- Khách hàng
- Người bán hàng
- Phiếu đặt hàng
- Phiếu (hoá đơn) thanh toán
- Xe ô tô

Tương tác và quan hệ giữa các đối tượng trên là:

- Người bán hàng dẫn khách hàng tham quan phòng trưng bày xe.
- Khách hàng chọn một chiếc xe
- Khách hàng viết phiếu đặt xe
- Khách hàng trả tiền xe
- Xe ô tô được giao đến cho khách hàng

Đối với ví dụ nhà băng lẻ, giai đoạn OOA sẽ nhận biết được các thực thể như:

- Loại tài khoản: ATM (rút tiền tự động), Savings (tiết kiệm), Current (bình thường), Fixed (đầu tư),...
- Khách hàng
- Nhân viên
- Phòng máy tính.

Tương tác và quan hệ giữa các đối tượng trên:

- Một khách hàng mới mở một tài khoản tiết kiệm
- Chuyển tiền từ tài khoản tiết kiệm sang tài khoản đầu tư
- Chuyển tiền từ tài khoản tiết kiệm sang tài khoản ATM

Xin chú ý là ở đây, như đã nói, ta chú ý đến cả **hai** khía cạnh: thông tin và cách hoạt động của hệ thống (tức là những gì có thể xảy ra với những thông tin đó).

Lỗi phân tích bằng kiểu ánh xạ "đời thực" vào máy tính như thế thật sự là ưu điểm lớn của phương pháp hướng đối tượng.

➡ **Thiết kế hướng đối tượng (Object Oriented Design - OOD):**

Là giai đoạn tổ chức chương trình thành các tập hợp đối tượng cộng tác, mỗi đối tượng trong đó là thực thể của một lớp. Các lớp là thành viên của một cây cấu trúc với mối quan hệ thừa kế.

Mục đích của giai đoạn OOD là tạo thiết kế dựa trên kết quả của giai đoạn OOA, dựa trên những quy định phi chức năng, những yêu cầu về môi trường, những yêu cầu về khả năng thực thi,... OOD tập trung vào việc cải thiện kết quả của OOA, tối ưu hóa giải pháp đã được cung cấp trong khi vẫn đảm bảo thỏa mãn tất cả các yêu cầu đã được xác lập.


Trong giai đoạn OOD, nhà thiết kế định nghĩa các chức năng, thủ tục (operations), thuộc tính (attributes) cũng như mối quan hệ của một hay nhiều lớp (class) và quyết định chúng cần phải được điều chỉnh sao cho phù hợp với môi trường phát triển. Đây cũng là giai đoạn để thiết kế ngân hàng dữ liệu và áp dụng các kỹ thuật tiêu chuẩn hóa.

Về cuối giai đoạn OOD, nhà thiết kế đưa ra một loạt các biểu đồ (diagram) khác nhau. Các biểu đồ này có thể được chia thành hai nhóm chính là Tĩnh và động. Các biểu đồ tĩnh biểu thị các lớp và đối tượng, trong khi biểu đồ động biểu thị tương tác giữa các lớp và phương thức hoạt động chính xác của chúng. Các lớp đó sau này có thể được nhóm thành các gói (Packages) tức là các đơn vị thành phần nhỏ hơn của ứng dụng.


Lập trình hướng đối tượng (Object Oriented Programming - OOP):

Giai đoạn xây dựng phần mềm có thể được thực hiện sử dụng kỹ thuật lập trình hướng đối tượng. Đó là phương thức thực hiện thiết kế hướng đối tượng qua việc sử dụng một ngôn ngữ lập trình có hỗ trợ các tính năng hướng đối tượng. Một vài ngôn ngữ hướng đối tượng thường được nhắc tới là C++ và Java. Kết quả chung cuộc của giai đoạn này là một loạt các code chạy được, nó chỉ được đưa vào sử dụng sau khi đã trải qua nhiều vòng quay của nhiều bước thử nghiệm khác nhau.

PHẦN CÂU HỎI

 **Hỏi:** Một số tập hợp dữ liệu phức tạp nhất định khi được trình bày bằng đồ thị sẽ truyền tải đến người đọc nhiều thông tin hơn so với các dữ liệu thô?

 **Đáp:** Đúng

 **Hỏi:** Mô hình giúp chúng ta tổ chức, trình bày trực quan, thấu hiểu và tạo nên các hệ thống phức tạp.

 **Đáp:** Đúng

🔗 **Hỏi:** Ưu điểm lớn nhất của mô hình hướng đối tượng là tính tái sử dụng (Reusable)?

👉 **Đáp:** Đúng.

☐ ☐ ☐

1- GIỚI THIỆU UML:

◆ 1.1- Mô hình hóa hệ thống phần mềm:

Như đã trình bày ở phần trước, mục tiêu của giai đoạn phân tích hệ thống là sản xuất ra một mô hình tổng thể của hệ thống cần xây dựng. Mô hình này cần phải được trình bày theo hướng nhìn (View) của khách hàng hay người sử dụng và làm sao để họ hiểu được. Mô hình này cũng có thể được sử dụng để xác định các yêu cầu của người dùng đối với hệ thống và qua đó giúp chúng ta đánh giá tính khả thi của dự án.

Tầm quan trọng của mô hình đã được lĩnh hội một cách thấu đáo trong hầu như tất cả các ngành khoa học kỹ thuật từ nhiều thế kỷ nay. Bất kỳ ở đâu, khi muốn xây dựng một vật thể nào đó, đầu tiên người ta đã tạo nên các bản vẽ để quyết định cả ngoại hình lẫn phương thức hoạt động của nó. Chẳng hạn các bản vẽ kỹ thuật thường gặp là một dạng mô hình quen thuộc. Mô hình nhìn chung là một cách mô tả của một vật thể nào đó. Vật đó có thể tồn tại trong một số giai đoạn nhất định, dù đó là giai đoạn thiết kế hay giai đoạn xây dựng hoặc chỉ là một kế hoạch. Nhà thiết kế cần phải tạo ra các mô hình mô tả tất cả các khía cạnh khác nhau của sản phẩm. Ngoài ra, một mô hình có thể được chia thành nhiều hướng nhìn, mỗi hướng nhìn trong số chúng sẽ mô tả một khía cạnh riêng biệt của sản phẩm hay hệ thống cần được xây dựng. Một mô hình cũng có thể được xây dựng trong nhiều giai đoạn và ở mỗi giai đoạn, mô hình sẽ được bổ sung thêm một số chi tiết nhất định.

Mô hình thường được mô tả trong ngôn ngữ trực quan, điều đó có nghĩa là đa phần các thông tin được thể hiện bằng các ký hiệu đồ họa và các kết nối giữa chúng, chỉ khi cần thiết một số thông tin mới được biểu diễn ở dạng văn bản; Theo đúng như câu ngạn ngữ "Một bức tranh nói nhiều hơn cả ngàn từ". Tạo mô hình cho các hệ thống phần mềm trước khi thực sự xây dựng nên chúng, đã trở thành một chuẩn mực trong việc phát triển phần mềm và được chấp nhận trong cộng đồng làm phần mềm giống như trong bất kỳ một ngành khoa học kỹ thuật nào khác. Việc biểu diễn mô hình phải thỏa mãn các yếu tố sau:

- Chính xác (accurate): Mô tả đúng hệ thống cần xây dựng.
- Đồng nhất (consistent): Các view khác nhau không được mâu thuẫn với nhau.
- Có thể hiểu được (understandable): Cho những người xây dựng lẫn sử dụng
- Dễ thay đổi (changeable)

- Dễ dàng liên lạc với các mô hình khác.

Có thể nói thêm rằng mô hình là một sự đơn giản hoá hiện thực. Mô hình được xây dựng nên để chúng ta dễ dàng hiểu và hiểu tốt hơn hệ thống cần xây dựng. Tạo mô hình sẽ giúp cho chúng ta hiểu thấu đáo một hệ thống phức tạp trong sự toàn thể của nó.

Nói tóm lại, mô hình hóa một hệ thống nhằm mục đích:

- Hình dung một hệ thống theo thực tế hay theo mong muốn của chúng ta.
- Chỉ rõ cấu trúc hoặc ứng xử của hệ thống.
- Tạo một khuôn mẫu hướng dẫn nhà phát triển trong suốt quá trình xây dựng hệ thống.
- Ghi lại các quyết định của nhà phát triển để sử dụng sau này.



1.2- Trước khi UML ra đời:

Đầu những năm 1980, ngành công nghệ phần mềm chỉ có duy nhất một ngôn ngữ hướng đối tượng là Simula. Sang nửa sau của thập kỷ 1980, các ngôn ngữ hướng đối tượng như Smalltalk và C++ xuất hiện. Cùng với chúng, nảy sinh nhu cầu mô hình hoá các hệ thống phần mềm theo hướng đối tượng. Và một vài trong số những ngôn ngữ mô hình hoá xuất hiện những năm đầu thập kỷ 90 được nhiều người dùng là:

- Grady Booch's Booch Modeling Methodology
- James Rumbaugh's Object Modeling Technique – OMT
- Ivar Jacobson's OOSE Methodology
- Hewlett- Packard's Fusion
- Coad and Yordon's OOA and OOD

Mỗi phương pháp luận và ngôn ngữ trên đều có hệ thống ký hiệu riêng, phương pháp xử lý riêng và công cụ hỗ trợ riêng, khiến nảy ra cuộc tranh luận phương pháp nào là tốt nhất. Đây là cuộc tranh luận khó có câu trả lời, bởi tất cả các phương pháp trên đều có những điểm mạnh và điểm yếu riêng. Vì thế, các nhà phát triển phần mềm nhiều kinh nghiệm thường sử dụng phối hợp các điểm mạnh của mỗi phương pháp cho ứng dụng của mình. Trong thực tế, sự khác biệt giữa các phương pháp đó hầu như không đáng kể và theo cùng tiến trình thời gian, tất cả những phương pháp trên đã tiệm cận lại và bổ sung lẫn cho nhau. Chính hiện thực này đã được những người tiên phong trong lĩnh vực mô hình hoá

hướng đối tượng nhận ra và họ quyết định ngồi lại cùng nhau để tích hợp những điểm mạnh của mỗi phương pháp và đưa ra một mô hình thống nhất cho lĩnh vực công nghệ phần mềm.

◆ **1.3- Sự ra đời của UML:**

Trong bối cảnh trên, người ta nhận thấy cần thiết phải cung cấp một phương pháp tiệm cận được chuẩn hoá và thống nhất cho việc mô hình hoá hướng đối tượng. Yêu cầu cụ thể là đưa ra một tập hợp chuẩn hoá các ký hiệu (Notation) và các biểu đồ (Diagram) để nắm bắt các quyết định về mặt thiết kế một cách rõ ràng, rành mạch. Đã có ba công trình tiên phong nhằm tới mục tiêu đó, chúng được thực hiện dưới sự lãnh đạo của James Rumbaugh, Grady Booch và Ivar Jacobson. Chính những cố gắng này dẫn đến kết quả là xây dựng được một Ngôn Ngữ Mô Hình Hoá Thống Nhất (Unified Modeling Language – UML).

UML là một ngôn ngữ mô hình hoá thống nhất có phần chính bao gồm những ký hiệu hình học, được các phương pháp hướng đối tượng sử dụng để thể hiện và miêu tả các thiết kế của một hệ thống. Nó là một ngôn ngữ để đặc tả, trực quan hoá, xây dựng và làm sơ liệu cho nhiều khía cạnh khác nhau của một hệ thống có nồng độ phần mềm cao. UML có thể được sử dụng làm công cụ giao tiếp giữa người dùng, nhà phân tích, nhà thiết kế và nhà phát triển phần mềm.

Trong quá trình phát triển có nhiều công ty đã hỗ trợ và khuyến khích phát triển UML có thể kể tới như: Hewlett Packard, Microsoft, Oracle, IBM, Unisys.

◆ **1.4- UML (Unified Modeling Language):**

Ngôn ngữ mô hình hóa thống nhất (Unified Modeling Language – UML) là một ngôn ngữ để biểu diễn mô hình theo hướng đối tượng được xây dựng bởi ba tác giả trên với chủ đích là:

- Mô hình hoá các hệ thống sử dụng các khái niệm hướng đối tượng.
- Thiết lập một kết nối từ nhận thức của con người đến các sự kiện cần mô hình hoá.
- Giải quyết vấn đề về mức độ thừa kế trong các hệ thống phức tạp, có nhiều ràng buộc khác nhau.
- Tạo một ngôn ngữ mô hình hoá có thể sử dụng được bởi người và máy.

◆ **1.5- Phương pháp và các ngôn ngữ mô hình hoá:**

Phương pháp hay phương thức (method) là một cách trực tiếp cấu trúc hoá sự suy nghĩ và hành động của con người. Phương pháp cho người sử dụng biết phải

làm gì, làm như thế nào, khi nào và tại sao (mục đích của hành động). Phương pháp chứa các mô hình (model), các mô hình được dùng để mô tả những gì sử dụng cho việc truyền đạt kết quả trong quá trình sử dụng phương pháp. Điểm khác nhau chính giữa một phương pháp và một ngôn ngữ mô hình hoá (modeling language) là ngôn ngữ mô hình hoá không có một tiến trình (process) hay các câu lệnh (instruction) mô tả những công việc người sử dụng cần làm.

Một mô hình được biểu diễn theo một ngôn ngữ mô hình hoá. Ngôn ngữ mô hình hoá bao gồm các ký hiệu – những biểu tượng được dùng trong mô hình – và một tập các quy tắc chỉ cách sử dụng chúng. Các quy tắc này bao gồm:

- Syntactic (Cú pháp): cho biết hình dạng các biểu tượng và cách kết hợp chúng trong ngôn ngữ.
- Semantic (Ngữ nghĩa): cho biết ý nghĩa của mỗi biểu tượng, chúng được hiểu thế nào khi nằm trong hoặc không nằm trong ngữ cảnh của các biểu tượng khác.
- Pragmatic: định nghĩa ý nghĩa của biểu tượng để sao cho mục đích của mô hình được thể hiện và mọi người có thể hiểu được.

2- UML TRONG PHÂN TÍCH THIẾT KẾ HỆ THỐNG:

UML có thể được sử dụng trong nhiều giai đoạn, từ phát triển, thiết kế cho tới thực hiện và bảo trì. Vì mục đích chính của ngôn ngữ này là dùng các biểu đồ hướng đối tượng để mô tả hệ thống nên miền ứng dụng của UML bao gồm nhiều loại hệ thống khác nhau như:

- **Hệ thống thông tin** (Information System): Giữ, lấy, biến đổi biểu diễn thông tin cho người sử dụng. Xử lý những khoảng dữ liệu lớn có các quan hệ phức tạp, mà chúng được lưu trữ trong các cơ sở dữ liệu quan hệ hay hướng đối tượng.
- **Hệ thống kỹ thuật** (Technical System): Xử lý và điều khiển các thiết bị kỹ thuật như viễn thông, hệ thống quân sự, hay các quá trình công nghiệp. Đây là loại thiết bị phải xử lý các giao tiếp đặc biệt, không có phần mềm chuẩn và thường là các hệ thống thời gian thực (real time).
- **Hệ thống nhúng** (Embedded System): Thực hiện trên phần cứng gắn vào các thiết bị như điện thoại di động, điều khiển xe hơi, ... Điều này được thực hiện bằng việc lập trình mức thấp với hỗ trợ thời gian thực. Những hệ thống này thường không có các thiết bị như màn hình đĩa cứng, ...

➡ **Hệ thống phân bố** (Distributed System): Được phân bố trên một số máy cho phép truyền dữ liệu từ nơi này đến nơi khác một cách dễ dàng. Chúng đòi hỏi các cơ chế liên lạc đồng bộ để đảm bảo toàn vẹn dữ liệu và thường được xây dựng trên một số các kỹ thuật đối tượng như CORBA, COM/DCOM, hay Java Beans/RMI.

➡ **Hệ thống Giao dịch** (Business System): Mô tả mục đích, tài nguyên (con người, máy tính, ...), các quy tắc (luật pháp, chiến thuật kinh doanh, cơ chế, ...), và công việc hoạt động kinh doanh.

➡ **Phần mềm hệ thống** (System Software): Định nghĩa cơ sở hạ tầng kỹ thuật cho phần mềm khác sử dụng, chẳng hạn như hệ điều hành, cơ sở dữ liệu, giao diện người sử dụng.

3- UML VÀ CÁC GIAI ĐOẠN PHÁT TRIỂN HỆ THỐNG

➡ **Preliminary Investigation**: use cases thể hiện các yêu cầu của người dùng. Phần miêu tả use case xác định các yêu cầu, phần diagram thể hiện mối quan hệ và giao tiếp với hệ thống.

➡ **Analysis**: Mục đích chính của giai đoạn này là trừu tượng hóa và tìm hiểu các cơ cấu có trong phạm vi bài toán. Class diagrams trên bình diện trừu tượng hóa các thực thể ngoài đời thực được sử dụng để làm rõ sự tồn tại cũng như mối quan hệ của chúng. Chỉ những lớp (class) nằm trong phạm vi bài toán mới đáng quan tâm.

➡ **Design**: Kết quả phần analysis được phát triển thành giải pháp kỹ thuật. Các lớp được mô hình hóa chi tiết để cung cấp hạ tầng kỹ thuật như giao diện, nền tảng cho database, ... Kết quả phần Design là các đặc tả chi tiết cho giai đoạn xây dựng phần mềm.

➡ **Development**: Mô hình Design được chuyển thành code. Programmer sử dụng các UML diagrams trong giai đoạn Design để hiểu vấn đề và tạo code.

➡ **Testing**: Sử dụng các UML diagrams trong các giai đoạn trước. Có 4 hình thức kiểm tra hệ thống:

- ➡ *Unit testing* (class diagrams & class specifications): kiểm tra từng đơn thể, được dùng để kiểm tra các lớp hay các nhóm đơn thể.

- ➡ *Integration testing* (integration diagrams & collaboration diagrams): kiểm tra tích hợp là kiểm tra kết hợp các

component với các lớp để xem chúng hoạt động với nhau có đúng không.

- *System testing* (use-case diagrams): kiểm tra xem hệ thống có đáp ứng được chức năng mà người sử dụng yêu cầu hay không.

- *Acceptance testing*: Kiểm tra tính chấp nhận được của hệ thống, thường được thực hiện bởi khách hàng, việc kiểm tra này thực hiện tương tự như kiểm tra hệ thống.

PHẦN CÂU HỎI

🔗 **Hỏi:** UML (Unified Modeling Language) là gì?

▶ **Đáp:** Ngôn ngữ mô hình hóa thống nhất – UML là một ngôn ngữ để biểu diễn mô hình theo hướng đối tượng.

🔗 **Hỏi:** Điểm khác nhau cơ bản giữa phương pháp (method) và một ngôn ngữ mô hình hoá (modeling language) là gì?

▶ **Đáp:** Điểm khác nhau cơ bản giữa một phương pháp và một ngôn ngữ mô hình hoá là ngôn ngữ mô hình hoá không có một tiến trình (process) hay các câu lệnh (instruction) mô tả những công việc người sử dụng cần làm mà nó bao gồm các ký hiệu – những biểu tượng được dùng trong mô hình – và một tập các quy tắc chỉ cách sử dụng chúng.

□ □ □

Chương 3: KHÁI QUÁT VỀ UML

1- UML VÀ CÁC GIAI ĐOẠN CỦA CHU TRÌNH PHÁT TRIỂN PHẦN MỀM

1.1- Giai đoạn nghiên cứu sơ bộ:

UML đưa ra khái niệm **Use Case** để nắm bắt các yêu cầu của khách hàng (người sử dụng). UML sử dụng biểu đồ Use case (Use Case Diagram) để nêu bật mối quan hệ cũng như sự giao tiếp với hệ thống.

Qua phương pháp mô hình hóa Use case, các tác nhân (Actor) bên ngoài quan tâm đến hệ thống sẽ được mô hình hóa song song với chức năng mà họ đòi hỏi từ phía hệ thống (tức là Use case). Các tác nhân và các Use case được mô hình hóa cùng các mối quan hệ và được miêu tả trong biểu đồ Use case của UML. Mỗi một Use case được mô tả trong tài liệu, và nó sẽ đặc tả các yêu cầu của khách hàng: Anh ta hay chị ta chờ đợi điều gì ở phía hệ thống mà không hề để ý đến việc chức năng này sẽ được thực thi ra sao.

1.2- Giai đoạn phân tích:

Giai đoạn phân tích quan tâm đến quá trình trừu tượng hóa đầu tiên (các lớp và các đối tượng) cũng như cơ chế hiện hữu trong phạm vi vấn đề. Sau khi nhà phân tích đã nhận biết được các lớp thành phần của mô hình cũng như mối quan hệ giữa chúng với nhau, các lớp cùng các mối quan hệ đó sẽ được miêu tả bằng công cụ biểu đồ lớp (class diagram) của UML. Sự cộng tác giữa các lớp nhằm thực hiện các Use case cũng sẽ được miêu tả nhờ vào các mô hình động (dynamic models) của UML. Trong giai đoạn phân tích, chỉ duy nhất các lớp có tồn tại trong phạm vi vấn đề (các khái niệm đời thực) là được mô hình hóa. Các lớp kỹ thuật định nghĩa chi tiết cũng như giải pháp trong hệ thống phần mềm, ví dụ như các lớp cho giao diện người dùng, cho ngân hàng dữ liệu, cho sự giao tiếp, trùng hợp, v.v..., chưa phải là mối quan tâm của giai đoạn này.

1.3- Giai đoạn thiết kế:

Trong giai đoạn này, kết quả của giai đoạn phân tích sẽ được mở rộng thành một giải pháp kỹ thuật. Các lớp mới sẽ được bổ sung để tạo thành một hạ tầng cơ sở kỹ thuật: Giao diện người dùng, các chức năng để lưu trữ các đối tượng trong ngân hàng dữ liệu, giao tiếp với các hệ thống khác, giao diện với các thiết bị ngoại vi và các máy móc khác trong hệ thống,.... Các lớp thuộc phạm vi vấn đề có từ giai đoạn phân tích sẽ được "nhúng" vào hạ tầng cơ sở kỹ thuật này, tạo ra khả năng thay đổi trong cả hai phương diện: Phạm vi vấn đề và hạ tầng cơ sở.

Giai đoạn thiết kế sẽ đưa ra kết quả là bản đặc tả chi tiết cho giai đoạn xây dựng hệ thống.

1.4- Giai đoạn xây dựng:

Trong giai đoạn xây dựng (giai đoạn lập trình), các lớp của giai đoạn thiết kế sẽ được biến thành những dòng code cụ thể trong một ngôn ngữ lập trình hướng đối tượng cụ thể (không nên dùng một ngôn ngữ lập trình hướng chức năng!). Phụ thuộc vào khả năng của ngôn ngữ được sử dụng, đây có thể là một công việc khó khăn hay dễ dàng. Khi tạo ra các mô hình phân tích và thiết kế trong UML, tốt nhất nên cố gắng né tránh việc ngay lập tức biến đổi các mô hình này thành các dòng code. Trong những giai đoạn trước, mô hình được sử dụng để dễ hiểu, dễ giao tiếp và tạo nên cấu trúc của hệ thống; vì vậy, vội vàng đưa ra những kết luận về việc viết code có thể sẽ thành một trở ngại cho việc tạo ra các mô hình chính xác và đơn giản. Giai đoạn xây dựng là một giai đoạn riêng biệt, nơi các mô hình được chuyển thành code.


1.5- Thử nghiệm:

Như đã trình bày trong phần Chu Trình Phát Triển Phần Mềm, một hệ thống phần mềm thường được thử nghiệm qua nhiều giai đoạn và với nhiều nhóm thử nghiệm khác nhau. Các nhóm sử dụng nhiều loại biểu đồ UML khác nhau làm nền tảng cho công việc của mình: Thử nghiệm đơn vị sử dụng biểu đồ lớp (class diagram) và đặc tả lớp, thử nghiệm tích hợp thường sử dụng biểu đồ thành phần (component diagram) và biểu đồ cộng tác (collaboration diagram), và giai đoạn thử nghiệm hệ thống sử dụng biểu đồ Use case (use case diagram) để đảm bảo hệ thống có phương thức hoạt động đúng như đã được định nghĩa từ ban đầu trong các biểu đồ này.

2- CÁC THÀNH PHẦN CỦA NGÔN NGỮ UML

Ngôn ngữ UML bao gồm một loạt các phần tử đồ họa (graphic element) có thể được kết hợp với nhau để tạo ra các biểu đồ. Bởi đây là một ngôn ngữ, nên UML cũng có các nguyên tắc để kết hợp các phần tử đó.

Một số những thành phần chủ yếu của ngôn ngữ UML:

 **Hướng nhìn (view):** Hướng nhìn chỉ ra những khía cạnh khác nhau của hệ thống cần phải được mô hình hóa. Một hướng nhìn không phải là một bản vẽ, mà là một sự trừu tượng hóa bao gồm một loạt các biểu đồ khác nhau. Chỉ qua việc định nghĩa của một loạt các hướng nhìn khác nhau, mỗi hướng nhìn chỉ ra một khía cạnh riêng biệt của hệ thống, người ta mới có thể tạo dựng nên một bức tranh hoàn thiện về hệ thống. Cũng chính các hướng nhìn này

nối kết ngôn ngữ mô hình hóa với quy trình được chọn cho giai đoạn phát triển.

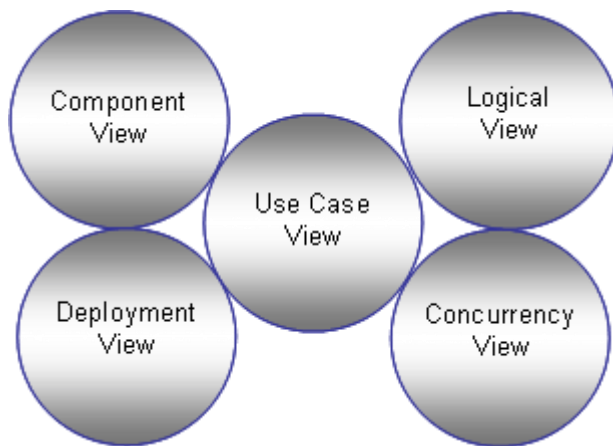
🔗 **Biểu đồ (diagram):** Biểu đồ là các hình vẽ miêu tả nội dung trong một hướng nhìn. UML có tất cả 9 loại biểu đồ khác nhau được sử dụng trong những sự kết hợp khác nhau để cung cấp tất cả các hướng nhìn của một hệ thống.

🔗 **Phần tử mô hình hóa (model element):** Các khái niệm được sử dụng trong các biểu đồ được gọi là các phần tử mô hình, thể hiện các khái niệm hướng đối tượng quen thuộc. Ví dụ như lớp, đối tượng, thông điệp cũng như các quan hệ giữa các khái niệm này, bao gồm cả liên kết, phụ thuộc, khái quát hóa. Một phần tử mô hình thường được sử dụng trong nhiều biểu đồ khác nhau, nhưng nó luôn luôn có chỉ một ý nghĩa và một kí hiệu.

🔗 **Cơ chế chung:** Cơ chế chung cung cấp thêm những lời nhận xét bổ sung, các thông tin cũng như các quy tắc ngữ pháp chung về một phần tử mô hình; chúng còn cung cấp thêm các cơ chế để có thể mở rộng ngôn ngữ UML cho phù hợp với một phương pháp xác định (một quy trình, một tổ chức hoặc một người dùng).

3- HƯỚNG NHÌN (VIEW)

Mô hình hóa một hệ thống phức tạp là một việc làm khó khăn. Lý tưởng nhất là toàn bộ hệ thống được miêu tả chỉ trong một bản vẽ, một bản vẽ định nghĩa một cách rõ ràng và mạch lạc toàn bộ hệ thống, một bản vẽ ngoài ra lại còn dễ giao tiếp và dễ hiểu. Mặc dù vậy, thường thì đây là chuyện bất khả thi. Một bản vẽ không thể nắm bắt tất cả các thông tin cần thiết để miêu tả một hệ thống. Một hệ thống cần phải được miêu tả với một loạt các khía cạnh khác nhau: Về mặt chức năng (cấu trúc tĩnh của nó cũng như các tương tác động), về mặt phi chức năng (yêu cầu về thời gian, về độ đáng tin cậy, về quá trình thực thi, v.v. và v.v.) cũng như về khía cạnh tổ chức (tổ chức làm việc, ánh xạ nó vào các code module,...). Vì vậy một hệ thống thường được miêu tả trong một loạt các hướng nhìn khác nhau, mỗi hướng nhìn sẽ thể hiện một bức ảnh ánh xạ của toàn bộ hệ thống và chỉ ra một khía cạnh riêng của hệ thống.



Hình 3.1- Các View trong UML

Mỗi một hướng nhìn được miêu tả trong một loạt các biểu đồ, chứa đựng các thông tin nêu bật khía cạnh đặc biệt đó của hệ thống. Trong thực tế khi phân tích và thiết kế rất dễ xảy ra sự trùng lặp thông tin, cho nên một biểu đồ trên thật tế có thể là thành phần của nhiều hướng nhìn khác nhau. Khi nhìn hệ thống từ nhiều hướng nhìn khác nhau, tại một thời điểm có thể người ta chỉ tập trung vào một khía cạnh của hệ thống. Một biểu đồ trong một hướng nhìn cụ thể nào đó cần phải đủ độ đơn giản để tạo điều kiện giao tiếp dễ dàng, để dính liền với các biểu đồ khác cũng như các hướng nhìn khác, làm sao cho bức tranh toàn cảnh của hệ thống được miêu tả bằng sự kết hợp tất cả các thông tin từ tất cả các hướng nhìn. Một biểu đồ chứa các kí hiệu hình học mô tả các phần tử mô hình của hệ thống. UML có tất cả các hướng nhìn sau:

- Hướng nhìn Use case (use case view): đây là hướng nhìn chỉ ra khía cạnh chức năng của một hệ thống, nhìn từ hướng tác nhân bên ngoài.
- Hướng nhìn logic (logical view): chỉ ra chức năng sẽ được thiết kế bên trong hệ thống như thế nào, qua các khái niệm về cấu trúc tĩnh cũng như ứng xử động của hệ thống.
- Hướng nhìn thành phần (component view): chỉ ra khía cạnh tổ chức của các thành phần code.
- Hướng nhìn song song (concurrency view): chỉ ra sự tồn tại song song/ trùng hợp trong hệ thống, hướng đến vấn đề giao tiếp và đồng bộ hóa trong hệ thống.
- Hướng nhìn triển khai (deployment view): chỉ ra khía cạnh triển khai hệ thống vào các kiến trúc vật lý (các máy tính hay trang thiết bị được coi là trạm công tác).

Khi bạn chọn công cụ để vẽ biểu đồ, hãy chọn công cụ nào tạo điều kiện dễ dàng chuyển từ hướng nhìn này sang hướng nhìn khác. Ngoài ra, cho mục đích quan sát một chức năng sẽ được thiết kế như thế nào, công cụ này cũng phải tạo điều kiện dễ dàng cho bạn chuyển sang hướng nhìn Use case (để xem chức năng này được miêu tả như thế nào từ phía tác nhân), hoặc chuyển sang hướng nhìn triển khai (để xem chức năng này sẽ được phân bố ra sao trong cấu trúc vật lý - Nói một cách khác là nó có thể nằm trong máy tính nào).

Ngoài các hướng nhìn kể trên, ngành công nghiệp phần mềm còn sử dụng cả các hướng nhìn khác, ví dụ hướng nhìn tĩnh-động, hướng nhìn logic-vật lý, quy trình nghiệp vụ (workflow) và các hướng nhìn khác. UML không yêu cầu chúng ta phải sử dụng các hướng nhìn này, nhưng đây cũng chính là những hướng nhìn mà các nhà thiết kế của UML đã nghĩ tới, nên có khả năng nhiều công cụ sẽ dựa trên các hướng nhìn đó.

3.1- Hướng nhìn Use case (Use case View):

Hướng nhìn Use case miêu tả chức năng của hệ thống sẽ phải cung cấp do được tác nhân từ bên ngoài mong đợi. Tác nhân là thực thể tương tác với hệ thống; đó có thể là một người sử dụng hoặc là một hệ thống khác. Hướng nhìn Use case là hướng nhìn dành cho khách hàng, nhà thiết kế, nhà phát triển và người thử nghiệm; nó được miêu tả qua các biểu đồ Use case (use case diagram) và thành thạo cũng bao gồm cả các biểu đồ hoạt động (activity diagram). Cách sử dụng hệ thống nhìn chung sẽ được miêu tả qua một loạt các Use case trong hướng nhìn Use case, nơi mỗi một Use case là một lời miêu tả mang tính đặc thù cho một tính năng của hệ thống (có nghĩa là một chức năng được mong đợi).

Hướng nhìn Use case mang tính trung tâm, bởi nó đặt ra nội dung thúc đẩy sự phát triển các hướng nhìn khác. Mục tiêu chung của hệ thống là cung cấp các chức năng miêu tả trong hướng nhìn này – cùng với một vài các thuộc tính mang tính phi chức năng khác – vì thế hướng nhìn này có ảnh hưởng đến tất cả các hướng nhìn khác. Hướng nhìn này cũng được sử dụng để thẩm tra (verify) hệ thống qua việc thử nghiệm xem hướng nhìn Use case có đúng với mong đợi của khách hàng (Hỏi: "Đây có phải là thứ bạn muốn") cũng như có đúng với hệ thống vừa được hoàn thành (Hỏi: "Hệ thống có hoạt động như đã đặc tả?").

3.2- Hướng nhìn logic (Logical View):

Hướng nhìn logic miêu tả phương thức mà các chức năng của hệ thống sẽ được cung cấp. Chủ yếu nó được sử dụng cho các nhà thiết kế và nhà phát triển. Ngược lại với hướng nhìn Use case, hướng nhìn logic nhìn vào phía bên trong của hệ thống. Nó miêu tả kể cả cấu trúc tĩnh (lớp, đối tượng, và quan hệ) cũng như sự tương tác động sẽ xảy ra khi các đối tượng gửi thông điệp cho nhau để cung

cấp chức năng đã định sẵn. Hướng nhìn logic định nghĩa các thuộc tính như trường tồn (persistency) hoặc song song (concurrency), cũng như các giao diện cũng như cấu trúc nội tại của các lớp.

Cấu trúc tĩnh được miêu tả bằng các biểu đồ lớp (class diagram) và biểu đồ đối tượng (object diagram). Quá trình mô hình hóa động được miêu tả trong các biểu đồ trạng thái (state diagram), biểu đồ trình tự (sequence diagram), biểu đồ tương tác (collaboration diagram) và biểu đồ hoạt động (activity diagram).

◆ **3.3- Hướng nhìn thành phần (Component View):**

Là một lời miêu tả của việc thực thi các modul cũng như sự phụ thuộc giữa chúng với nhau. Nó thường được sử dụng cho nhà phát triển và thường bao gồm nhiều biểu đồ thành phần. Thành phần ở đây là các modul lệnh thuộc nhiều loại khác nhau, sẽ được chỉ ra trong biểu đồ cùng với cấu trúc cũng như sự phụ thuộc của chúng. Các thông tin bổ sung về các thành phần, ví dụ như vị trí của tài nguyên (trách nhiệm đối với một thành phần), hoặc các thông tin quản trị khác, ví dụ như một bản báo cáo về tiến trình của công việc cũng có thể được bổ sung vào đây.

◆ **3.4- Hướng nhìn song song (Concurrency View):**

Hướng nhìn song song nhằm tới sự chia hệ thống thành các quy trình (process) và các bộ xử lý (processor). Khía cạnh này, vốn là một thuộc tính phi chức năng của hệ thống, cho phép chúng ta sử dụng một cách hữu hiệu các nguồn tài nguyên, thực thi song song, cũng như xử lý các sự kiện không đồng bộ từ môi trường. Bên cạnh việc chia hệ thống thành các tiểu trình có thể được thực thi song song, hướng nhìn này cũng phải quan tâm đến vấn đề giao tiếp và đồng bộ hóa các tiểu trình đó.

Hướng nhìn song song giành cho nhà phát triển và người tích hợp hệ thống, nó bao gồm các biểu đồ động (trạng thái, trình tự, tương tác và hoạt động) cùng các biểu đồ thực thi (biểu đồ thành phần và biểu đồ triển khai).

◆ **3.5- Hướng nhìn triển khai (Deployment View):**

Cuối cùng, hướng nhìn triển khai chỉ cho chúng ta sơ đồ triển khai về mặt vật lý của hệ thống, ví dụ như các máy tính cũng như các máy móc và sự liên kết giữa chúng với nhau. Hướng nhìn triển khai giành cho các nhà phát triển, người tích hợp cũng như người thử nghiệm hệ thống và được thể hiện bằng các biểu đồ triển khai. Hướng nhìn này cũng bao gồm sự ánh xạ các thành phần của hệ thống vào cấu trúc vật lý; ví dụ như chương trình nào hay đối tượng nào sẽ được thực thi trên máy tính nào.

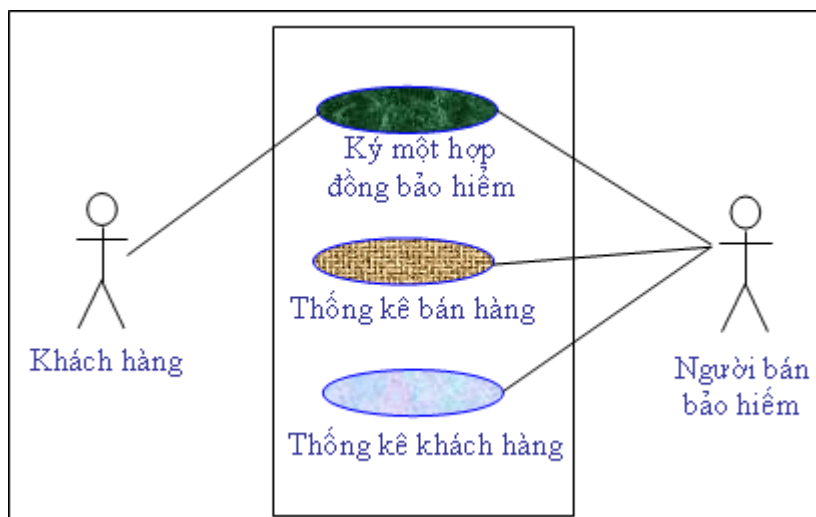
4- BIỂU ĐỒ (DIAGRAM)

Biểu đồ là các hình vẽ bao gồm các ký hiệu phần tử mô hình hóa được sắp xếp để minh họa một thành phần cụ thể hay một khía cạnh cụ thể của hệ thống. Một mô hình hệ thống thường có nhiều loại biểu đồ, mỗi loại có nhiều biểu đồ khác nhau. Một biểu đồ là một thành phần của một hướng nhìn cụ thể; và khi được vẽ ra, nó thường thường cũng được xếp vào một hướng nhìn. Mặt khác, một số loại biểu đồ có thể là thành phần của nhiều hướng nhìn khác nhau, tùy thuộc vào nội dung của biểu đồ.

Phần sau miêu tả các khái niệm căn bản nằm đằng sau mỗi loại biểu đồ. Tất cả các chi tiết về biểu đồ, ngữ cảnh của chúng, ý nghĩa chính xác của chúng và sự tương tác giữa chúng với nhau được miêu tả chi tiết trong các chương sau (mô hình đối tượng – mô hình động). Các biểu đồ lấy làm ví dụ ở đây được lấy ra từ nhiều loại hệ thống khác nhau để chỉ ra nét phong phú và khả năng áp dụng rộng khắp của ULM.

◆ 4.1- Biểu đồ Use case (Use Case Diagram):

Một biểu đồ Use case chỉ ra một số lượng các tác nhân ngoại cảnh và mối liên kết của chúng đối với Use case mà hệ thống cung cấp (nhìn hình 3.2). Một Use case là một lời miêu tả của một chức năng mà hệ thống cung cấp. Lời miêu tả Use case thường là một văn bản tài liệu, nhưng kèm theo đó cũng có thể là một biểu đồ hoạt động. Các Use case được miêu tả duy nhất theo hướng nhìn từ ngoài vào của các tác nhân (hành vi của hệ thống theo như sự mong đợi của người sử dụng), không miêu tả chức năng được cung cấp sẽ hoạt động nội bộ bên trong hệ thống ra sao. Các Use case định nghĩa các yêu cầu về mặt chức năng đối với hệ thống. Các biểu đồ Use case sẽ được miêu tả chi tiết hơn trong chương 4 (Use case).

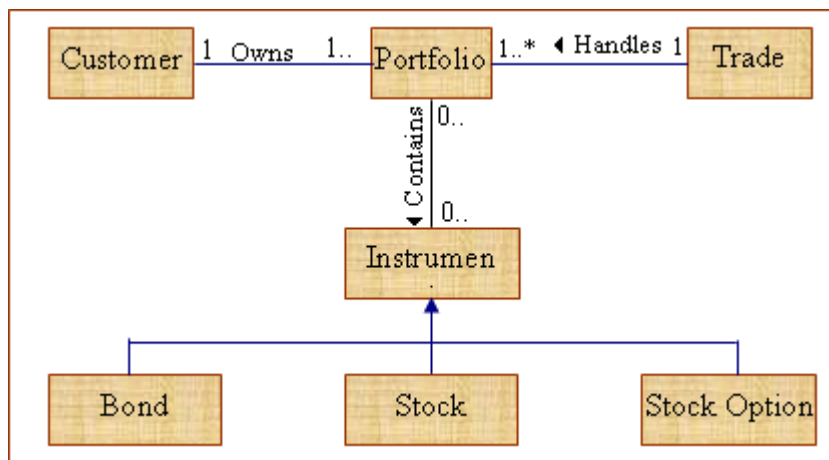


Hình 3.2- Biểu đồ use case của một công ty bảo hiểm

◆ 4.2- Biểu đồ lớp (Class Diagram):

Một biểu đồ lớp chỉ ra cấu trúc tĩnh của các lớp trong hệ thống (nhìn hình 3.3). Các lớp là đại diện cho các “vật” được xử lý trong hệ thống. Các lớp có thể quan hệ với nhau trong nhiều dạng thức: liên kết (associated - được nối kết với nhau), phụ thuộc (dependent - một lớp này phụ thuộc vào lớp khác), chuyên biệt hóa (specialized - một lớp này là một kết quả chuyên biệt hóa của lớp khác), hay đóng gói (packaged - hợp với nhau thành một đơn vị). Tất cả các mối quan hệ đó đều được thể hiện trong biểu đồ lớp, đi kèm với cấu trúc bên trong của các lớp theo khái niệm thuộc tính (attribute) và thủ tục (operation). Biểu đồ được coi là biểu đồ tĩnh theo phương diện cấu trúc được miêu tả ở đây có hiệu lực tại bất kỳ thời điểm nào trong toàn bộ vòng đời hệ thống.

Một hệ thống thường sẽ có một loạt các biểu đồ lớp – chẳng phải bao giờ tất cả các biểu đồ lớp này cũng được nhập vào một biểu đồ lớp tổng thể duy nhất – và một lớp có thể tham gia vào nhiều biểu đồ lớp. Biểu đồ lớp được miêu tả chi tiết trong chương sau.

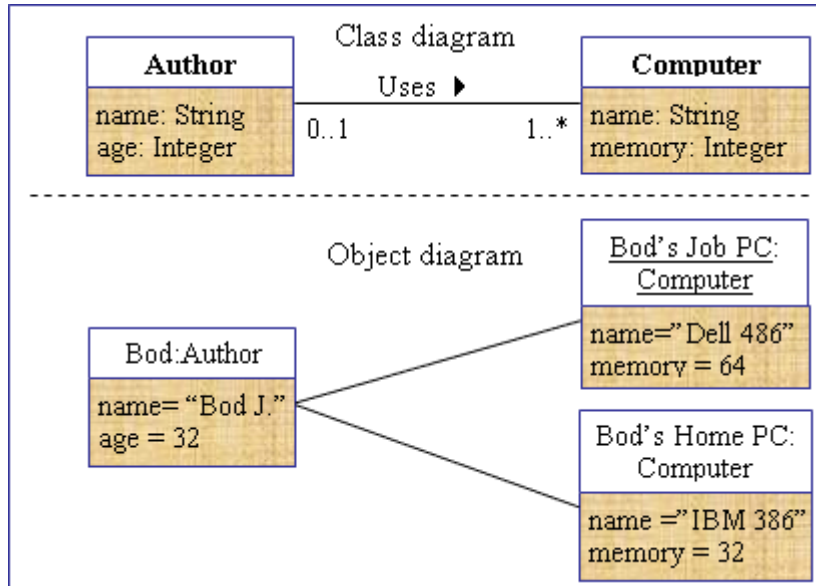


Hình 3.3 - Biểu đồ lớp cho một giao dịch Tài chính

◆ 4.3- Biểu đồ đối tượng (Object Diagram):

Một biểu đồ đối tượng là một phiên bản của biểu đồ lớp và thường cũng sử dụng các ký hiệu như biểu đồ lớp. Sự khác biệt giữa hai loại biểu đồ này nằm ở chỗ biểu đồ đối tượng chỉ ra một loạt các đối tượng thực thể của lớp, thay vì các lớp. Một biểu đồ đối tượng vì vậy là một ví dụ của biểu đồ lớp, chỉ ra một bức tranh thực tế có thể xảy ra khi hệ thống thực thi: bức tranh mà hệ thống có thể có tại một thời điểm nào đó. Biểu đồ đối tượng sử dụng chung các ký hiệu của biểu đồ lớp, chỉ trừ hai ngoại lệ: đối tượng được viết với tên được gạch dưới và tất cả các thực thể trong một mối quan hệ đều được chỉ ra (nhìn hình 3.4).

Biểu đồ đối tượng không quan trọng bằng biểu đồ lớp, chúng có thể được sử dụng để ví dụ hóa một biểu đồ lớp phức tạp, chỉ ra với những thực thể cụ thể và những mối quan hệ như thế thì bức tranh toàn cảnh sẽ ra sao. Một biểu đồ đối tượng thường thường được sử dụng làm một thành phần của một biểu đồ cộng tác (collaboration), chỉ ra lối ứng xử động giữa một loạt các đối tượng.

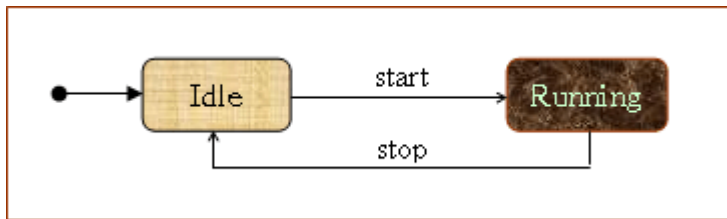


Hình 3.4 - Biểu đồ lớp và biểu đồ đối tượng thể hiện của lớp

◆ 4.4- Biểu đồ trạng thái (State Diagram):

Một biểu đồ trạng thái thường là một sự bổ sung cho lời miêu tả một lớp. Nó chỉ ra tất cả các trạng thái mà đối tượng của lớp này có thể có, và những sự kiện (event) nào sẽ gây ra sự thay đổi trạng thái (hình 3.5). Một sự kiện có thể xảy ra khi một đối tượng tự gửi thông điệp đến cho nó - ví dụ như để thông báo rằng một khoảng thời gian được xác định đã qua đi - hay là một số điều kiện nào đó đã được thỏa mãn. Một sự thay đổi trạng thái được gọi là một sự *chuyển đổi trạng thái* (State Transition). Một chuyển đổi trạng thái cũng có thể có một hành động liên quan, xác định điều gì phải được thực hiện khi sự chuyển đổi trạng thái này diễn ra.

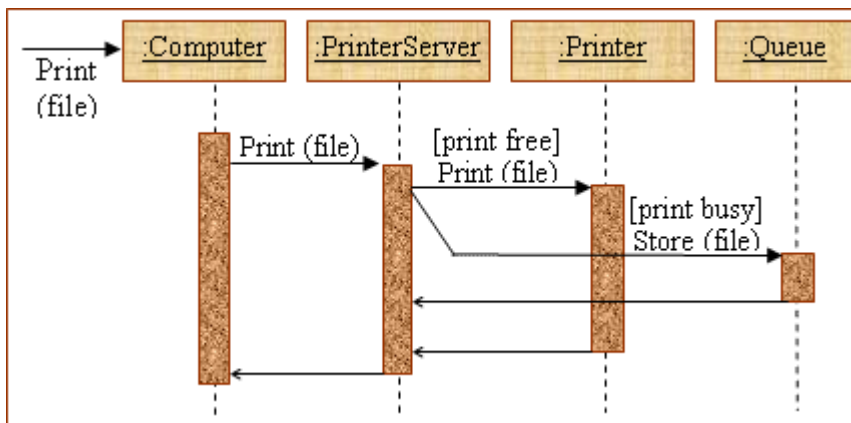
Biểu đồ trạng thái không được vẽ cho tất cả các lớp, mà chỉ riêng cho những lớp có một số lượng các trạng thái được định nghĩa rõ ràng và hành vi của lớp bị ảnh hưởng và thay đổi qua các trạng thái khác nhau. Biểu đồ trạng thái cũng có thể được vẽ cho hệ thống tổng thể. Biểu đồ trạng thái được miêu tả chi tiết hơn trong chương sau (Mô hình động).



Hình 3.5- Một ví dụ về biểu đồ trạng thái

◆ 4.5- Biểu đồ trình tự (Sequence Diagram):

Một biểu đồ trình tự chỉ ra một cộng tác động giữa một loạt các đối tượng (xem hình 3.6). Khía cạnh quan trọng của biểu đồ này là chỉ ra trình tự các thông điệp (message) được gửi giữa các đối tượng. Nó cũng chỉ ra trình tự tương tác giữa các đối tượng, điều sẽ xảy ra tại một thời điểm cụ thể nào đó trong trình tự thực thi của hệ thống. Các biểu đồ trình tự chứa một loạt các đối tượng được biểu diễn bằng các đường thẳng đứng. Trục thời gian có hướng từ trên xuống dưới trong biểu đồ, và biểu đồ chỉ ra sự trao đổi thông điệp giữa các đối tượng khi thời gian trôi qua. Các thông điệp được biểu diễn bằng các đường gạch ngang gắn liền với mũi tên (biểu thị thông điệp) nối liền giữa những đường thẳng đứng thể hiện đối tượng. Trục thời gian cùng những lời nhận xét khác thường sẽ được đưa vào phần lề của biểu đồ.



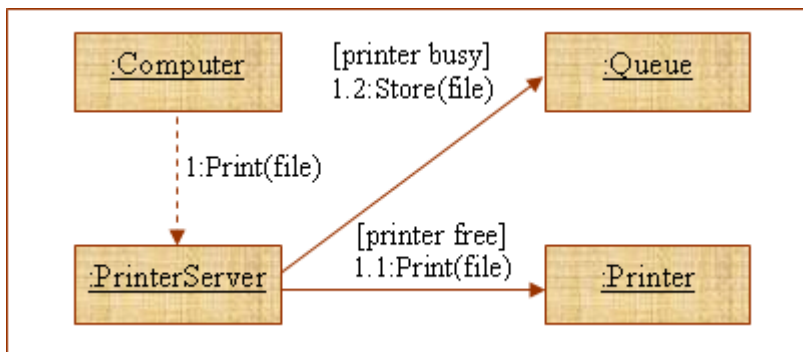
Hình 3.6 - Một biểu đồ trình tự cho Print Server

◆ 4.6- Biểu đồ cộng tác (Collaboration Diagram):

Một biểu đồ cộng tác chỉ ra một sự cộng tác động, cũng giống như một biểu đồ trình tự. Thường người ta sẽ chọn hoặc dùng biểu đồ trình tự hoặc dùng biểu đồ cộng tác. Bên cạnh việc thể hiện sự trao đổi thông điệp (được gọi là *tương tác*), biểu đồ cộng tác chỉ ra các đối tượng và quan hệ của chúng (nhiều khi được gọi là ngữ cảnh). Việc nên sử dụng biểu đồ trình tự hay biểu đồ cộng tác thường sẽ được quyết định theo nguyên tắc chung sau: Nếu thời gian hay trình tự là yếu tố quan trọng nhất cần phải nhấn mạnh thì hãy chọn biểu đồ trình tự; nếu ngữ cảnh

là yếu tố quan trọng hơn, hãy chọn biểu đồ cộng tác. Trình tự tương tác giữa các đối tượng được thể hiện trong cả hai loại biểu đồ này.

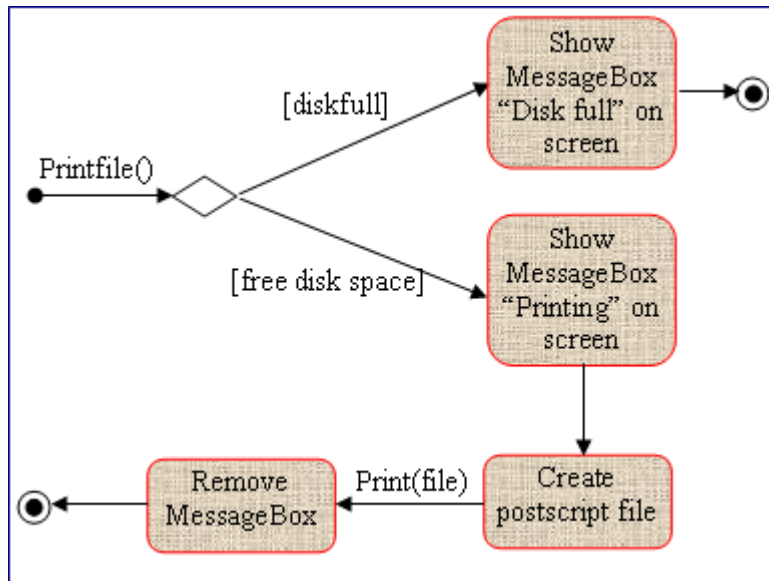
Biểu đồ cộng tác được vẽ theo dạng một biểu đồ đối tượng, nơi một loạt các đối tượng được chỉ ra cùng với mối quan hệ giữa chúng với nhau (sử dụng những ký hiệu như trong biểu đồ lớp/ biểu đồ đối tượng). Các mũi tên được vẽ giữa các đối tượng để chỉ ra dòng chảy thông điệp giữa các đối tượng. Các thông điệp thường được đính kèm theo các nhãn (label), một trong những chức năng của nhãn là chỉ ra thứ tự mà các thông điệp được gửi đi. Nó cũng có thể chỉ ra các điều kiện, chỉ ra những giá trị được trả về, v.v... Khi đã làm quen với cách viết nhãn, một nhà phát triển có thể đọc biểu đồ cộng tác và tuân thủ theo dòng thực thi cũng như sự trao đổi thông điệp. Một biểu đồ cộng tác cũng có thể chứa cả các đối tượng tích cực (active objects), hoạt động song song với các đối tượng tích cực khác (hình 3.7). Biểu đồ cộng tác được miêu tả chi tiết trong chương sau.



Hình 3.7 - Một biểu đồ công tác của một printer server

◆ 4.7- Biểu đồ hoạt động (Activity Diagram):

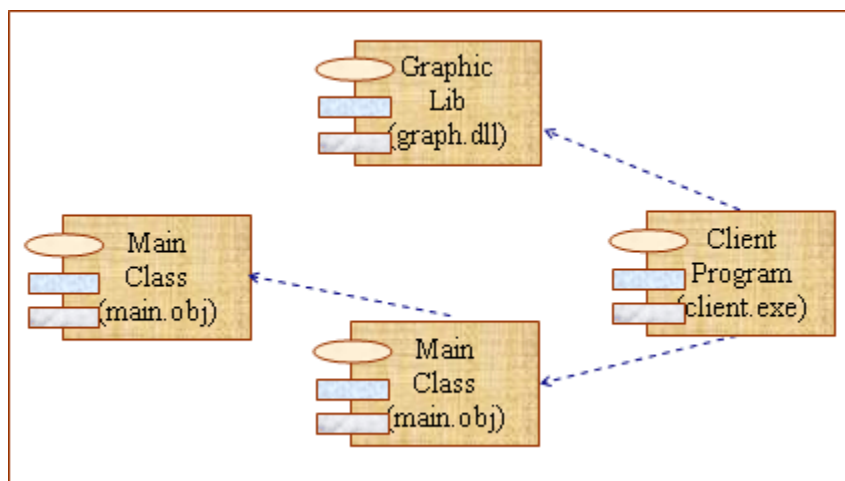
Một biểu đồ hoạt động chỉ ra một trình tự lần lượt của các hoạt động (activity) (hình 3.8). Biểu đồ hoạt động thường được sử dụng để miêu tả các hoạt động được thực hiện trong một thủ tục, mặc dù nó cũng có thể được sử dụng để miêu tả các dòng chảy hoạt động khác, ví dụ như trong một Use case hay trong một trình tự tương tác. Biểu đồ hoạt động bao gồm các trạng thái hành động, chứa đặc tả của một hoạt động cần phải được thực hiện (một hành động - action). Một trạng thái hành động sẽ qua đi khi hành động được thực hiện xong (khác với biểu đồ trạng thái: một trạng thái chỉ chuyển sang trạng thái khác sau khi đã xảy ra một sự kiện rõ ràng !). Dòng điều khiển ở đây chạy giữa các trạng thái hành động liên kết với nhau. Biểu đồ còn có thể chỉ ra các quyết định, các điều kiện, cũng như phần thực thi song song của các trạng thái hành động. Biểu đồ ngoài ra còn có thể chứa các loại đặc tả cho các thông điệp được gửi đi hoặc được nhận về, trong tư cách là thành phần của hành động được thực hiện.



Hình 3.8 - Một biểu đồ hoạt động cho một printer server

◆ 4.8- Biểu đồ thành phần (Component Diagram):

Một biểu đồ thành phần chỉ ra cấu trúc vật lý của các dòng lệnh (code) theo khái niệm thành phần code. Một thành phần code có thể là một tập tin source code, một thành phần nhị phân (binary) hay một thành phần thực thi được (executable). Một thành phần chứa các thông tin về các lớp logic hoặc các lớp mà nó thi hành, như thế có nghĩa là nó tạo ra một ánh xạ từ hướng nhìn logic vào hướng nhìn thành phần. Biểu đồ thành phần cũng chỉ ra những sự phụ thuộc giữa các thành phần với nhau, trợ giúp cho công việc phân tích hiệu ứng mà một thành phần được thay đổi sẽ gây ra đối với các thành phần khác. Thành phần cũng có thể được miêu tả với bất kỳ loại giao diện nào mà chúng bộc lộ, ví dụ như giao diện OLE/COM; và chúng có thể được nhóm gộp lại với nhau thành từng gói (package). Biểu đồ thành phần được sử dụng trong công việc lập trình cụ thể (xem hình 3.9).

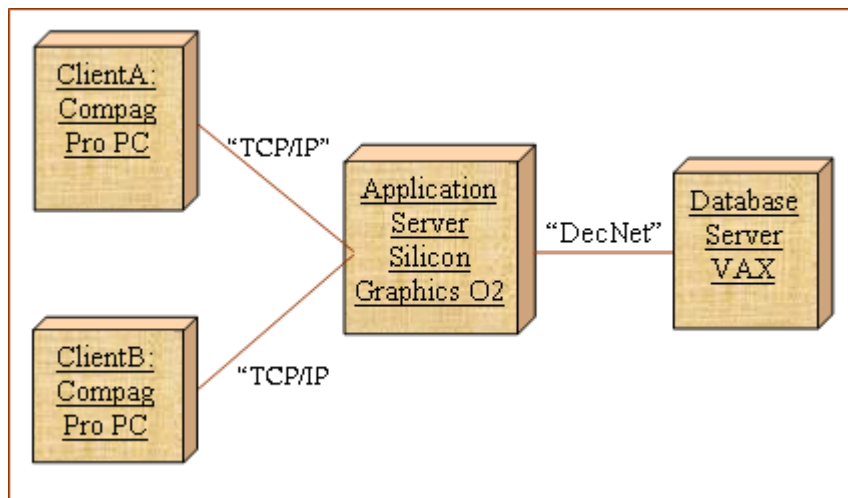


Hình 3.9 - Một biểu đồ thành phần chỉ ra sự phụ thuộc giữa các thành phần mã

4.9- Biểu đồ triển khai (Deployment Diagram):

Biểu đồ triển khai chỉ ra kiến trúc vật lý của phần cứng cũng như phần mềm trong hệ thống. Bạn có thể chỉ ra từng máy tính cụ thể và từng trang thiết bị cụ thể (node) đi kèm sự nối kết giữa chúng với nhau, bạn cũng có thể chỉ ra loại của các mối nối kết đó. Bên trong các nút mạng (node), các thành phần thực thi được cũng như các đối tượng sẽ được xác định vị trí để chỉ ra những phần mềm nào sẽ được thực thi tại những nút mạng nào. Bạn cũng có thể chỉ ra sự phụ thuộc giữa các thành phần.

Biểu đồ triển khai chỉ ra hướng nhìn triển khai, miêu tả kiến trúc vật lý thật sự của hệ thống. Đây là một hướng nhìn rất xa lối miêu tả duy chức năng của hướng nhìn Use case. Mặc dù vậy, trong một mô hình tốt, người ta có thể chỉ tất cả những con đường dẫn từ một nút mạng trong một kiến trúc vật lý cho tới những thành phần của nó, cho tới lớp mà nó thực thi, cho tới những tương tác mà các đối tượng của lớp này tham gia để rồi cuối cùng, tiến tới một Use case. Rất nhiều hướng nhìn khác nhau của hệ thống được sử dụng đồng thời để tạo ra một lời miêu tả thấu đáo đối với hệ thống trong sự tổng thể của nó.

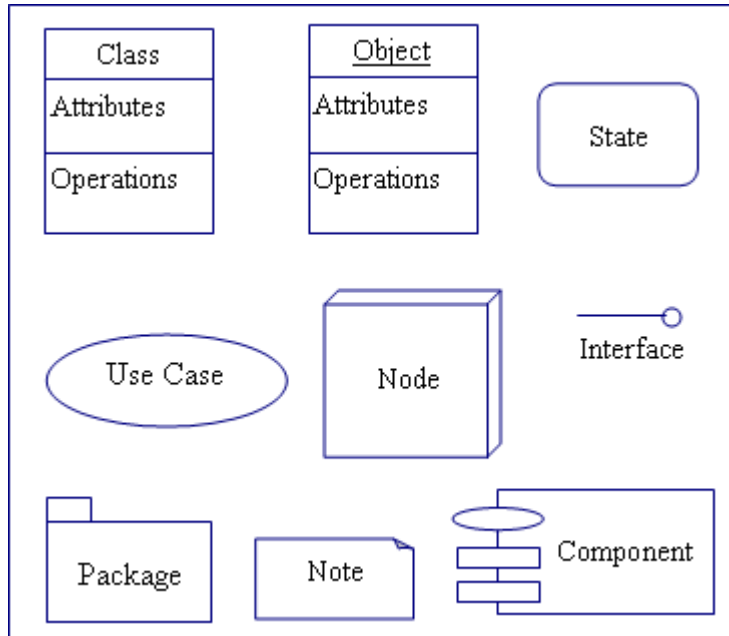


Hình 3.10 - Một biểu đồ triển khai chỉ ra kiến trúc vật lý của hệ thống

5- PHẦN TỬ MÔ HÌNH (MODEL ELEMENT):

Các khái niệm được sử dụng trong các biểu đồ được gọi là các phần tử mô hình (model element). Một phần tử mô hình được định nghĩa với ngữ nghĩa (semantic), đó là một định nghĩa về bản chất phần tử hay là một xác định ý nghĩa chính xác xem nó sẽ thể hiện điều gì trong những lời khẳng định rõ ràng. Mỗi phần tử mô hình còn có một sự miêu tả trực quan, một ký hiệu hình học được sử dụng để miêu tả phần tử này trong biểu đồ. Một phần tử có thể tồn tại trong nhiều dạng

biểu đồ khác nhau, nhưng cũng có những nguyên tắc xác định loại phần tử nào có thể được chỉ ra trong loại biểu đồ nào. Một vài ví dụ cho phần tử vô hình là lớp, đối tượng, trạng thái, nút mạng, gói, thành phần (hình 3.11).

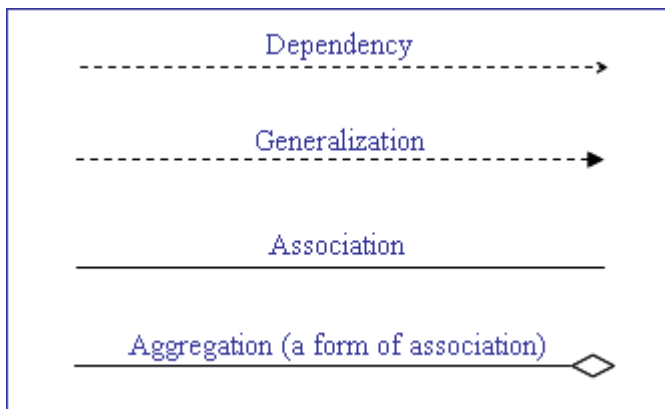


Hình 3.11- Các thành phần mô hình thường dùng

Hình 3.12 chỉ ra một vài ví dụ của mối quan hệ, đây cũng là một dạng phần tử mô hình, chúng được sử dụng để nối các phần tử mô hình khác với nhau. Một vài loại quan hệ đáng chú ý:

- *Nối kết* (Association): nối các phần tử và các thực thể nối (link).
- *Khái quát hóa* (Generalization): còn được gọi là tính thừa kế, có ý nghĩa rằng một phần tử này có thể là một sự chuyên biệt hóa của một phần tử khác.
- *Sự phụ thuộc* (Dependency): chỉ ra rằng một phần tử này phụ thuộc trong một phương thức nào đó vào một phần tử khác.
- *Kết tập* (Aggregation): Một dạng của nối kết, trong đó một phần tử này chứa các phần tử khác.

Ngoài ra còn có các phần tử mô hình khác như thông điệp (Message), hành động (action) và khuôn mẫu (stereotype). Tất cả các phần tử mô hình, ý nghĩa của chúng cũng như những ứng dụng đều được giải thích kỹ lưỡng hơn trong các chương sau.



Hình 3.12 – các ví dụ về vài loại quan hệ

6- CƠ CHẾ CHUNG (GENERAL MECHANISM):

UML thể hiện một số các cơ chế chung trong tất cả các biểu đồ nhằm mục đích cung cấp thêm các thông tin bổ sung, thường đây là những thông tin không thể được thể hiện qua các chức năng và khả năng cơ bản của các phần tử mô hình.

◆ 6.1- Trang trí (Adornment)

Các sự trang trí trực quan có thể được sử dụng kèm thêm vào các phần tử mô hình trong biểu đồ. Động tác trang trí bổ sung thêm ngữ nghĩa cho phần tử. Một ví dụ là kỹ thuật được sử dụng để phân biệt một loại thực thể (lớp) và một thực thể. Khi thể hiện một loại, tên phần tử sẽ được in đậm. Khi cũng chính phần tử đó thể hiện chỉ một thực thể của loại này, tên phần tử sẽ được gạch dưới và có thể được coi là cả tên của thực thể lẫn tên của loại đó. Một hình chữ nhật thể hiện lớp với tên được in đậm sẽ thể hiện một lớp và tên được gạch dưới sẽ thể hiện một đối tượng, đây là một ví dụ tiêu biểu của adornment. Cũng nguyên tắc đó được áp dụng cho các nút mạng, khi ký hiệu nút được in đậm là thể hiện một loại nút, ví dụ như máy in (**Printer**), khi ký hiệu được gạch dưới là thể hiện một thực thể của lớp nút mạng này ví dụ John's HP 5MP-printer. Các kiểu trang trí khác là các lời đặc tả về số lượng trong quan hệ (multiplicity), nơi số lượng là một số hay một khoảng số chỉ ra bao nhiêu thực thể của các loại thực thể được nối với nhau sẽ có thể tham gia trong một quan hệ. Ký hiệu trang trí được viết gần phần tử mô hình được mà nó bổ sung thông tin (hình 3.13).

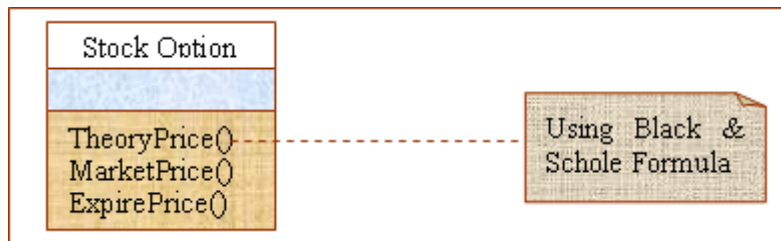


Hình 3.13 - Phân biệt giữa lớp và đối tượng bằng trang trí

◆ 6.2- Ghi chú (Note)

Cho dù một ngôn ngữ mô hình hóa có được mở rộng đến bao nhiêu chăng nữa, nó cũng không thể định nghĩa tất cả mọi việc. Nhằm tạo điều kiện bổ sung thêm cho một mô hình những thông tin không thể được thể hiện bằng phần tử mô hình, UML cung cấp khả năng kèm theo lời ghi chú. Một lời ghi chú có thể được để bất kỳ nơi nào trong bất kỳ biểu đồ nào, và nó có thể chứa bất kỳ loại thông tin nào. Dạng thông tin của bản thân nó là chuỗi ký tự (string), không được UML diễn giải. Lời ghi chú thường đi kèm theo một số các phần tử mô hình trong biểu đồ, được nối bằng một đường chấm chấm, chỉ ra phần tử mô hình nào được chi tiết hóa hoặc được giải thích (hình 3.14).

Một lời ghi chú thường chứa lời nhận xét hoặc các câu hỏi của nhà tạo mô hình, ví dụ lời nhắc nhở cần phải xử lý vấn đề nào đó trong thời gian sau này. Lời ghi chú cũng có thể chứa các thông tin dạng khuôn mẫu (stereotype).

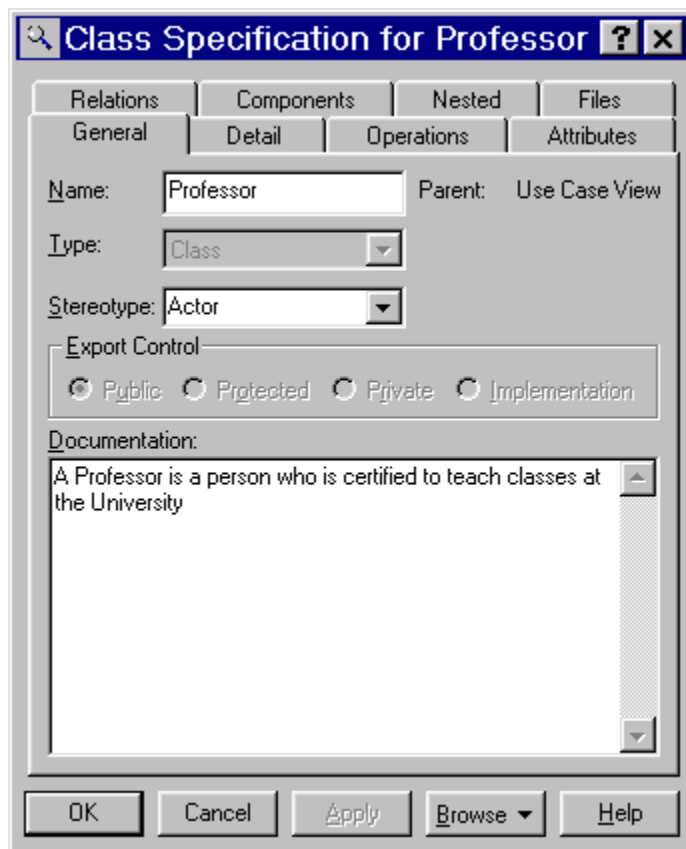


Hình 3.14 - Một ví dụ về ghi chú

◆ 6.3- Đặc tả (Specification)

Các phần tử mô hình có thuộc tính (Property) chứa các giá trị dữ liệu về phần tử này. Một thuộc tính được định nghĩa với một tên và một *giá trị đính kèm* (tagged value), thường chúng ở trong một dạng thông tin được xác định trước, ví dụ như số nguyên hay chuỗi ký tự. Có một loạt thuộc tính đã được định nghĩa trước, ví dụ như tài liệu (document), trách nhiệm (Responsibility), sự trường tồn (Persistence) và tính song song (Concurrency).

Thuộc tính được sử dụng để thêm các đặc tả bổ sung về một phần tử, những thông tin bình thường ra không được thể hiện trong biểu đồ. Ví dụ tiêu biểu là một lớp sẽ được miêu tả bằng một tài liệu văn bản nhất định, cung cấp nhiều thông tin hơn về trách nhiệm cũng như khả năng của lớp này. Loại đặc tả này bình thường ra không được chỉ ra trong các biểu đồ, nhưng thường thì trong đa phần các công cụ mô hình hóa chúng sẽ có thể được truy cập qua hành động nhấp nút vào một phần tử nào đó, hiệu quả là một cửa sổ chứa đặc tả với tất cả các thuộc tính sẽ được chỉ ra (Hình 3.15).



Hình 3.15- Một cửa sổ đặc tả thể hiện các đặc tính của class

7- MỞ RỘNG UML

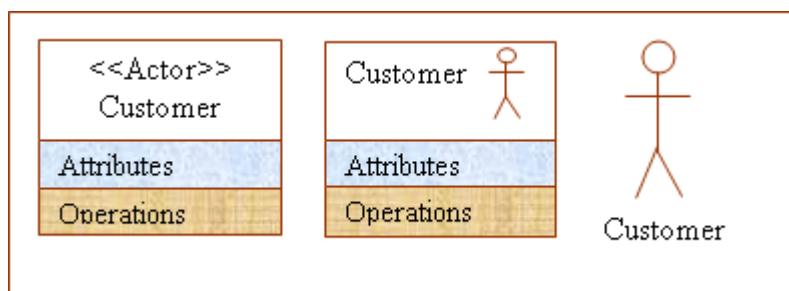
UML có thể được mở rộng hoặc có thể được sửa đổi để phù hợp với một phương pháp đặc biệt, một tổ chức cụ thể hay một người dùng cụ thể. Chúng ta sẽ bàn luận sơ qua đến ba cơ chế mở rộng UML: khuôn mẫu (stereotype), giá trị đính kèm (tagged value) và hạn chế (constraint).

7.1- Khuôn mẫu (Stereotype)

Cơ chế mở rộng khuôn mẫu định nghĩa một loại phần tử mô hình mới dựa trên một phần tử mô hình đã tồn tại. Khuôn mẫu có thể được coi là "tương tự" như một phần tử đã có sẵn, cộng thêm phần quy định ngữ nghĩa (semantic) riêng biệt không có trong phần tử gốc kia. Khuôn mẫu của một phần tử có thể được sử dụng trong cùng tình huống như phần tử căn bản. Khuôn mẫu dựa trên tất cả các loại phần tử mô hình sẵn có - lớp, nút mạng, thành phần, cũng như các mối quan hệ như liên kết, khái quát hóa, sự phụ thuộc. Ngôn ngữ UML có chứa một số lượng lớn các khuôn mẫu được định nghĩa sẵn và chúng được sử dụng để sửa đổi các phần tử mô hình sẵn có, thay cho việc phải định nghĩa hoàn toàn mới. Cơ chế này giúp gìn giữ tính đơn giản của nền tảng ngôn ngữ UML.

Khuôn mẫu được miêu tả qua việc đưa tên của chúng vào trong một cặp ký tự ngoặc nhọn <<>>, theo như trong hình 3.16. Ký tự ngoặc nhọn này được gọi là guillelements. Khuôn mẫu cũng có thể có kí hiệu hình học riêng. Một phần tử của một loại khuôn mẫu cụ thể có thể được thể hiện bởi tên khuôn mẫu đi kèm ký hiệu hình học mô tả phần tử căn bản, hay là sự kết hợp của cả hai yếu tố này. Bất kỳ khi nào một phần tử mô hình được nối kết với một tên hoặc kí hiệu khuôn mẫu, ta sẽ đọc "đây là một loại phần tử thuộc loại khuôn mẫu...". Ví dụ, một lớp với <<Window>> sẽ được gọi là "một lớp trong dạng khuôn mẫu cửa sổ", ý nghĩa của nó là một dạng lớp cửa sổ. Những thuộc tính cụ thể mà một lớp cửa sổ cần phải có sẽ được định nghĩa khi khuôn mẫu này được định nghĩa.

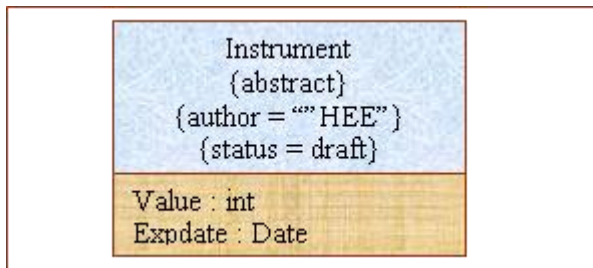
Như đã nói, khuôn mẫu là một cơ chế mở rộng xuất sắc, là một cơ chế ngăn cho ngôn ngữ UML không trở nên quá phức tạp, mặc dù vẫn cho phép thực hiện sự mở rộng và sửa đổi cần thiết. Đa phần các phần tử mô hình mới mà bạn cần đến đều có một khuôn mẫu nền tảng trong ngôn ngữ UML. Một khuôn mẫu sau đó có thể được sử dụng để cộng thêm các ngữ nghĩa cần thiết, nhằm mục đích định nghĩa nên các phần tử mô hình còn thiếu.



Hình 3.16- Customer là một lớp khuôn mẫu <<Actor>>

7.2- Giá trị đính kèm (Tagged Value)

Như đã nói, các phần tử mô hình có thể có các thuộc tính chứa một cặp tên-giá trị về bản thân chúng (hình 3.17). Các thuộc tính này cũng còn được gọi là các giá trị đính kèm. UML có chứa một loạt các thuộc tính được định nghĩa trước, nhưng kể cả người sử dụng cũng có thể định nghĩa ra các thuộc tính mới để chứa các thông tin bổ sung về các phần tử mô hình. Mọi hình dạng thông tin đều có thể được đính kèm vào phần tử: các thông tin chuyên biệt về phương pháp, các thông tin của nhà quản trị về tiến trình mô hình hóa, các thông tin được sử dụng bởi các công cụ khác, ví dụ như các công cụ tạo code, hay bất kỳ một loại thông tin nào mà người sử dụng muốn đính kèm vào phần tử mô hình.



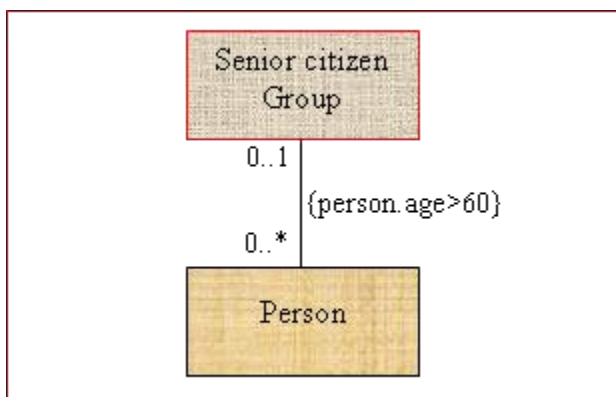
Hình 3.17 - Một ví dụ về Tagged Value

7.3- Hạn chế (Constraint)

Một sự hạn chế là một sự giới hạn về sự sử dụng hoặc ý nghĩa của một phần tử. Sự hạn chế hoặc sẽ được khai báo trong công cụ và được sử dụng nhiều lần trong rất nhiều biểu đồ khác nhau, hay được định nghĩa và sử dụng trong chỉ một biểu đồ, theo như nhu cầu.

Hình 3.18 chỉ ra mối quan hệ nối kết giữa nhóm các công dân lớn tuổi và lớp con người, chỉ ra rằng nhóm công dân có thể có nhiều người liên quan. Mặc dù vậy, để miêu tả rằng chỉ những người nào lớn hơn 60 tuổi mới có thể tham gia vào nhóm này, người ta định nghĩa một sự hạn chế, hạn hẹp tiêu chuẩn tham gia đối với chỉ những người nào mà thuộc tính tuổi tác có giá trị lớn hơn 60. Định nghĩa này sẽ hạn chế số lượng những người được sử dụng trong mỗi quan hệ. Nếu không có nó, người ta rất dễ hiểu lầm khi diễn tả biểu đồ. Trong trường hợp tồi tệ, nó có thể dẫn đến sự thực thi sai trái của hệ thống.

Trong trường hợp này, hạn chế được định nghĩa và ứng dụng trực tiếp trong chính biểu đồ mà nó được cần tới. Nhưng nhìn chung thì hạn chế cũng có thể được định nghĩa với tên cùng lời đặc tả riêng, ví dụ như: "công dân già" và "người có tuổi lớn hơn 60", và hạn chế này sẽ được sử dụng trong nhiều biểu đồ khác nhau. UML có chứa một loạt các hạn chế được định nghĩa sẵn, chúng được miêu tả chi tiết trong các chương sau.

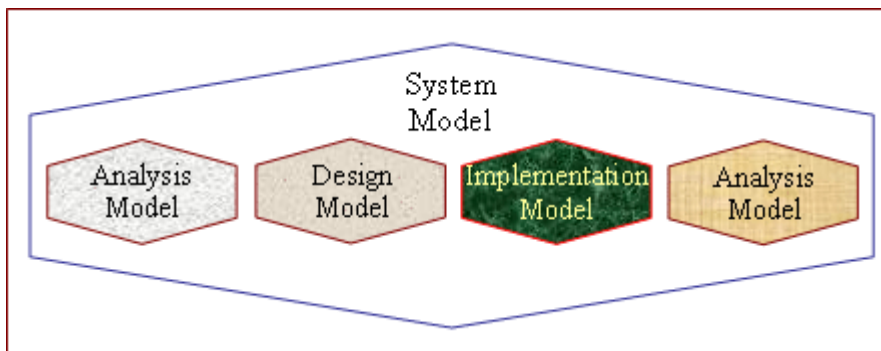


Hình 3.18- Một ràng buộc hạn chế đối tượng Person góp phần vào quan hệ kết hợp

8- MÔ HÌNH HÓA VỚI UML

Khi xây dựng hệ thống với UML, người ta không chỉ xây dựng duy nhất một mô hình. Sẽ có nhiều mô hình khác nhau trong những giai đoạn phát triển khác nhau, nhằm đến các mục đích khác nhau. Trong giai đoạn phân tích, mục đích của mô hình là nắm bắt tất cả các yêu cầu đối với hệ thống và mô hình hóa nền tảng bao gồm các lớp và các cộng tác "đời thực". Trong giai đoạn thiết kế, mục đích của mô hình là mở rộng mô hình phân tích, tạo thành một giải pháp kỹ thuật khả thi, có chú ý đến môi trường của công việc xây dựng (viết code). Trong giai đoạn xây dựng code, mô hình chính là những dòng code nguồn thật sự, được viết nên và được dịch thành các chương trình. Và cuối cùng, trong giai đoạn triển khai, một lời miêu tả sẽ giải thích hệ thống cần được triển khai ra sao trong kiến trúc vật lý. Khả năng theo dõi xuyên suốt nhiều giai đoạn và nhiều mô hình khác nhau được đảm bảo qua các thuộc tính hoặc các mối quan hệ nâng cao (refinement).

Mặc dù đó là các mô hình khác nhau, nhưng chúng đều được xây dựng nên để mở rộng nội dung của các mô hình ở giai đoạn trước. Chính vì thế, tất cả các mô hình đều cần phải được gìn giữ tốt để người ta có thể dễ dàng đi ngược lại, mở rộng ra hay tái thiết lập mô hình phân tích khởi đầu và rồi dần dần từng bước đưa các sự thay đổi vào mô hình thiết kế cũng như các mô hình xây dựng (hình 3.19).

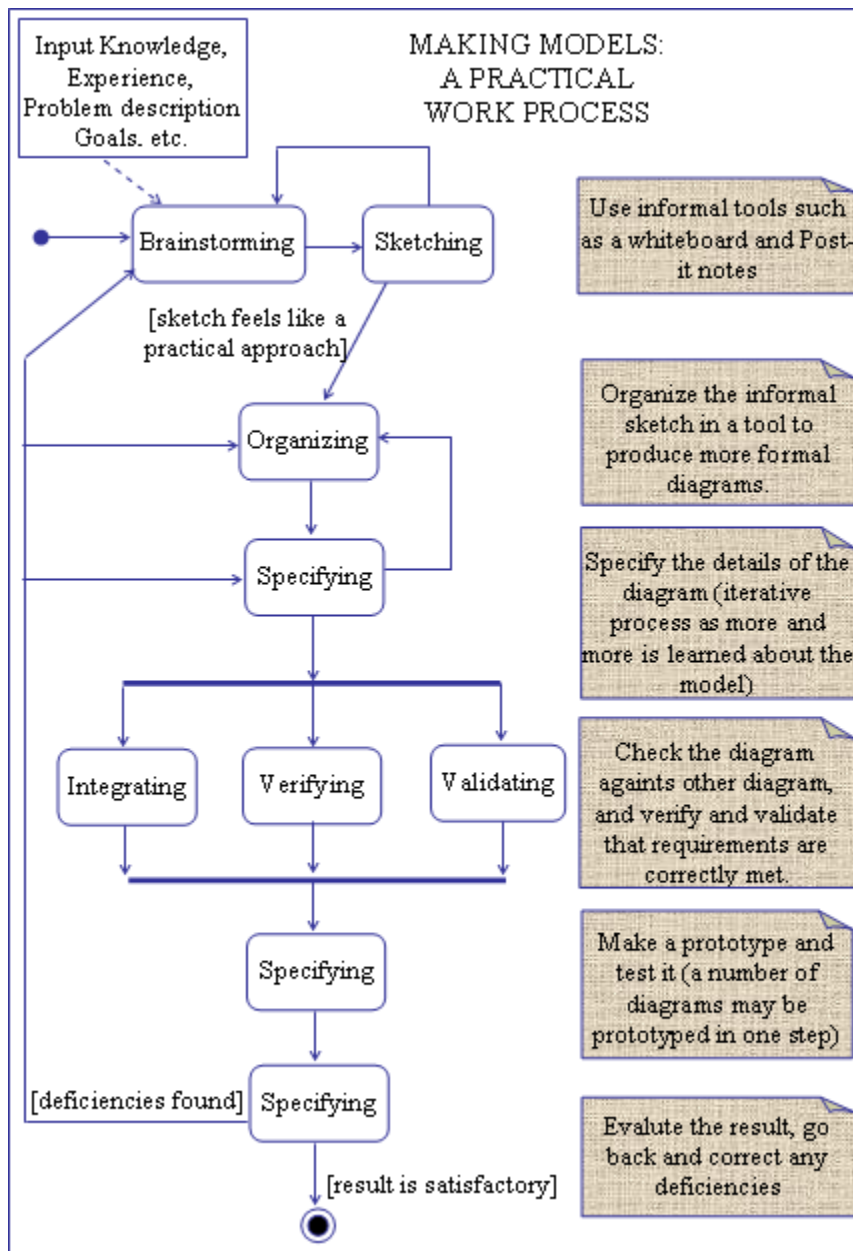


Hình 3.19- Một hệ thống được mô tả trong nhiều mô hình

Bản thân ngôn ngữ UML không phụ thuộc vào giai đoạn, có nghĩa là cũng những nguyên tắc ngôn ngữ đó và cũng những biểu đồ đó được sử dụng để mô hình hóa những sự việc khác nhau trong những giai đoạn khác nhau. Nhà thiết kế nắm quyền quyết định xem một mô hình sẽ phải thay đổi nhằm đạt được những mục đích nào và bao trùm những phạm vi nào. Ngôn ngữ mô hình hóa chỉ cung cấp khả năng để tạo ra các mô hình trong một phong cách mở rộng và nhất quán.

Khi mô hình hóa bằng ngôn ngữ UML, toàn bộ công việc cần phải được thực hiện theo một phương pháp hay một qui trình, xác định rõ những bước công việc nào phải được tiến hành và chúng phải được thực thi ra sao. Một qui trình như vậy

thường sẽ chia công việc ra thành các vòng lặp kế tiếp, mỗi vòng lặp bao gồm các công việc: *phân tích yêu cầu/ phân tích/ thiết kế/ thực hiện/ triển khai*. Mặc dù vậy, cũng có một quy trình nhỏ hơn đề cập tới nội dung của việc mô hình hóa. Bình thường ra, khi sản xuất một mô hình hoặc sản xuất chỉ một biểu đồ duy nhất, công việc sẽ bắt đầu bằng việc thu thập một nhóm thích hợp các cá nhân khác nhau, trình bày vấn đề và mục tiêu; họ cộng tác cho một giai đoạn hội thảo khoa học và phác thảo, trao đổi những sáng kiến và ý tưởng về mô hình có thể. Công cụ được sử dụng trong giai đoạn này là hết sức khác biệt và mang tính ngẫu hứng - thường là giấy dán *post it* hay bảng trắng. Công việc được quyết định chừng nào những người tham gia có cảm giác họ đã có được một nền tảng thực tiễn cho một mô hình (giống như một tiêu đề). Kết quả sau đó sẽ được đưa vào một công cụ, mô hình tiêu đề được tổ chức, và sau đó một biểu đồ thực sự sẽ được tạo dựng nên, phù hợp với những quy định của ngôn ngữ mô hình hóa. Sau đó, mô hình được chi tiết hóa qua những công việc mang tính vòng lặp, càng ngày càng có nhiều chi tiết về giải pháp được phát hiện, được dữ liệu hóa và được bổ sung. Khi đã có nhiều thông tin hơn được thu thập về vấn đề cũng như giải pháp của nó, tiêu đề ban đầu dần dần trở thành một lời chuẩn đoán cho một mô hình có khả năng sử dụng. Khi mô hình đã gần hoàn thiện, một sự tích hợp và thẩm định sẽ được thực hiện, dẫn tới việc mô hình hoặc biểu đồ sẽ được tích hợp với những mô hình và biểu đồ khác trong cùng dự án để đảm bảo sự nhất quán. Mô hình sau đó cũng được kiểm tra lại để chắc chắn nó đang giải quyết đúng vấn đề cần giải quyết (hình 3.20).



Hình 3.20 - Một tiến trình cho công việc mô hình hoá thực tế

Cuối cùng, mô hình sẽ được thực thi và triển khai thành một loạt các nguyên mẫu (prototype), nguyên mẫu này sẽ được kiểm tra để tìm khiếm khuyết. Các khiếm khuyết bao gồm kể cả các chức năng còn thiếu, sự thực hiện tồi tệ hay phí sản xuất và phát triển quá cao. Những khiếm khuyết thường sẽ ép nhà phát triển rà đi rà lại công việc của mình để khắc phục chúng. Nếu vấn đề là quá lớn, nhà phát triển có thể sẽ đi ngược lại tất cả các bước công việc của mình cho tới tận giai đoạn sơ phác đầu tiên. Nếu các vấn đề này không lớn, nhà phát triển có lẽ chỉ cần thay đổi một vài thành phần trong tổ chức hoặc đặc tả của mô hình. Xin nhớ rằng bước tạo nguyên mẫu không thể được thực hiện ngay lập tức sau khi hoàn tất biểu đồ; nó chỉ nên được thực hiện khi đã có một số lượng lớn các biểu

đồ liên quan. Nguyên mẫu sau này có thể được vứt đi, có thể được tạo dựng nên chỉ để nhằm mục đích kiểm tra, hoặc là nếu bước tạo nguyên mẫu này thành công, nó sẽ trở thành một vòng lặp trong quy trình phát triển thật sự.

9- CÔNG CỤ (TOOL)

Sử dụng một ngôn ngữ mô hình hóa phức tạp và rộng mở như UML cần thiết sự trợ giúp của công cụ. Mặc dù phác thảo đầu tiên của một mô hình có thể được thực hiện bằng bảng trắng cùng giấy và mực, nhưng công việc bảo trì, đồng bộ hóa và đảm bảo sự nhất quán trong một loạt các biểu đồ khác nhau thường lại không thể trở thành khả thi nếu không có công cụ.

Thị trường công cụ mô hình hóa đã dừng trong mức độ sơ khởi suốt một thời gian dài kể từ khi xuất hiện ý tưởng đầu tiên về các chương trình trợ giúp cho việc tạo chương trình. Rất nhiều công cụ trong thực tế chỉ thông minh hơn các chương trình vẽ một chút, sử dụng một vài quy chế kiểm tra tính nhất quán hoặc một vài kiến thức về phương pháp và ngôn ngữ mô hình hóa. Mặc dù đã có một vài bước tiến nhất định và nhiều công cụ hôm nay đã tới gần sáng kiến khởi thủy kia nhiều hơn (Rational Rose), nhưng thị trường vẫn còn không ít công cụ chưa được gọt giũa, vẫn còn chứa lỗi hoặc những nét kỳ quặc, kể cả những vấn đề đơn giản như copy và dán. Những công cụ này còn hạn chế ở phương diện rằng tất cả bọn chúng đều có ngôn ngữ mô hình hóa riêng, hay ít nhất thì cũng có những định nghĩa riêng của chúng về ngôn ngữ này.

Cùng với sự ra đời của ngôn ngữ UML, các nhà cung cấp công cụ mô hình hóa giờ đây có thể dành nhiều thời gian hơn cho việc nâng cấp công cụ, bởi họ không cần phải dồn tâm dồn sức cho việc định nghĩa các phương pháp mới cũng như các ngôn ngữ mới.

Một công cụ mô hình hóa hiện đại cần phải cung cấp các chức năng sau:

🎯 **Vẽ biểu đồ:** cần phải tạo điều kiện dễ dàng vẽ ra các biểu đồ trong ngôn ngữ mô hình hóa. Công cụ cần phải đủ khả năng thông minh để hiểu mục đích của các biểu đồ và biết được những ngữ nghĩa cũng như các quy tắc đơn giản, đủ để nó có thể cảnh báo hoặc ngăn chặn việc sử dụng không thích hợp các phần tử mô hình.

🎯 **Hoạt động như một nhà kho (Repository):** công cụ cần phải hỗ trợ một nhà kho trung tâm để tất cả các thông tin về mô hình được lưu trữ trong cùng một chỗ. Nếu ví dụ tên của một lớp bị thay đổi trong một biểu đồ, thì sự thay đổi này cần phải xảy ra trong tất cả các biểu đồ khác có sử dụng lớp này.

🔗➡️**Hỗ trợ định hướng (Navigation):** công cụ cần phải tạo điều kiện dễ dàng cho người sử dụng định hướng và chuyển dịch trong mô hình để theo dõi một phần tử từ biểu đồ này sang biểu đồ khác, hoặc để mở rộng lời miêu tả của một phần tử.

🔗➡️**Hỗ trợ nhiều người sử dụng (multiuser support):** Công cụ cần hỗ trợ cho nhiều người sử dụng, và tạo điều kiện cho họ cùng làm việc với một mô hình mà không ngăn chặn hoặc quấy phá lẫn nhau.

🔗➡️**Tự động tạo code (code generate):** một công cụ cao cấp cần phải có khả năng tạo ra code, nơi tất cả các thông tin trong mô hình được chuyển tải thành các khung code (code skeletons), được sử dụng làm nền tảng cho giai đoạn xây dựng chương trình.

🔗➡️**Tái tạo mô hình (Reverse engineer):** Một công cụ cao cấp cần phải có khả năng đọc những thành phần code đang tồn tại và từ đó sản xuất ra mô hình. Từ đó suy ra, một mô hình có thể được làm từ những dòng code đã tồn tại; hoặc một nhà phát triển có thể dễ dàng chuyển đi chuyển về giữa công việc mô hình hóa và công việc lập trình.

🔗➡️**Tích hợp với các công cụ khác:** một công cụ cần phải có khả năng tích hợp với những công cụ khác, với cả việc phát triển môi trường, ví dụ như các trình soạn thảo (editor), chương trình dịch (compiler), chương trình tìm lỗi (debugger) cũng như các công cụ của doanh nghiệp khác như công cụ quản trị cấu hình, hệ thống theo dõi các phiên bản.

🔗➡️**Bao quát mô hình ở tất cả các mức độ trừu tượng hóa khác nhau:** công cụ cần phải dễ chuyển tải từ lời miêu tả ở cấp trừu tượng hóa cao nhất của hệ thống (tức là ở dạng một lượng các gói khác nhau) đi xuống cho tới cấp của những dòng code thật sự. Sau đó, để truy xuất những dòng lệnh code cho một thủ tục cụ thể nào đó trong một lớp nào đó, bạn có thể chỉ cần nhấp chuột vào tên của thủ tục đó trong một biểu đồ.

🔗➡️**Trao đổi mô hình:** Một mô hình hay một biểu đồ của một mô hình nào đó cần phải có khả năng được xuất ra từ một công cụ này rồi nhập vào một công cụ khác, giống như những dòng lệnh code được sản sinh trong một công cụ này có thể được sử dụng trong một công cụ khác. Nguyên tắc trao đổi đó cần phải được áp dụng cho

các mô hình trong một ngôn ngữ mô hình hóa được định nghĩa chính xác.

10- TÓM TẮT VỀ UML

UML tổ chức một mô hình thành một loạt các hướng nhìn, thể hiện các khía cạnh khác nhau của hệ thống. Chỉ khi kết hợp tất cả các hướng nhìn lại với nhau, người ta mới có được một bức tranh trọn vẹn về hệ thống. Một hướng nhìn không phải là một hình vẽ, nội dung của nó được miêu tả qua các biểu đồ, đây là những hình vẽ chứa đựng các phần tử mô hình hóa. Một biểu đồ bình thường chỉ trình bày một phần nội dung của một hướng nhìn, và một hướng nhìn được định nghĩa với rất nhiều biểu đồ. Một biểu đồ chứa các phần tử mô hình, ví dụ như lớp, đối tượng, nút mạng, thành phần và những mối quan hệ như nối kết, khái quát hóa, phụ thuộc. Các phần tử này có ý nghĩa (semantic) và các ký hiệu hình học.

Các loại biểu đồ trong UML là: biểu đồ lớp, biểu đồ đối tượng, biểu đồ Use case, biểu đồ trạng thái, biểu đồ trình tự, biểu đồ cộng tác, biểu đồ hành động, biểu đồ thành phần và biểu đồ triển khai. Mục đích của các loại biểu đồ cũng như quy tắc vẽ chúng sẽ được miêu tả chi tiết trong chương sau.

UML có một số cơ chế chung để bổ sung thông tin không thể được thể hiện trong quá trình vẽ biểu đồ. Những thông tin này bao gồm ví dụ những thành phần trang trí, các lời ghi chú có thể chứa bất kỳ loại thông tin nào cũng như các thuộc tính đặc tả. Ngoài ra còn có các cơ chế mở rộng, bao gồm giá trị đính kèm, hạn chế đối với phần tử, và khuôn mẫu, định nghĩa một loại phần tử mô hình mới dựa trên một phần tử sẵn có.

Một hệ thống sẽ được miêu tả trong nhiều loại mô hình khác nhau, mỗi loại mô hình nhằm một mục đích khác nhau. Mô hình phân tích miêu tả những yêu cầu về mặt chức năng và mô hình hóa các lớp ngoài đời thực. Mô hình thiết kế chuyển tải kết quả phân tích thành một giải pháp kỹ thuật, theo khái niệm của một thiết kế phần mềm hoạt động hoàn chỉnh. Mô hình xây dựng code thể hiện hệ thống qua việc thảo chương cho nó trong một ngôn ngữ lập trình hướng đối tượng. Và cuối cùng, mô hình triển khai định vị chương trình vừa được tạo nên trong một kiến trúc vật lý bao gồm các máy tính và các trang thiết bị. Công việc được làm theo nhiều vòng lặp khác nhau chứ không phải chỉ là một chuỗi thực hiện một lần.

Để sử dụng UML một cách nghiêm chỉnh cho một dự án có thật ngoài đời, bạn cần công cụ. Một công cụ tân tiến có khả năng cho người dùng vẽ biểu đồ, trữ tất cả các thông tin vào một kho chung, cho phép dễ dàng dịch chuyển giữa các hướng nhìn và biểu đồ khác nhau trong mô hình, tạo báo cáo và tài liệu, tạo

khung code từ mô hình, đọc những dòng code sẵn có rồi sản sinh ra mô hình từ đó, và dễ dàng tích hợp với các công cụ phát triển khác.

PHẦN CÂU HỎI

❏ **Hỏi:** UML có công cụ nào giúp nắm bắt các yêu cầu của khách hàng (người sử dụng)?

▶ **Đáp:** Use Case

❏ **Hỏi:** Một biểu đồ trong UML có bao chứa các hướng nhìn khác nhau.

▶ **Đáp:** Sai, một hướng nhìn bao gồm một loại các biểu đồ khác nhau

❏ **Hỏi:** Hãy liệt kê các thành phần chủ yếu của ngôn ngữ UML

▶ **Đáp:** Hướng nhìn(View), Biểu đồ (Diagram), Phần tử mô hình, Cơ chế chung.

❏ **Hỏi:** UML có công cụ nào phục vụ cho giai đoạn thử nghiệm đơn vị (Unit Testing)?

▶ **Đáp:** Biểu đồ lớp và đặc tả lớp

❏ **Hỏi:** UML có công cụ nào phục vụ cho giai đoạn thử nghiệm hệ thống (System Testing)?

▶ **Đáp:** Use case Diagram

❏ **Hỏi:** UML tạo nền tảng cho việc giao tiếp giữa khách hàng, nhà phân tích, nhà thiết kế và lập trình viên.

▶ **Đáp:** Đúng

□ □ □

Chương 4: Mô hình hóa USE CASE

1- GIỚI THIỆU USE CASE

Trong giai đoạn phân tích, người sử dụng cộng tác cùng nhóm phát triển phần mềm tạo nên một tổ hợp thông tin quan trọng về yêu cầu đối với hệ thống. Không chỉ là người cung cấp thông tin, bản thân người sử dụng còn là một thành phần hết sức quan trọng trong bức tranh toàn cảnh đó và nhóm phát triển cần phải chỉ ra được phương thức hoạt động của hệ thống tương lai theo hướng nhìn của người sử dụng. Hiểu được điểm quan trọng này là chìa khóa để tạo dựng được những hệ thống vừa thoả mãn các yêu cầu đặt ra vừa dễ dàng sử dụng, thậm chí tạo niềm vui thích trong sử dụng.

Như vậy công cụ giúp ta mô hình hoá hệ thống từ hướng nhìn của người sử dụng gọi là Use Case. Và để trả lời rõ hơn về Use Case ta xét một trường hợp sau:

Giả sử tôi quyết định mua một chiếc máy fax mới. Khi đến cửa hàng máy văn phòng, tôi mới nhận ra là phải chọn lựa trong một danh sách máy móc rất phong phú. Loại máy nào sẽ được chọn đây? Tôi tự hỏi thật chính xác mình muốn làm gì với chiếc máy fax sẽ mua? Tôi muốn có những tính năng nào? Tôi muốn dùng bằng giấy thường hay giấy thermal ? Tôi muốn copy bằng cái máy đó? Tôi muốn nối nó với máy tính của mình? Tôi muốn dùng nó vừa làm máy fax vừa làm scanner? Tôi có cần phải gửi fax thật nhanh đến mức độ cần một chức năng chọn số tăng tốc? Liệu tôi có muốn sử dụng máy fax này để phân biệt giữa một cú điện thoại gọi tới và một bản fax gửi tới ?.

Tất cả chúng ta đều trải qua những kinh nghiệm như vậy khi quyết định mua một món hàng nào đó không phải vì niềm vui bộc phát. Việc chúng ta sẽ làm trong những trường hợp như vậy là một dạng phân tích Use Case: Chúng ta tự hỏi mình sẽ sử dụng sản phẩm (hay hệ thống) sắp bắt ta bỏ ra một khoản tiền đáng kể đó ra sao? Trả lời xong câu hỏi trên ta mới có khả năng chọn ra sản phẩm thoả mãn những đòi hỏi của mình. Điều quan trọng ở đây là phải biết những đòi hỏi đó là gì.

Loại quy trình này đóng vai trò rất quan trọng đối với giai đoạn phân tích của một nhóm phát triển hệ thống. Người dùng muốn sử dụng hệ thống tương lai, hệ thống mà bạn sắp thiết kế và xây dựng, như thế nào?

Use Case là một công cụ trợ giúp cho công việc của nhà phân tích cùng người sử dụng quyết định tính năng của hệ thống. Một tập hợp các Use Case sẽ làm nổi bật một hệ thống theo phương diện những người dùng định làm gì với hệ thống này.

Để làm rõ hơn, ta hãy xét một ví dụ nhà băng lẻ. Hệ thống tương lai trong trường hợp này sẽ có nhiều người sử dụng, mỗi người sẽ giao tiếp với hệ thống cho một mục đích khác biệt:

- Quản trị gia sử dụng hệ thống cho mục đích thống kê
- Nhân viên tiếp khách sử dụng hệ thống để thực hiện các dịch vụ phục vụ khách hàng.
- Nhân viên phòng đầu tư sử dụng hệ thống để thực hiện các giao dịch liên quan đến đầu tư.
- Nhân viên thẩm tra chữ ký sử dụng hệ thống cho mục đích xác nhận chữ ký và bảo trì thông tin liên quan đến khách hàng.
- Khách hàng giao tiếp với hệ thống (nhà băng) cho các hoạt động sử dụng dịch vụ như mở tài khoản, gửi tiền vào, rút tiền mặt, ...

Quá trình tương tác giữa người sử dụng và hệ thống trong mỗi một tình huống kể trên sẽ khác nhau và phụ thuộc vào chức năng mà người sử dụng muốn thực thi cùng hệ thống.

Nhóm phát triển hệ thống cần phải xây dựng nên một kịch bản nêu bật sự tương tác cần thiết giữa người sử dụng và hệ thống trong mỗi khả năng hoạt động. Ví dụ như kịch bản cho sự tương tác giữa nhân viên thu ngân và hệ thống của bộ phận tiết kiệm trong suốt tiến trình của một giao dịch. Một kịch bản khác ví dụ là chuỗi tương tác xảy ra giữa bộ phận tiết kiệm và bộ phận đầu tư trong một giao dịch chuyển tiền.

Nhìn chung, có thể coi một Use case như là tập hợp của một loạt các cảnh kịch về việc sử dụng hệ thống. Mỗi cảnh kịch mô tả một chuỗi các sự kiện. Mỗi một chuỗi này sẽ được kích hoạt bởi một người nào đó, một hệ thống khác hay là một phần trang thiết bị nào đó, hoặc là một chuỗi thời gian. Những thực thể kích hoạt nên các chuỗi sự kiện như thế được gọi là các **Tác Nhân (Actor)**. Kết quả của chuỗi này phải có giá trị sử dụng đối với hoặc là tác nhân đã gây nên nó hoặc là một tác nhân khác.

2- MỘT SỐ VÍ DỤ USE CASE

Trong ví dụ nhà băng lẻ ở trên, một số những Use Case dễ thấy nhất là:

- Một khách hàng mở một tài khoản mới.
- Phòng đầu tư tính toán tiền lãi cho các tài khoản đầu tư.
- Một chương trình đầu tư mới được đưa vào áp dụng.

- Yêu cầu chuyển tiền của khách hàng được thực hiện.
- Chuyển tiền theo kỳ hạn từ một tài khoản đầu tư sang một tài khoản tiết kiệm.

3- SỰ CẦN THIẾT PHẢI CÓ USE CASE

Use Case là một công cụ xuất sắc để khuyến khích những người dùng tiềm năng nói về hệ thống từ hướng nhìn của họ. Đối với người dùng, chẳng phải bao giờ việc thể hiện và mô tả những ý định trong việc sử dụng hệ thống cũng là chuyện dễ dàng. Một hiện thực có thật là người sử dụng thường biết nhiều hơn những gì mà họ có thể diễn tả ra: Công cụ Use Case sẽ giúp cho nhóm phát triển bẻ gãy "lớp băng" đó, ngoài ra một sự trình bày trực quan cũng cho phép bạn kết hợp các biểu đồ Use Case với các loại biểu đồ khác.

Sáng kiến chủ đạo là lôi cuốn được người dùng tham gia vào những giai đoạn đầu tiên của quá trình phân tích và thiết kế hệ thống. Việc này sẽ nâng cao xác suất cho việc hệ thống chung cuộc trở thành một công cụ quen thuộc đối với các người dùng mà nó dự định sẽ trợ giúp – thay vì là một tập hợp khó hiểu và rối rắm của các khái niệm máy tính mà người dùng trong giới doanh thương có cảm giác không bao giờ hiểu được và không thể làm việc cùng.

Công tác lôi kéo người sử dụng tham gia tích cực vào quá trình phân tích là nền tảng quan trọng cho việc tạo dựng một mô hình "thành công", một mô hình dễ được người sử dụng hiểu và chấp nhận sau khi đã thẩm xác các nhiệm vụ căn bản. Ngoài ra, Use Case còn giúp nhóm phát triển quyết định các lớp mà hệ thống phải triển khai.

4- MÔ HÌNH HÓA USE CASE

Trường hợp sử dụng là một kỹ thuật mô hình hóa được sử dụng để mô tả một hệ thống mới sẽ phải làm gì hoặc một hệ thống đang tồn tại làm gì. Một mô hình Use Case được xây dựng qua một quá trình mang tính vòng lặp (iterative), trong đó những cuộc hội thảo bàn luận giữa nhóm phát triển hệ thống và khách hàng (hoặc/và người sử dụng cuối) sẽ dẫn tới một đặc tả yêu cầu được tất cả mọi người chấp nhận. Người cha tinh thần của mô hình hóa Use Case là Ivar Jacobson, ông đã tạo nên kỹ thuật mô hình hóa dựa trên những kinh nghiệm thu thập được trong quá trình tạo hệ thống AXE của hãng Ericsson. Use Case đã nhận được một sự quan tâm đặc biệt lớn lao từ phía cộng đồng hướng đối tượng và đã tác động lên rất nhiều phương pháp hướng đối tượng khác nhau.

Những thành phần quan trọng nhất của một mô hình Use Case là Use Case, tác nhân và hệ thống. Ranh giới của hệ thống được định nghĩa qua chức năng tổng thể mà hệ thống sẽ thực thi. Chức năng tổng thể được thể hiện qua một loạt các

Use Case và mỗi một Use Case đặc tả một chức năng trọn vẹn, có nghĩa là Use Case phải thực thi toàn bộ chức năng đó, từ sự kiện được kích hoạt đầu tiên bởi một tác nhân ngoại cảnh cho tới khi chức năng đòi hỏi được thực hiện hoàn tất. Một Use Case luôn luôn phải cung cấp một giá trị nào đó cho một tác nhân, giá trị này là những gì mà tác nhân mong muốn từ phía hệ thống. Tác nhân là bất kỳ một thực thể ngoại cảnh nào mong muốn tương tác với hệ thống. Thường thường, đó là một người sử dụng của hệ thống, nhưng nhiều khi cũng có thể là một hệ thống khác hoặc là một dạng máy móc thiết bị phần cứng nào đó cần tương tác với hệ thống.

Trong kỹ thuật mô hình hóa Use Case, hệ thống sẽ có hình dạng của một "hộp đen" và cung cấp các Use Case. Hệ thống làm điều đó như thế nào, các Use Case được thực thi ra sao, đó là những khía cạnh chưa được đề cập tới trong giai đoạn này. Trong thực tế, nếu mô hình hóa Use Case được thực hiện trong những giai đoạn đầu của dự án thì thường nhà phát triển sẽ không biết Use Case sau này sẽ được thực thi (tức là biến thành những dòng code thật sự) như thế nào.

Mục tiêu chính yếu đối với các Use Case là:

- Để quyết định và mô tả các yêu cầu về mặt chức năng của hệ thống, đây là kết quả rút ra từ sự thỏa thuận giữa khách hàng (và/hoặc người sử dụng cuối) và nhóm phát triển phần mềm.
- Để tạo nên một lời mô tả rõ ràng và nhất quán về việc hệ thống cần phải làm gì, làm sao để mô hình có thể được sử dụng nhất quán suốt toàn bộ quá trình phát triển, được sử dụng làm công cụ giao tiếp cho tất cả những người phát triển nên các yêu cầu này, và để tạo nên một nền tảng cho việc tạo nên các mô hình thiết kế cung cấp các chức năng được yêu cầu.
- Để tạo nên một nền tảng cho các bước thử nghiệm hệ thống, đảm bảo hệ thống thỏa mãn đúng những yêu cầu do người sử dụng đưa ra. Trong thực tế thường là để trả lời câu hỏi: Liệu hệ thống cuối cùng có thực hiện những chức năng mà khởi đầu khách hàng đã đề nghị?
- Để cung cấp khả năng theo dõi các yêu cầu về mặt chức năng được chuyển thành các lớp cụ thể cũng như các thủ tục cụ thể trong hệ thống.
- Để đơn giản hóa việc thay đổi và mở rộng hệ thống qua việc thay đổi và mở rộng mô hình Use Case, sau đó chỉ theo dõi riêng những Use Case đã bị thay đổi cùng những hiệu ứng của chúng trong thiết kế hệ thống và xây dựng hệ thống.

Những công việc cụ thể cần thiết để tạo nên một mô hình Use Case bao gồm:

1. Định nghĩa hệ thống (xác định phạm vi hệ thống)
2. Tìm ra các tác nhân cũng như các Use Case
3. Mô tả Use Case
4. Định nghĩa mối quan hệ giữa các Use Case
5. Kiểm tra và phê chuẩn mô hình.

Đây là một công việc mang tính tương tác rất cao, bao gồm những cuộc thảo luận với khách hàng và những người đại diện cho các loại tác nhân. Mô hình Use Case bao gồm các biểu đồ Use Case chỉ ra các tác nhân, Use Case và mối quan hệ của chúng với nhau. Các biểu đồ này cho ta một cái nhìn tổng thể về mô hình, nhưng những lời mô tả thực sự của từng Use Case thường lại là văn bản. Vì các mô hình trực quan không thể cung cấp tất cả các thông tin cần thiết, nên cần thiết phải dùng cả hai kỹ thuật trình bày đó.

Có rất nhiều người quan tâm đến việc sử dụng các mô hình Use Case. Khách hàng (và/hoặc người sử dụng cuối) quan tâm đến chúng vì mô hình Use Case đặc tả chức năng của hệ thống và mô tả xem hệ thống có thể và sẽ được sử dụng ra sao. Các Use Case vì vậy phải được mô tả trong những thuật ngữ và ngôn ngữ của khách hàng/người sử dụng.

Nhà phát triển cần đến các mô hình Use Case để hiểu hệ thống cần phải làm gì, và qua đó có được một nền tảng cho những công việc tương lai (các mô hình khác, các cấu trúc thiết kế và việc thực thi xây dựng hệ thống bằng code).

Các nhóm chuyên gia thử nghiệm tích hợp và thử nghiệm hệ thống cần đến Use Case để thử nghiệm và kiểm tra xem hệ thống có đảm bảo sẽ thực hiện đúng chức năng đã được đặc tả trong giai đoạn đầu.

Và cuối cùng, bất kỳ người nào liên quan đến những hoạt động liên kết đến chức năng của hệ thống đều có thể quan tâm đến các mô hình Use Case; ví dụ như các nhóm tiếp thị, bán hàng, hỗ trợ khách hàng và các nhóm soạn thảo tài liệu.

Mô hình Use Case mô tả hướng nhìn Use Case của hệ thống. Hướng nhìn này là rất quan trọng, bởi nó ảnh hưởng đến tất cả các hướng nhìn khác của hệ thống. Cả cấu trúc logic lẫn cấu trúc physic đều chịu ảnh hưởng từ các Use Case, bởi chức năng được đặc tả trong mô hình này chính là những chức năng được thực thi trong các cấu trúc kia. Mục đích cuối cùng là thiết kế ra một giải pháp thỏa mãn các yêu cầu đó.

Mô hình hóa các Use Case chẳng phải chỉ được dùng để nắm bắt các yêu cầu của hệ thống mới; nó cũng còn được sử dụng để hỗ trợ cho việc phát triển một phiên

bản mới của hệ thống. Khi phát triển một phiên bản mới của hệ thống đang tồn tại, người ta sẽ bổ sung thêm các chức năng mới vào mô hình Use Case đã có bằng cách thêm vào các tác nhân mới cũng như các Use Case mới, hoặc là thay đổi đặc tả của các Use Case đã có. Khi bổ sung thêm vào mô hình Use Case đang tồn tại, hãy chú ý để không bỏ ra bất kỳ một chức năng nào vẫn còn được cần tới.

5- BIỂU ĐỒ USE CASE

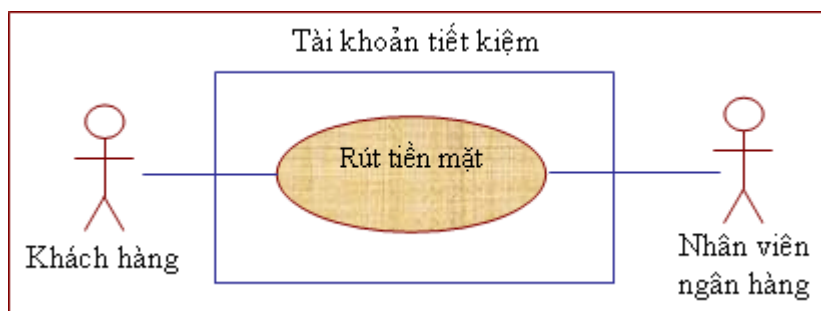
Use Case được mô tả trong ngôn ngữ UML qua biểu đồ Use Case (Use Case Diagram), và một mô hình Use Case có thể được chia thành một số lượng lớn các biểu đồ như thế. Một biểu đồ Use Case chứa các phần tử mô hình biểu thị hệ thống, tác nhân cũng như Use Case và chỉ ra các mối quan hệ giữa các Use Case.

Lời mô tả nội dung Use Case thường được cung cấp dưới dạng văn bản. Trong UML, lời mô tả đó được coi là thuộc tính "văn bản" (document) của Use Case. Lời mô tả này bao chứa những thông tin quan trọng, định nghĩa các yêu cầu và chức năng cụ thể. Thay cho việc mô tả Use Case bằng văn bản, bạn cũng có thể vẽ một biểu đồ hoạt động (activity diagram). Mặc dầu vậy, nên nhớ rằng một Use Case cần phải được mô tả sao cho dễ hiểu và dễ giao tiếp đối với người sử dụng, mà những cấu trúc phức tạp như một biểu đồ hoạt động có thể gây cảm giác xa lạ đối với những người không quen sử dụng.

Tóm tắt: Một biểu đồ Use Case thể hiện:

- Hệ thống
- Tác nhân và
- Use Case.

Ví dụ biểu đồ Use Case trong UML:



Hình 4.1- Một ví dụ biểu đồ Use case trong UML

Trong đó:

- Hệ thống được thể hiện qua hình chữ nhật với tên hệ thống ở bên trên

- Tác nhân được thể hiện qua kí hiệu hình nhân
- Use Case được thể hiện qua hình ellipse

◆ 5.1- Hệ thống:

Vì hệ thống là một thành phần của mô hình Use Case nên ranh giới của hệ thống mà ta muốn phát triển cần phải được định nghĩa rõ ràng. Xin nhớ rằng một hệ thống không phải bao giờ cũng nhất thiết là một hệ thống phần mềm; nó có thể là một chiếc máy, hoặc là một doanh nghiệp. Định nghĩa các ranh giới và trách nhiệm của hệ thống không phải bao giờ cũng là việc dễ dàng, bởi không phải bao giờ người ta cũng rõ ràng nhìn ra tác vụ nào có khả năng được tự động hóa tốt nhất ở hệ thống này và tác vụ nào thì tốt nhất nên thực hiện thủ công hoặc dành cho các hệ thống khác. Một khía cạnh khác cần chú ý là hệ thống cần phải lớn tới mức độ nào trong phiên bản đầu tiên của nó. Cố gắng tối đa cho phiên bản đầu tiên của hệ thống thường là cách mà người ta hay thực hiện, thế nhưng những mục tiêu quá tầm như vậy có thể khiến cho hệ thống trở nên quá lớn và thời gian để cung cấp hệ thống quá lâu. Một sáng kiến tốt hơn là xác nhận cho rõ các chức năng cần bản và tập trung vào việc định nghĩa một kiến trúc hệ thống thích hợp, rõ ràng, có nền tảng rộng mở để nhiều chức năng hơn có thể được bổ sung vào hệ thống này trong các phiên bản sau.

Yếu tố quan trọng là bạn phải tạo dựng được một bản catalog của các khái niệm (các thực thể) trung tâm cùng với các thuật ngữ và định nghĩa thích hợp trong những giai đoạn đầu của thời kỳ phân tích. Đây chưa phải mô hình phạm vi đối tượng, mà đúng hơn là một cố gắng để mô tả các thuật ngữ của hệ thống hoặc doanh nghiệp mà chúng ta cần mô hình hóa. Các thuật ngữ sau đó sẽ được dùng để mô tả Use Case. Phương thức cụ thể của catalog này có thể rất khác nhau; nó có thể là một mô hình khái niệm chỉ ra các mối quan hệ đơn giản hoặc chỉ là một văn bản chứa các thuật ngữ cùng lời mô tả văn tắt những thuật ngữ này trong thế giới thực.

◆ 5.2- Tác nhân:

Một tác nhân là một người hoặc một vật nào đó tương tác với hệ thống, sử dụng hệ thống. Trong khái niệm "tương tác với hệ thống", ý chúng ta muốn nói rằng tác nhân sẽ gửi thông điệp đến hệ thống hoặc là nhận thông điệp xuất phát từ hệ thống, hoặc là thay đổi các thông tin cùng với hệ thống. Nói một cách ngắn gọn, tác nhân thực hiện các Use Case. Thêm một điều nữa, một tác nhân có thể là người mà cũng có thể là một hệ thống khác (ví dụ như là một chiếc máy tính khác được nối kết với hệ thống của chúng ta hoặc một loại trang thiết bị phần cứng nào đó tương tác với hệ thống).

Một tác nhân là một dạng thực thể (một lớp), chứ không phải một thực thể. Tác nhân mô tả và đại diện cho một vai trò, chứ không phải là một người sử dụng thật sự và cụ thể của hệ thống. Nếu một anh chàng John nào đó muốn mua hợp đồng bảo hiểm từ một hãng bảo hiểm, thì vai trò của anh ta sẽ là người mua hợp đồng bảo hiểm, và đây mới là thứ mà chúng ta muốn mô hình hóa, chứ không phải bản thân anh chàng John. Trong sự thực, một con người cụ thể có thể đóng vai trò làm nhiều tác nhân trong một hệ thống: một nhân viên ngân hàng đồng thời cũng có thể là khách hàng của chính ngân hàng đó. Mặt khác, số lượng các vai trò mà một con người cụ thể được phép đảm trách trong một hệ thống cũng có thể bị hạn chế, ví dụ cùng một người không được phép vừa soạn hóa đơn vừa phê duyệt hóa đơn đó. Một tác nhân sẽ có một tên, và cái tên này cần phải phản ánh lại vai trò của tác nhân. Cái tên đó không được phản ánh lại một thực thể riêng biệt của một tác nhân, mà cũng không phản ánh chức năng của tác nhân đó.

Một tác nhân giao tiếp với hệ thống bằng cách gửi hoặc là nhận thông điệp, giống như khái niệm chúng ta đã quen biết trong lập trình hướng đối tượng. Một Use Case bao giờ cũng được kích hoạt bởi một tác nhân gửi thông điệp đến cho nó. Khi một Use Case được thực hiện, Use Case có thể gửi thông điệp đến một hay là nhiều tác nhân. Những thông điệp này cũng có thể đến với các tác nhân khác, bên cạnh chính tác nhân đã kích hoạt và gây ra Use Case.

Tác nhân cũng có thể được xếp loại. Một tác nhân chính (Primary Actor) là tác nhân sử dụng những chức năng căn bản của hệ thống, tức là các chức năng chính. Ví dụ, trong một hệ thống bảo hiểm, một tác nhân căn bản có thể là tác nhân xử lý việc ghi danh và quản lý các hợp đồng bảo hiểm. Một tác nhân phụ (secondary actor) là tác nhân sử dụng các chức năng phụ của hệ thống, ví dụ như các chức năng bảo trì hệ thống như quản trị ngân hàng dữ liệu, giao tiếp, back-up và các tác vụ quản trị khác. Một ví dụ cho tác nhân phụ có thể là nhà quản trị hoặc là một nhân viên sử dụng chức năng trong hệ thống để rút ra các thông tin thống kê về doanh nghiệp. Cả hai loại tác nhân này đều được mô hình hóa để đảm bảo mô tả đầy đủ các chức năng của hệ thống, mặc dù các chức năng chính mới thật sự nằm trong mối quan tâm chủ yếu của khách hàng.

Tác nhân còn có thể được định nghĩa theo dạng tác nhân chủ động (active actor) hay tác nhân thụ động (passive actor). Một tác nhân chủ động là tác nhân gây ra Use Case, trong khi tác nhân thụ động không bao giờ gây ra Use Case mà chỉ tham gia vào một hoặc là nhiều Use Case.

5.3- Tìm tác nhân:

Khi nhận diện tác nhân, có nghĩa là chúng ta lọc ra các thực thể đáng quan tâm theo khía cạnh sử dụng và tương tác với hệ thống. Sau đó chúng ta có thể thử đặt mình vào vị trí của tác nhân để cố gắng nhận ra các yêu cầu và đòi hỏi của tác nhân đối với hệ thống và xác định tác nhân cần những Use Case nào. Có thể nhận diện ra các tác nhân qua việc trả lời một số các câu hỏi như sau:

- Ai sẽ sử dụng những chức năng chính của hệ thống (tác nhân chính)?
- Ai sẽ cần sự hỗ trợ của hệ thống để thực hiện những tác vụ hàng ngày của họ?
- Ai sẽ cần bảo trì, quản trị và đảm bảo cho hệ thống hoạt động (tác nhân phụ)?
- Hệ thống sẽ phải xử lý và làm việc với những trang thiết bị phần cứng nào?
- Hệ thống cần phải tương tác với các hệ thống khác nào? Nhóm các hệ thống này được chia ra làm hai nhóm, nhóm kích hoạt cho mối quan hệ với hệ thống, và nhóm mà hệ thống cần phải xây dựng của chúng ta sẽ thiết lập quan hệ. Khái niệm hệ thống bao gồm cả các hệ thống máy tính khác cũng như các ứng dụng khác trong chính chiếc máy tính mà hệ thống này sẽ hoạt động.
- Ai hay cái gì quan tâm đến kết quả (giá trị) mà hệ thống sẽ sản sinh ra?

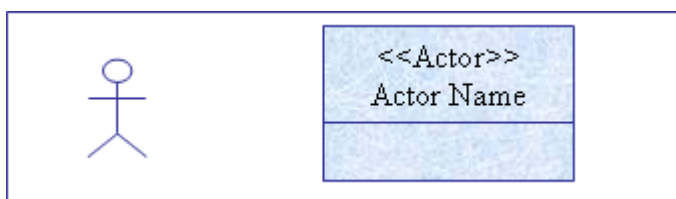
Khi đi tìm những người sử dụng hệ thống, đừng quan sát những người đang ngồi ở trước màn hình máy tính. Nên nhớ rằng, người sử dụng có thể là bất kỳ người nào hay bất kỳ vật nào tương tác hoặc trực tiếp hoặc gián tiếp với hệ thống và sử dụng các dịch vụ của hệ thống này để đạt đến một kết quả nào đó. Đừng quên rằng mô hình hóa Use Case được thực hiện để mô hình hóa một doanh nghiệp, vì thế tác nhân thường thường là khách hàng của doanh nghiệp đó. Từ đó suy ra họ không phải là người sử dụng theo nghĩa đơn giản và trực tiếp là người ngồi trước màn hình máy tính và thao tác với máy tính.

Để có thể nhận dạng được tốt nhiều tác nhân khác nhau, hãy tiến hành nghiên cứu những người sử dụng của hệ thống hiện thời (một hệ thống thủ công hoặc một hệ thống đang tồn tại), hỏi xem họ đóng những vai trò nào khi thực thi công việc hàng ngày của họ với hệ thống. Cũng người sử dụng đó có thể thực thi nhiều vai trò khác nhau tại nhiều thời điểm khác nhau, tùy thuộc vào việc chức năng nào trong hệ thống đang được sử dụng.

Xin nhắc lại, một tác nhân là một vai trò (một lớp), chứ không phải một thực thể riêng lẻ. Mặc dù vậy, khi cung cấp ví dụ là một vài các thực thể của một tác nhân, bạn có thể đảm bảo rằng tác nhân đó thật sự tồn tại. Một tác nhân phải có một sự liên kết (Association) nào đó với một hoặc là nhiều Use Case. Mặc dù có những tác nhân có thể không kích hoạt nên một Use Case nào, nhưng tác nhân đó sẽ giao tiếp ít nhất với một Use Case tại một thời điểm nào đó. Cần phải đặt tên cho tác nhân làm sao để tên phản ánh đúng vai trò của tác nhân đó trong hệ thống.

◆ 5.4- Biểu diễn tác nhân trong ngôn ngữ UML:

Tác nhân trong UML là một lớp với biệt ngữ "Actor" (Tác nhân) và tên của lớp này là tên của tác nhân (phản ánh vai trò của tác nhân). Một lớp tác nhân có thể vừa có thuộc tính (attribute) lẫn hành vi (method) cũng như một thuộc tính tài liệu (document) mô tả tác nhân đó. Một lớp tác nhân có một biểu tượng chuẩn hóa, biểu tượng "hình nhân":



Hình 4.2- biểu tượng tác nhân trong UML

◆ 5.5- Use Case:

Một Use Case là đại diện cho một chức năng nguyên vẹn mà một tác nhân nhận được. Một Use Case trong ngôn ngữ UML được định nghĩa là một tập hợp của các chuỗi hành động mà một hệ thống thực hiện để tạo ra một kết quả có thể quan sát được, tức là một giá trị đến với một tác nhân cụ thể. Những hành động này có thể bao gồm việc giao tiếp với một loạt các tác nhân cũng như thực hiện tính toán và công việc nội bộ bên trong hệ thống.

Các tính chất tiêu biểu của một Use Case là:

- Một Use Case bao giờ cũng được gây ra bởi một tác nhân, được thực hiện nhân danh một tác nhân nào đó. Tác nhân phải ra lệnh cho hệ thống để thực hiện Use Case đó, dù là trực tiếp hay gián tiếp. Hiếm khi có tác nhân không liên quan đến việc gây ra một Use Case nào đó.
- Một Use Case phải cung cấp một giá trị cho một tác nhân. Giá trị đó không phải bao giờ cũng cần thiết phải nổi trội ra ngoài, nhưng luôn phải được thấy rõ.

- Một Use Case là phải hoàn tất. Một trong những lỗi thường gặp là sẽ chia một Use Case thành các Use Case nhỏ hơn, và các Use Case này thực thi lẫn nhau giống như việc gọi hàm cho một ngôn ngữ lập trình. Một Use Case sẽ không được coi là hoàn tất chừng nào mà giá trị cuối cùng của nó chưa được sản sinh ra, thậm chí ngay cả khi đã xảy ra nhiều động tác giao tiếp (ví dụ như đối thoại với người sử dụng).

Use Case được nối với tác nhân qua liên kết (association). Đường liên kết chỉ ra những tác nhân nào giao tiếp với Use Case nào. Mỗi liên kết bình thường ra là một mối quan hệ 1-1 và không có hướng. Điều đó muốn nói lên rằng một thực thể của lớp tác nhân sẽ giao tiếp với một thực thể của một Use Case và cả hai có thể giao tiếp với nhau trong cả hai chiều. Một Use Case sẽ được đặt tên theo một thực thể mà Use Case sẽ thực hiện, ví dụ như ký hợp đồng bảo hiểm, cập nhật danh sách, v.v..., và thường là một cụm từ hơn là chỉ một từ riêng lẻ.

Một Use Case là một lớp, chứ không phải một thực thể. Nó mô tả trọn vẹn một chức năng, kể cả các giải pháp bổ sung và thay thế có thể có, các lỗi có thể xảy ra cũng như những ngoại lệ có thể xảy ra trong quá trình thực thi. Một kết quả của sự thực thể hóa một Use Case được gọi là một cảnh kịch (scenario) và nó đại diện cho một sự sử dụng cụ thể của hệ thống (một đường dẫn thực thi riêng biệt qua hệ thống). Ví dụ một cảnh kịch của Use Case "Ký hợp đồng bảo hiểm" có thể là "John liên hệ với hệ thống qua điện thoại rồi sau đó ký hợp đồng bảo hiểm ô tô cho chiếc xe Toyota Carolla mà anh ta vừa mua."

5.6- Tìm Use Case:

Quá trình tìm các Use Case bắt đầu với các tác nhân đã được xác định ở phần trước. Đối với mỗi tác nhân, hãy hỏi các câu hỏi sau:

- a. Tác nhân này cần những chức năng nào từ hệ thống? Hành động chính của tác nhân là gì ?.

Ví dụ cho một giao dịch rút tiền bên máy ATM trong một nhà băng lẻ, các hành động chính của khách hàng (tác nhân) có thể là:

- Đút thẻ vào máy ATM
- Nhập password
- Nhập loại chuyển dịch
- Nhập số tiền mặt muốn rút ra
- Yêu cầu về loại tiền

- Nhặt tiền ra từ máy
- Rút thẻ và tờ in kết quả giao dịch

b. Tác nhân có cần phải đọc, phải tạo, phải hủy bỏ, phải sửa chữa, hay là lưu trữ một loại thông tin nào đó trong hệ thống?

Ví dụ:

- Nhân viên nhà băng liệu có quyền truy xuất hay thay đổi mức tiền lãi?
- Khách hàng có thể thay đổi password của mình.

c. Tác nhân có cần phải báo cho hệ thống biết về những sự kiện nào đó? Những sự kiện như thế sẽ đại diện cho những chức năng nào?

Ví dụ:

- Khách hàng kết thúc tài khoản, nhân viên cung cấp những thông tin này cho hệ thống.
- Có một chương trình đầu tư mới, các chi tiết của chương trình này sẽ phải được nhân viên nhà băng nhập vào hệ thống.

d. Hệ thống có cần phải thông báo cho Actor về những thay đổi bất ngờ trong nội bộ hệ thống?

- Trong tài khoản còn quá ít tiền.
- Ba kỳ liên tiếp tiền lương chưa đổ về tài khoản.

e. Công việc hàng ngày của tác nhân có thể được đơn giản hóa hoặc hữu hiệu hóa qua các chức năng mới trong hệ thống (thường đây là những chức năng tiêu biểu chưa được tự động hóa trong hệ thống)?

f. Các câu hỏi khác:

- Use Case có thể được gây ra bởi các sự kiện nào khác?

Ví dụ:

Sự kiện thời gian: Cuối tháng, hết hạn đầu tư.

Sự kiện bình thường của hệ thống: Tự động chuyển tiền theo các lệnh xác định trước.

Các sự kiện bất bình thường: Hợp đồng đầu tư kết thúc trước thời hạn.

- Hệ thống cần những thông tin đầu vào/đầu ra nào? Những thông tin đầu vào/đầu ra đó từ đâu tới và sẽ đi đâu?
- Khó khăn và thiếu hụt chính trong hệ thống hiện thời nằm ở đâu (thủ công /tự động hóa)?

Đối với nhóm câu hỏi cuối không có nghĩa là Use Case ở đây không có tác nhân, mà tác nhân sẽ được nhận ra chỉ khi chúng ta nhận diện ra các Use Case này và sau đó xác định tác nhân dựa trên cơ sở là Use Case. Xin nhắc lại, một Use Case bao giờ cũng phải được liên kết với ít nhất một tác nhân.

5.7- Ví dụ tìm Use Case:

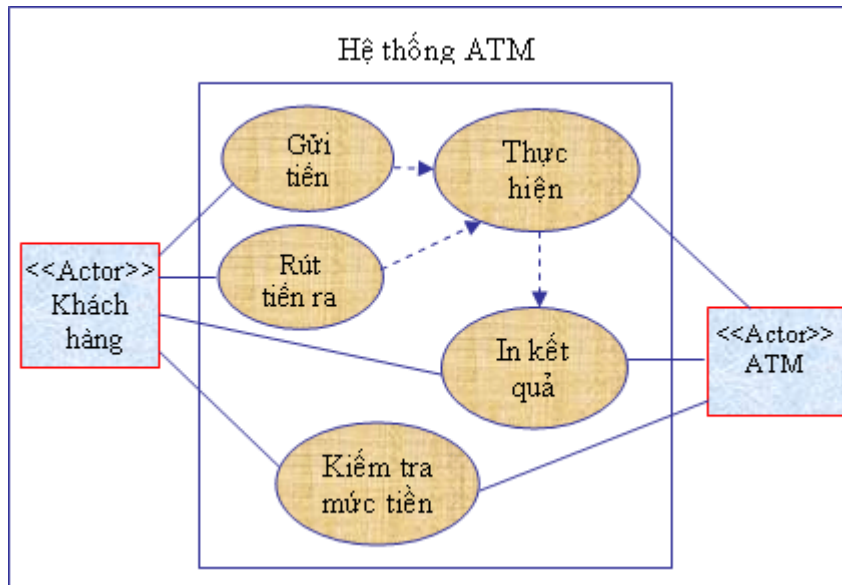
Nhà băng ABC đưa ra các yêu cầu sau:

- Một khách hàng có thể muốn gửi tiền vào, rút tiền ra hoặc đơn giản kiểm tra lại số tiền trong tài khoản của anh ta qua máy tự động rút tiền (ATM). Khi đưa tiền vào hoặc rút tiền ra, cần phải ghi ra giấy kết quả những chuyển dịch đã thực hiện và trao tờ giấy này cho khách hàng.

Quan sát các chức năng căn bản và các thành phần tham gia, ta thấy có hai tác nhân dễ nhận ra nhất là khách hàng và nhân viên thu ngân.

Qua đó, có thể nhận dạng các Use Case sau:

- Gửi tiền vào.
- Rút tiền ra.
- Kiểm tra mức tiền trong tài khoản
- Thực hiện các chuyển dịch nội bộ hệ thống
- In kết quả các chuyển dịch đã thực hiện.



Hình 4.3 – Các Use case trong hệ thống ATM

Use Case gửi tiền vào và rút tiền ra phụ thuộc vào Use Case thực hiện các chuyển dịch trong nội bộ hệ thống, việc thực hiện này về phần nó lại phụ thuộc vào chức năng in ra các công việc đã được thực hiện. Kiểm tra mức tiền trong tài khoản là một Use Case độc lập, không phụ thuộc vào các Use Case khác.

6- CÁC BIẾN THỂ (VARIATIONS) TRONG MỘT USE CASE

Mỗi Use Case sẽ có một dòng hành động chính (Basic Course). Đó là tiến trình bình thường hay tiến trình mong đợi đối với Use Case này.

Ngoài ra, có thể còn có một hay nhiều dòng hành động thay thế (Alternative) khác. Chúng có thể được chia làm hai nhóm chính:

- Thay thế bình thường (Normal Alternative)
- Điều kiện gây lỗi (Error Condidtions)

Những gì mang tính bình thường hơn trong Use Case được gọi là Thay thế bình thường.

Có thể miêu tả các dòng hành động thay thế bằng từ ngữ (xem phần tài liệu Use Case).

Ví dụ một khách hàng có thể chọn các loại giao dịch sau của ATM:

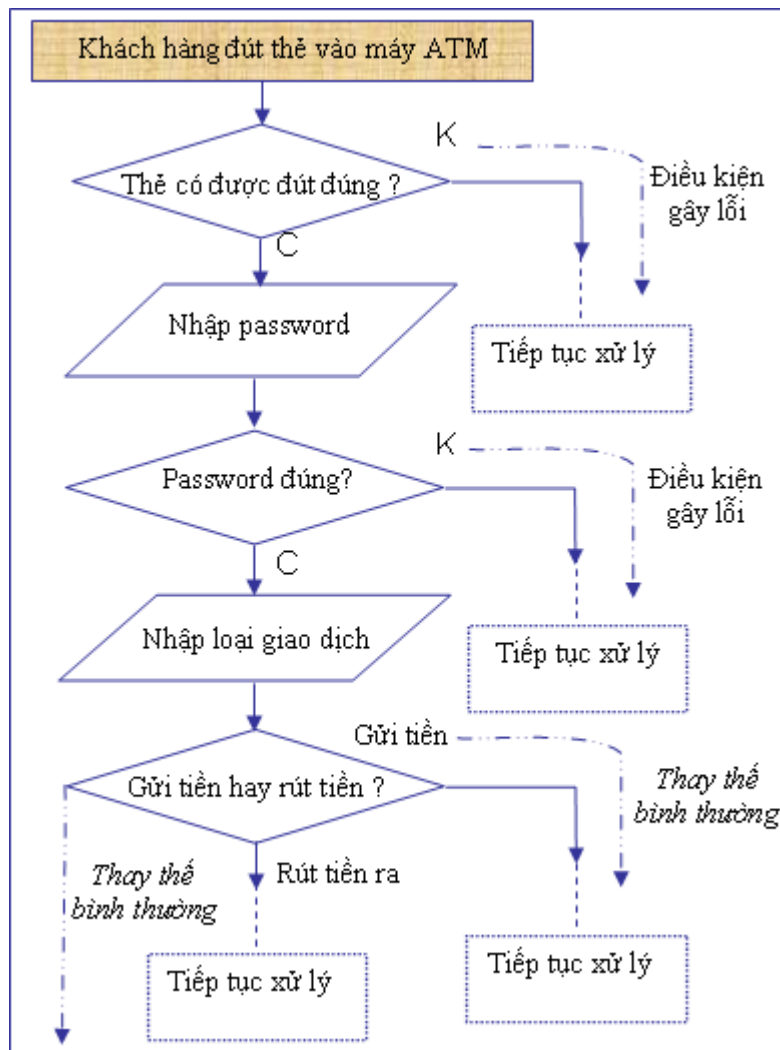
- Gửi tiền vào
- Rút tiền ra
- Kiểm tra mức tiền trong tài khoản

Đây là những ví dụ cho các dòng hành động thay thế bình thường.

Điều kiện gây lỗi đại diện cho những bước tiến hành bất bình thường trong một Use Case. Cần phải tính trước đến những điều kiện gây lỗi đó, ví dụ:

- Mức tiền trong tài khoản không đủ để tiến hành giao dịch
- Password không đúng
- ATM bị nghiền thẻ

Hình sau nêu bật dòng hành động chính và những dòng hành động thay thế cũng như sự khác biệt của chúng đối với tiến trình mong đợi của Use Case.



Hình 4.4 – Các tiến trình trong hệ thống ATM

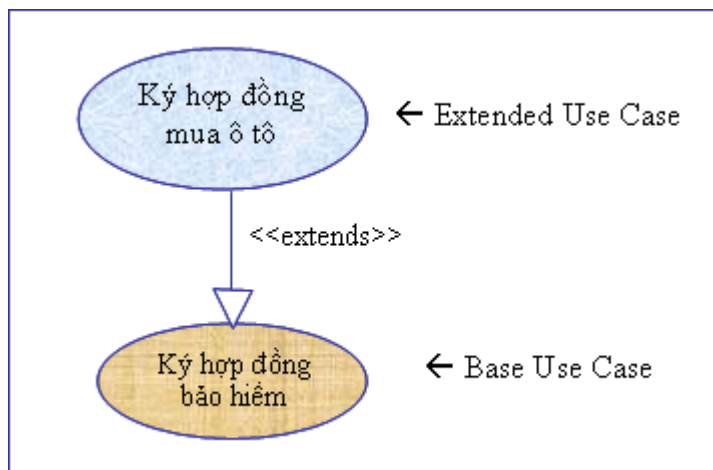
7- QUAN HỆ GIỮA CÁC USE CASE

Có ba loại quan hệ Use Case: Quan hệ mở rộng, quan hệ sử dụng và quan hệ tạo nhóm. Quan hệ mở rộng và quan hệ sử dụng là hai dạng khác nhau của tính thừa kế. Quan hệ tạo nhóm là một phương cách để đặt nhiều Use Case chung với nhau vào trong một gói.

7.1- Quan hệ mở rộng

Nhiều khi trong quá trình phát triển Use Case, người ta thấy một số Use Case đã tồn tại cung cấp một phần những chức năng cần thiết cho một Use Case mới. Trong một trường hợp như vậy, có thể định nghĩa một Use Case mới là Use Case cũ cộng thêm một phần mới. Một Use Case như vậy được gọi là một Use Case mở rộng (Extended Use Case). Trong quan hệ mở rộng, Use Case gốc (Base Use Case) được dùng để mở rộng phải là một Use Case hoàn thiện. Use Case mở rộng không nhất thiết phải sử dụng toàn bộ hành vi của Use Case gốc.

Biểu đồ sau chỉ ra Use Case "Ký hợp đồng mua ô tô" là Use Case mở rộng của "Ký hợp đồng bảo hiểm".



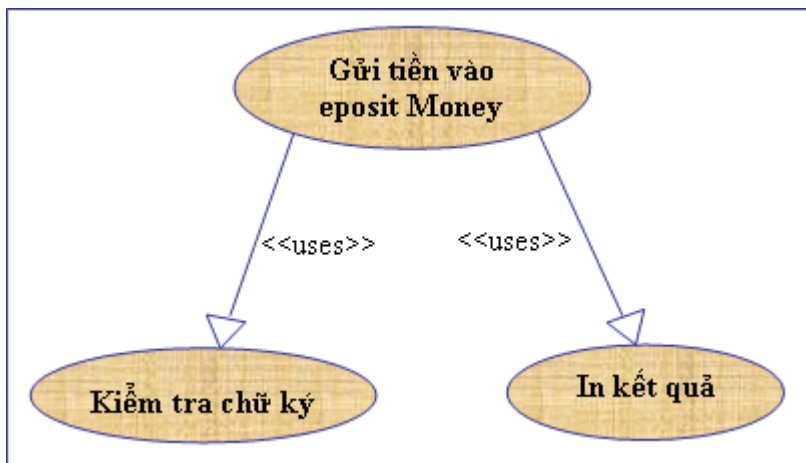
Hình 4.5 - Quan hệ mở rộng giữa các Use Case

Quan hệ mở rộng giữa các Use Case được biểu thị bằng đoạn thẳng với hình tam giác rỗng trỏ về phía Use Case được dùng để mở rộng, đi kèm với stereotype <<extends>>.

7.2- Quan hệ sử dụng

Khi một nhóm các Use Case cùng chung một hành vi nào đó thì hành vi này có thể được tách riêng ra thành một Use Case riêng biệt và nó có thể được sử dụng bởi các Use Case kia, một mối quan hệ như vậy được gọi là quan hệ sử dụng.

Trong quan hệ sử dụng, phải sử dụng toàn bộ Use Case khái quát hóa, nói một cách khác, ta có một Use Case này sử dụng toàn bộ một Use Case khác. Các hành động trong Use Case khái quát hóa không cần phải được sử dụng trong cùng một tiến trình. Chúng có thể được trộn lẫn với các hành động xảy ra trong Use Case chuyên biệt hóa.



Hình 4.6 - Quan hệ sử dụng giữa các Use Case

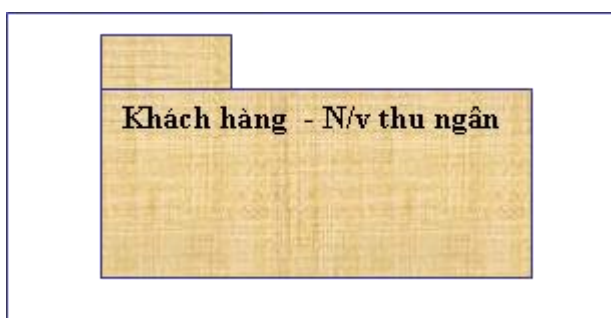
Quan hệ sử dụng giữa các Use Case được biểu thị bằng đoạn thẳng với hình tam giác rỗng trở về phía Use Case được sử dụng, đi kèm với stereotype <<uses>>.

◆ 7.3- Quan hệ chung nhóm

Khi một số các Use Case cùng xử lý các chức năng tương tự hoặc có thể liên quan đến nhau theo một phương thức nào đó, người ta thường nhóm chúng lại với nhau.

Nhóm các Use Case được thực hiện bằng khái niệm "Gói" (Package) của UML. Gói không cung cấp giá trị gia tăng cho thiết kế.

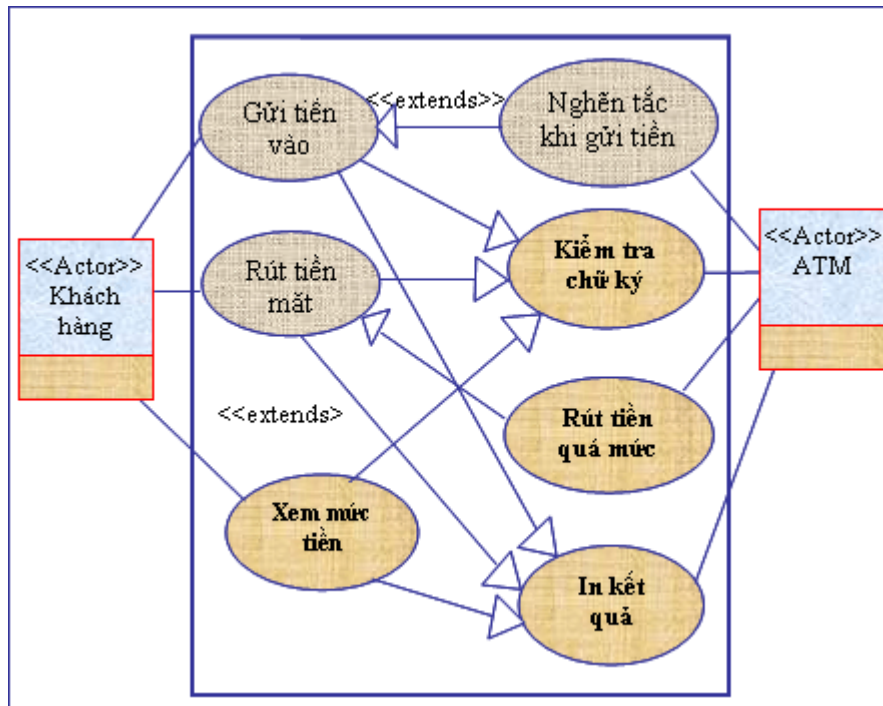
Ví dụ: tất cả các Use Case có liên quan đến sự tương tác giữa khách hàng và nhân viên thu ngân sẽ được nhóm thành "Package Khách hàng- N/v thu ngân"



Hình 4.7 – Package của UML

Tóm tắt về Use Case với máy ATM trong ngân hàng lẻ:

Cho tới nay chúng ta đã xác định được một vài Use Case, phân tích dòng hành động chính cũng như các dòng hành động thay thế, cũng như rút ra các mối quan hệ giữa chúng. Biểu đồ sau tổng hợp những thông tin đã thu thập được về nhóm các Use Case căn bản của một hệ thống ATM.



Hình 4.8 - Biểu đồ một số Use Case trong hệ thống ATM

8- MIÊU TẢ USE CASE

Như đã trình bày, lời miêu tả một Use Case thường được thực hiện trong văn bản. Đây là lời đặc tả đơn giản và nhất quán về việc các tác nhân và các Use Case (hệ thống) tương tác với nhau ra sao. Nó tập trung vào ứng xử đối ngoại của hệ thống và không đề cập tới việc thực hiện nội bộ bên trong hệ thống. Ngôn ngữ và các thuật ngữ được sử dụng trong lời miêu tả chính là ngôn ngữ và các thuật ngữ được sử dụng bởi khách hàng/người dùng.

Văn bản miêu tả cần phải bao gồm những điểm sau:

- Mục đích của Use Case: Mục đích chung cuộc của Use Case là gì? Cái gì cần phải được đạt tới? Use Case nói chung đều mang tính hướng mục đích và mục đích của mỗi Use Case cần phải rõ ràng.
- Use Case được khởi chạy như thế nào: Tác nhân nào gây ra sự thực hiện Use Case này? Trong hoàn cảnh nào?
- Chuỗi các thông điệp giữa tác nhân và Use Case: Use Case và các tác nhân trao đổi thông điệp hay sự kiện nào để thông báo lẫn cho nhau, cập nhật hoặc nhận thông tin và giúp đỡ nhau quyết định? Yếu tố nào sẽ miêu tả dòng chảy chính của các thông điệp giữa hệ thống và tác nhân, và những thực thể nào trong hệ thống được sử dụng hoặc là bị thay đổi?

- Dòng chảy thay thế trong một Use Case: Một Use Case có thể có những dòng thực thi thay thế tùy thuộc vào điều kiện. Hãy nhắc đến các yếu tố này, nhưng chú ý đừng miêu tả chúng quá chi tiết đến mức độ chúng có thể "che khuất" dòng chảy chính của các hoạt động trong trường hợp căn bản. Những động tác xử lý lỗi đặc biệt sẽ được miêu tả thành các Use Case khác.
- Use Case sẽ kết thúc với một giá trị đối với tác nhân như thế nào: Hãy miêu tả khi nào Use Case được coi là đã kết thúc, và loại giá trị mà nó cung cấp đến tác nhân.

Hãy nhớ rằng lời miêu tả này sẽ xác định những gì được thực thi có liên quan đến tác nhân bên ngoài, chứ không phải những sự việc được thực hiện bên trong hệ thống. Văn bản phải rõ ràng, nhất quán, khiến cho khách hàng có thể dễ dàng hiểu và thẩm tra chúng (để rồi đồng ý rằng nó đại diện cho những gì mà anh/cô ta muốn từ phía hệ thống). Tránh dùng những câu văn phức tạp, khó diễn giải và dễ hiểu lầm.

Một Use Case cũng có thể được miêu tả qua một biểu đồ hoạt động. Biểu đồ hoạt động này chỉ ra chuỗi các hành động, thứ tự của chúng, các quyết định chọn lựa để xác định xem hành động nào sau đó sẽ được thực hiện.

Để bổ sung cho lời miêu tả một Use Case, người ta thường đưa ra một loạt các cảnh kịch cụ thể để minh họa điều gì sẽ xảy ra một khi Use Case này được thực thể hóa. Lời miêu tả cảnh kịch minh họa một trường hợp đặc biệt, khi cả tác nhân lẫn Use Case đều được coi là một thực thể cụ thể. Khách hàng có thể dễ dàng hiểu hơn toàn bộ một Use Case phức tạp nếu có những cảnh kịch được miêu tả thực tiễn hơn, minh họa lại lỗi ứng xử và phương thức hoạt động của hệ thống. Nhưng xin nhớ rằng, một lời miêu tả cảnh kịch chỉ là một sự bổ sung chứ không phải là ứng cử viên để thay thế cho lời miêu tả Use Case.

Sau khi các Use Case đã được miêu tả, một hoạt động và một công việc đặc biệt cần phải thực hiện là thẩm tra xem các mối quan hệ (đã đề cập tới trong phần 2.7) có được nhận diện không. Trước khi tất cả các Use Case được miêu tả, nhà phát triển chưa thể có được những kiến thức hoàn tất và tổng thể để xác định các mối quan hệ thích hợp, thử nghiệm làm theo phương thức đó có thể sẽ dẫn đến một tình huống nguy hiểm. Trong thời gian thực hiện công việc này, hãy trả lời các câu hỏi sau:

- Tất cả các tác nhân liên quan đến một Use Case có mối liên kết giao tiếp với Use Case đó không?

- Có tồn tại những sự tương tự giữa một loạt các tác nhân minh họa một vai trò chung và nhóm này liệu có thể được miêu tả là một lớp tác nhân căn bản (base class)?
- Có tồn tại những sự tương tự giữa một loạt các Use Case, minh họa một dòng chảy hành động chung? Nếu có, liệu điều này có thể được miêu tả là một mối quan hệ sử dụng đến với một Use Case khác?
- Có tồn tại những trường hợp đặc biệt của một Use Case có thể được miêu tả là một mối quan hệ mở rộng?
- Có tồn tại một tác nhân nào hay một Use Case nào không có mối liên kết giao tiếp? Nếu có, chắc chắn ở đây đã có chuyện nhầm lẫn, sai trái: Tại sao lại xuất hiện tác nhân này?
- Có lời yêu cầu nào về chức năng đã được xác định, nhưng lại không được bất kỳ một Use Case nào xử lý? Nếu thế, hãy tạo một Use Case cho yêu cầu đó.

Văn bản miêu tả một Use Case đơn giản:

Ví dụ Use Case "Cung Cấp Thông Tin Về Một Tài Khoản Tại Nhà Băng ABC": Sau khi phân tích hệ thống, ta nhận thấy cần có một Use Case để in lên màn hình của nhân viên nhà băng tất cả những chi tiết về một tài khoản của một khách hàng.

Đặc tả Use Case:

Chi tiết tài khoản: // tên Use Case

Số Use Case: UCSEC35

Miêu tả ngắn: // miêu tả ngắn gọn Use Case

Use Case "chi tiết tài khoản" cho phép nhân viên nhà băng xem các chi tiết của một tài khoản mà anh ta định tìm hiểu.

Dòng chảy các sự kiện: // dòng logic chung

Nhân viên chọn Chi Tiết Tài Khoản trên menu. Một con đường khác để chỉ ra các thông tin chi tiết của tài khoản là gọi từ Màn Hình Tóm Tắt Thông Tin Về Tài Khoản (xem Use Case số UCSEC99).

Dòng hành động chính: // dòng logic chi tiết.

Mỗi khách hàng sẽ có một số định danh gọi là CustomerId. Một khách hàng có thể có nhiều tài khoản. Sau khi nhân viên nhập CustomerId vào hệ thống, màn hình phải in ra tất cả những tài khoản thuộc về khách hàng này và thuộc về nhà

bảng ABC, rải rác tại tất cả các chi nhánh. Khi chọn tiếp loại tài khoản và số tài khoản, các chi tiết của tài khoản mong muốn sẽ được in ra.

Loại tài khoản tiết kiệm: Nếu loại tài khoản được chọn là tài khoản tiết kiệm, thì theo Use Case số UCSEC45, các chi tiết sau đây sẽ được in ra:

- Mức tiền hiện có
- Các tờ séc chưa thanh toán
- Lượng tiền tín dụng được phép
- Lượng tiền lãi cho tới ngày hôm nay
- Lượng tiền tối thiểu cần phải có trong tài khoản

Loại tài khoản đầu tư: Nếu loại tài khoản được chọn là loại đầu tư, thì theo Use Case số UCSEC46, các chi tiết sau đây sẽ được in ra.

- Hạn đầu tư
- Số tiền đầu tư
- Ngày đầu tư
- Lượng tiền cuối hạn
- Ngày cuối hạn
- Tỷ lệ lời

Dòng hành động thay thế: // chuỗi logic thay thế

Không tìm thấy chi tiết: Khi chọn một số tài khoản không thích hợp (không có tài khoản tương ứng) dù vì lý do chức năng hay kỹ thuật, theo Use Case số UCSEC12, hệ thống sẽ đưa ra một màn hình báo lỗi.

Điều kiện thoát: // Use Case kết thúc như thế nào?

Nút Thoát: Khi chọn nút thoát, người sử dụng sẽ quay trở lại màn hình chính.

Nút Xem Thêm: Khi chọn nút này, người sử dụng sẽ được yêu cầu chọn loại tài khoản từ một danh sách đổ xuống.

Nút Xem Giao Dịch: Khi chọn nút này, người sử dụng sẽ được chuyển sang màn hình "Giao dịch" và theo Use Case số UCSEC91, màn hình sẽ chỉ ra những giao dịch đã xảy ra đối với tài khoản, bên cạnh những chi tiết chính của tài khoản.

Nút Yêu Cầu In Kết Quả: Khi chọn phần thực đơn này, kết quả giao dịch theo Use Case số UCSEC70 sẽ được in ra ở một máy in địa phương nối trực tiếp với máy tính của nhân viên.

Các yêu cầu đặc biệt: // các yêu cầu đặc biệt

Theo Use Case số UCSEC110, hệ thống có khả năng in lên màn hình bằng những ngôn ngữ khác. Chức năng này sẽ được kích hoạt khi người sử dụng chọn mục Ngoại Ngữ trên menu.

Điều kiện trước đó: // điều xảy ra trước khi Use Case được thực hiện

Bảo an: Người sử dụng (nhân viên tiếp khách) được cung cấp một số định danh riêng biệt để truy nhập vào hệ thống.

Dịch chuyển: Người sử dụng chỉ đến được màn hình Chi Tiết Tài Khoản sau khi đã truy nhập thành công và Identify thành công.

Điều kiện sau đó: // điều gì xảy ra sau khi Use Case được thực hiện?

Hệ thống sẽ không lưu trữ lại bất kỳ một thông tin nào liên quan tới khách hàng lên đĩa cứng cục bộ.

9- THỬ USE CASE

Một trong các mục đích chính của Use Case là thử nghiệm (testing). Có hai loại thử nghiệm khác nhau được thực hiện ở đây: **kiểm tra** (verification) và **phê duyệt xác nhận** (validation). Kiểm tra đảm bảo là hệ thống đã được phát triển đúng đắn và phù hợp với các đặc tả đã được tạo ra. Phê duyệt xác nhận đảm bảo rằng hệ thống sẽ được phát triển chính là thứ mà khách hàng hoặc người sử dụng cuối thật sự cần đến.

Công việc **phê duyệt xác nhận** được thực hiện kể trước giai đoạn phát triển. Ngay khi một mô hình Use Case được hoàn tất (hay thậm chí có thể đang trong giai đoạn phát triển), mô hình này phải được trình bày và thảo luận với khách hàng cũng như người sử dụng. Họ cần phải xác nhận rằng mô hình này là đúng đắn, hoàn tất và thỏa mãn sự mong đợi của họ đối với hệ thống; đặc biệt là phương cách mà hệ thống cung cấp chức năng cho họ. Để làm điều đó, nhà phát triển phải đảm bảo rằng khách hàng thật sự hiểu được mô hình và ý nghĩa của chúng, để tránh trường hợp tạo ra những thứ không thể chấp nhận nổi. Trong giai đoạn này, rõ ràng là các câu hỏi và các ý tưởng sẽ xuất hiện và chúng cần phải được bổ sung thêm vào mô hình Use Case trước khi đến giai đoạn phê duyệt chung cuộc. Giai đoạn xác nhận cũng có thể được thực hiện trong thời kỳ thử nghiệm hệ thống, nhưng điểm yếu của phương thức làm này là nếu hệ thống không thỏa mãn những yêu cầu cụ thể của người sử dụng thì toàn bộ dự án rất có thể sẽ phải làm lại từ đầu.

Kiểm tra hệ thống là để đảm bảo nó hoạt động đúng như đặc tả. Điều này không thể được thực hiện trước khi đã có những thành phần của hệ thống được tạo ra. Chỉ sau đó người ta mới có thể thử xem hệ thống có hoạt động đúng như đặc tả mà người sử dụng đã đưa ra, rằng các Use Case thực hiện đúng theo như những lời đã miêu tả trong mô hình, rằng chúng hoạt động theo đúng phương thức đã được miêu tả trong văn bản miêu tả Use Case.

Đi bộ dọc Use Case.

Một trong những kỹ thuật hữu dụng được dùng trong cả giai đoạn định nghĩa lẫn thử nghiệm Use Case gọi là "Đi Bộ Dọc Use Case". Theo kỹ thuật này, nhiều người khác nhau trong nhóm làm mô hình sẽ đóng vai các tác nhân cũng như hệ thống trong một Use Case cụ thể. Người đảm nhận vai tác nhân sẽ bắt đầu bằng việc nói ra tác nhân làm gì với hệ thống. Kết quả của công việc này là hệ thống sẽ khởi chạy một Use Case cụ thể được bắt đầu từ hành động trên. Người đóng vai hệ thống sau đó sẽ nói anh ta làm gì khi Use Case được thực hiện. Nhà phát triển đứng ngoài trò chơi diễn kịch sẽ ghi chép và tìm cách phát hiện ra các điểm yếu trong các Use Case được miêu tả bằng các diễn viên. Trong trường hợp đặc thù, bạn sẽ tìm thấy rằng có một vài chuỗi hành động bổ sung không được miêu tả cũng như một vài hành động không được miêu tả với đầy đủ chi tiết.

Các "diễn viên" càng hiểu thấu đáo khía cạnh sử dụng của hệ thống bao nhiêu thì công việc thử Use Case sẽ càng hiệu quả bấy nhiêu. Việc thay đổi các diễn viên để đóng những vai trò khác nhau sẽ dẫn tới những thay đổi trong miêu tả và hướng nhìn, cung cấp dữ liệu đầu vào cho các nhà tạo mô hình để họ biết được làm cách nào có thể đưa ra những lời miêu tả Use Case rõ ràng hơn, minh bạch hơn, và chỉ ra những điểm còn thiếu. Một khi vai trò của tất cả các tác nhân đã được diễn và thực thi, và tất cả các Use Case đã được thực thi theo kiểu này, đó là thời điểm mà người ta nói một quá trình thử nghiệm của mô hình Use Case đã hoàn tất.

10- THỰC HIỆN CÁC USE CASE

Use Case là những lời miêu tả độc lập với sự thực thi các chức năng của hệ thống. Điều đó có nghĩa là Use Case sẽ được thực hiện (thực thể hóa) trong hệ thống, vậy nên trách nhiệm để thực thi các hành động được miêu tả trong tài liệu Use Case đều được phân bổ về cho các đối tượng cộng tác thực thi chức năng đó.

Các nguyên tắc của UML cho việc thực hiện các Use Case là: Một Use Case sẽ được thực hiện trong một sự cộng tác (collaboration): Một sự cộng tác chỉ ra một giải pháp (phụ thuộc vào sự thực thi nội bộ) của một Use Case sử dụng các khái niệm lớp/đối tượng và mối quan hệ giữa chúng đối với nhau (gọi là *ngữ cảnh* – *context* của sự cộng tác) cũng như sự tương tác giữa chúng để đạt tới chức năng

mong muốn (gọi là chuỗi tương tác của sự cộng tác). Kí hiệu cho sự cộng tác là một hình ellipse có chứa tên của sự cộng tác đó.

Một sự cộng tác được trình bày trong UML qua một loạt các biểu đồ chỉ ra cả ngữ cảnh lẫn chuỗi tương tác giữa các thành phần tham gia: thành phần tham gia trong một sự cộng tác là một loạt các lớp (và trong một thực thể cộng tác: các đối tượng). Các biểu đồ sử dụng ở đây là biểu đồ cộng tác, biểu đồ chuỗi và biểu đồ hoạt động. Cần phải sử dụng loại biểu đồ nào để tạo ra một bức tranh bao quát về sự cộng tác là quyết định tùy thuộc vào từng trường hợp cụ thể. Trong một vài trường hợp, chỉ một biểu đồ cộng tác đã có thể là đủ; nhưng trong các trường hợp khác, người ta nhất thiết cần tới sự kết hợp của nhiều loại biểu đồ khác nhau.

Một cảnh kịch (Scenario) là một thực thể (instance) của một Use Case hay là một sự cộng tác: một cảnh kịch là một chuỗi thực thi cụ thể (một dòng chảy cụ thể của các sự kiện) trình bày một sự thực thể hóa của một Use Case (tức là một lần sử dụng hệ thống). Khi một cảnh kịch được quan sát trong tư cách một Use Case, người ta chỉ miêu tả những ứng xử bên ngoài hướng về phía tác nhân. Khi quan sát một cảnh kịch trong tư cách là một thực thể của sự cộng tác, người ta sẽ miêu tả cả sự thực thi nội tại (các dòng lệnh code) của các lớp tham gia ở đây, thuật toán cũng như thủ tục của chúng cùng sự giao tiếp giữa chúng với nhau.

Tác vụ thực hiện một Use Case là chuyển các bước và hành động khác nhau trong lời miêu tả Use Case thành lớp (Class), thủ tục trong những lớp này cũng như quan hệ giữa chúng với nhau. Nó được miêu tả là động tác phân bổ trách nhiệm của mỗi bước đi trong Use Case vào các lớp tham gia sự cộng tác thực hiện Use Case đó. Tại giai đoạn này, người ta phải tìm ra một giải pháp cung cấp những hành vi hướng ngoại đã được xác định của Use Case; nó được miêu tả trong những thuật ngữ của một sự cộng tác nội bộ trong hệ thống.

Mỗi bước hành động trong Use Case sẽ được chuyển thành thủ tục (operation) trong các lớp tham gia. Một bước trong Use Case sẽ được chuyển thành một loạt các thủ tục tại nhiều lớp; rất hiếm khi xảy ra ánh xạ 1-1 giữa các hành động trong Use Case và các thủ tục được thực thi trong tương tác giữa các đối tượng của các lớp tham gia. Cũng xin nhớ rằng một lớp có thể tham gia nhiều Use Case khác nhau và trách nhiệm cao nhất của lớp nằm chính trong việc kết tập tất cả các vai trò mà lớp này đảm nhận trong các Use Case khác nhau.

Mỗi quan hệ giữa một Use Case và sự thực thi nó theo khái niệm cộng tác được chỉ ra hoặc qua một mối quan hệ nâng cao (refinement relationship) – biểu thị bằng đoạn thẳng chấm chấm với mũi tên - - - -> hay một *hyperlink* ngầm trong một công cụ nào đó. Một hyperlink trong một công cụ sẽ tạo điều kiện chuyển từ

việc quan sát một Use Case trong một biểu đồ Use Case sang ngay sự cộng tác thực thi Use Case đó. Các hyperlink cũng được sử dụng để chuyển từ Use Case này sang một cảnh kịch (thường là một mô hình động – biểu đồ hoạt động, biểu đồ chuỗi hay biểu đồ cộng tác) miêu tả một sự thực hiện cụ thể nào đó của Use Case.

Phân bổ trách nhiệm cho các lớp một cách thành công là một tác vụ đòi hỏi kinh nghiệm. Cũng giống như mọi công đoạn hướng đối tượng khác, công việc này mang tính vòng lặp (iterative). Nhà phát triển thử nghiệm với nhiều sự phân bổ trách nhiệm khác nhau và dần dần nâng cấp chúng trong giải pháp của mình cho tới khi tạo ra được một mô hình thực hiện chức năng đó, đồng thời lại đủ mức độ năng động để cho phép tiến hành các sự thay đổi trong tương lai.

Jacobson sử dụng phương pháp định nghĩa ba loại đối tượng căn bản (có nghĩa là ba loại lớp): các đối tượng biên (boundary objects), đối tượng chỉ huy (control objects) và đối tượng thực thể (entity objects). Đối với mỗi Use Case, các loại đối tượng này được sử dụng để miêu tả một sự cộng tác thực thi Use Case. Trách nhiệm của các loại đối tượng kể trên như sau:

🔗➡️**Đối tượng thực thể:** loại đối tượng này đại diện cho các thực thể của bài toán nằm trong phạm vi mà hệ thống xử lý. Thường chúng mang tính thụ động, theo khái niệm là chúng không tự gây nên các tương tác đối với chúng. Trong một hệ thống thông tin, các đối tượng thực thể thường mang tính trường tồn (persistent) và được lưu trữ trong một hệ ngân hàng dữ liệu. Các đối tượng thực thể thường tham gia vào nhiều Use Case khác nhau.

🔗➡️**Đối tượng biên:** loại đối tượng này nằm gần đường ranh giới của hệ thống (mặc dù vẫn nằm bên trong hệ thống). Chúng tương tác với các tác nhân nằm bên ngoài hệ thống và nhận thông điệp cũng như gửi thông điệp đến các loại đối tượng khác nằm bên trong hệ thống.

🔗➡️**Đối tượng chỉ huy:** loại đối tượng này chỉ huy sự tương tác giữa các nhóm đối tượng. Một đối tượng như thế có thể đóng vai trò "bộ phận điều khiển" cho toàn bộ một Use Case hoàn tất, hay nó có thể thực thi một chuỗi hành động chung của nhiều Use Case. Thường thì một đối tượng như vậy chỉ tồn tại trong quá trình thực thi Use Case.

Ba loại đối tượng này có ba kí hiệu khác nhau và có thể được sử dụng khi vẽ các loại biểu đồ miêu tả cộng tác hoặc biểu đồ lớp. Sau khi đã định nghĩa nhiều loại đối tượng khác nhau và xác nhận các cộng tác, người ta có thể dễ dàng đi tìm sự tương tự giữa chúng để một số lớp có thể được sử dụng trong một loạt các Use

Case khác nhau. Sử dụng các Use Case theo phương thức này ta có thể tạo nên nền tảng cho việc phân tích và thiết kế hệ thống; qui trình phát triển được Ivar Jacobson gọi là "Qui Trình Phát Triển Theo Use Case" (Use case – driven).

Nhìn chung có nhiều phương pháp khác nhau để phân bổ trách nhiệm từ Use Case về cho các lớp. Có phương pháp đề nghị đầu tiên phải tiến hành phân tích phạm vi bài toán, chỉ ra tất cả các lớp thực thể (thuộc phạm vi bài toán) với mối quan hệ của chúng với nhau. Sau đó nhà phát triển sẽ phân tích từng Use Case và phân bổ trách nhiệm cho các lớp trong mô hình phân tích (analysis model), nhiều khi sẽ thay đổi chúng hoặc bổ sung thêm các lớp mới. Một phương pháp khác lại đề nghị là nên lấy các Use Case làm nền tảng để tìm các lớp, làm sao trong quá trình phân bổ trách nhiệm thì mô hình phân tích của phạm vi bài toán sẽ từng bước từng bước được thiết lập.

Một điểm quan trọng cần phải nhắc lại là công việc ở đây mang tính vòng lặp. Khi phân bổ trách nhiệm cho các lớp, ta có thể phát hiện ra sự thiếu đồng bộ hoặc lỗi trong các biểu đồ lớp và qua đó, dẫn đến việc sửa chữa trong biểu đồ lớp. Những lớp mới sẽ được nhận dạng và tìm ra nhằm mục đích hỗ trợ cho các Use Case. Trong một số trường hợp, thậm chí có thể xảy ra chuyện phải thay đổi hoặc sửa chữa cả biểu đồ Use Case vì khi hiểu hệ thống một cách sâu sắc hơn, nhà phát triển sẽ nhận ra rằng có một Use Case nào đó đã không được miêu tả chính xác và đúng đắn. Các Use Case giúp chúng ta tập trung vào khía cạnh chức năng của hệ thống, làm sao phải đảm bảo cho nó được miêu tả chính xác và được xây dựng chính xác trong hệ thống. Một trong những vấn đề xảy ra với nhiều phương pháp hướng đối tượng mà không sử dụng đến khái niệm Use Case là chúng tập trung quá nhiều vào cấu trúc tĩnh của các lớp và các đối tượng (nhiều khi người ta gọi là phương pháp mô hình hóa khái niệm – conceptual modeling) nhưng lại bỏ qua các khía cạnh chức năng và khía cạnh động của hệ thống.

11- TÓM TẮT VỀ USE CASE

Mô hình Use Case là một kỹ thuật được sử dụng để miêu tả những yêu cầu mang tính chức năng của một hệ thống. Use Case được miêu tả qua các khái niệm tác nhân bên ngoài, Use Case và hệ thống. Tác nhân tượng trưng cho một vai trò và một thực thể bên ngoài ví dụ như một người dùng, một bộ phận phần cứng hoặc một hệ thống khác tương tác với hệ thống. Tác nhân gây ra và giao tiếp với các Use Case, trong khi một Use Case là một tập hợp của các chuỗi hành động được thực hiện trong hệ thống. Một Use Case phải cung cấp một giá trị cần hướng tới nào đó cho tác nhân, và bình thường nó được miêu tả bằng văn bản. Tác nhân và Use Case là các lớp. Một tác nhân được liên kết với một hoặc nhiều Use Case qua mối liên kết (Association) và cả tác nhân lẫn Use Case đều có thể có mối quan hệ khái quát hóa, mỗi quan hệ này miêu tả những ứng xử chung trong các lớp cha,

sẽ được thừa kế bởi một hoặc nhiều lớp con. Một mô hình Use Case được miêu tả bằng một hay nhiều biểu đồ trường hợp thuộc ngôn ngữ UML.

Use Case được thực hiện qua các sự cộng tác. Một sự cộng tác là một lời miêu tả một ngữ cảnh, chỉ ra các lớp/ đối tượng và mối quan hệ của chúng và một tương tác chỉ ra các lớp/đối tượng đó tương tác với nhau ra sao để thực hiện một chức năng cụ thể. Một sự cộng tác được miêu tả bằng biểu đồ hoạt động, biểu đồ cộng tác và biểu đồ chuỗi. Khi một Use Case được thực hiện, trách nhiệm cho mỗi bước hành động trong Use Case cần phải được phân bổ cho các lớp tham gia sự cộng tác đó, thường là qua việc xác định các thủ tục của các lớp này, đi song song với phương thức mà chúng tương tác với nhau. Một cảnh kịch là một thực thể của một Use Case, hay một sự cộng tác, chỉ ra một chuỗi thực thi cụ thể. Vì thế, một cảnh kịch là một sự minh họa hay là một ví dụ của một Use Case hay là một sự cộng tác. Khi cảnh kịch được chỉ ra trong tư cách một thực thể của một Use Case, chỉ duy nhất sự tương tác giữa Use Case và tác nhân ngoại lai sẽ được miêu tả, nhưng khi cảnh kịch được quan sát và được chỉ ra theo hướng là một thực thể của một sự cộng tác, thì sự tương tác giữa các lớp/đối tượng phía bên trong hệ thống cũng sẽ được miêu tả.

PHẦN CÂU HỎI

❖ **Hỏi:** Một tác nhân (Actor) trong một Use Case luôn là một con người

▶ **Đáp:** Sai, tác nhân là một người hoặc một vật nào đó tương tác với hệ thống.

❖ **Hỏi:** Hệ thống khác cũng có thể đóng vai trò tác nhân trong một Use Case?

▶ **Đáp:** Đúng

❖ **Hỏi:** Mỗi hệ thống chỉ có một Use Case?

▶ **Đáp:** Sai

❖ **Hỏi:** Biểu đồ Use case mô tả chức năng hệ thống?

▶ **Đáp:** Đúng

☐ ☐ ☐

Chương 5: MÔ HÌNH ĐỐI TƯỢNG

☐ ☐ ☐

1- Lớp, đối tượng và quan hệ – các thành phần cơ bản của mô hình:

Trong mô hình hóa hướng đối tượng, những phần tử cấu thành căn bản nhất của mô hình là lớp, đối tượng và mối quan hệ giữa chúng với nhau. Lớp và đối tượng sẽ mô hình hóa những gì có trong hệ thống mà chúng ta muốn miêu tả, các mối quan hệ sẽ biểu thị cấu trúc. Động tác phân lớp (classification) đã được sử dụng từ hàng ngàn năm nay để đơn giản hóa việc miêu tả các hệ thống phức tạp. Khi loài người biết đến việc lập trình hướng đối tượng để xây dựng các hệ thống phần mềm thì lớp và các mối quan hệ của chúng được chuyển thành các dòng code cụ thể.

◆ 1.1- Đối tượng (Object)

Một đối tượng là một sự tượng trưng cho một thực thể, hoặc là thực thể tồn tại trong thế giới đời thực hoặc thực thể mang tính khái niệm. Một đối tượng có thể tượng trưng cho cái gì đó cụ thể, ví dụ như một chiếc xe ô tô chở hàng của bạn hoặc chiếc máy tính của tôi, hoặc tượng trưng cho một khái niệm ví dụ như một quy trình hóa học, một giao dịch trong nhà băng, một lời đặt hàng, những thông tin trong quá trình sử dụng tín dụng của khách hàng hay một tỷ lệ tiền lãi.

Cũng có những đối tượng (ví dụ như các đối tượng thực thi một trong hệ thống phần mềm) không thật sự tồn tại ở ngoài thế giới thực, nhưng là kết quả dẫn xuất từ quá trình nghiên cứu cấu trúc và ứng xử của các đối tượng ngoài thế giới thực. Những đối tượng đó, dù là bằng cách này hay cách khác, đều liên quan đến quan niệm của chúng ta về thế giới thực.

Một đối tượng là một khái niệm, một sự trừu tượng hóa, hoặc là một đồ vật với ranh giới và ý nghĩa được định nghĩa rõ ràng cho một ứng dụng nào đó. Mỗi đối tượng trong một hệ thống đều có ba đặc tính: trạng thái, ứng xử và sự nhận diện.

◆ 1.2- Trạng thái, ứng xử và nhận diện của đối tượng

Trạng thái (state) của một đối tượng là một trong những hoàn cảnh nơi đối tượng có thể tồn tại. Trạng thái của một đối tượng thường sẽ thay đổi theo thời gian, và nó được định nghĩa qua một tổ hợp các thuộc tính, với giá trị của các thuộc tính này cũng như mối quan hệ mà đối tượng có thể có với các đối tượng khác. Ví dụ một danh sách ghi danh cho một lớp học trong hệ thống trường học có thể có hai trạng thái: trạng thái đóng và trạng thái mở. Nếu danh sách sinh viên ghi danh cho lớp học này còn nhỏ hơn số tối đa cho phép (ví dụ là 10), thì trạng thái của bảng ghi danh này là mở. Một khi đã đủ 10 sinh viên ghi danh cho lớp, danh sách sẽ chuyển sang trạng thái đóng.

Ứng xử (Behaviour) xác định một đối tượng sẽ phản ứng như thế nào trước những yêu cầu từ các đối tượng khác, nó tiêu biểu cho những gì mà đối tượng này có thể làm. Ứng xử được thực thi qua loạt các Phương thức (operation) của

đối tượng. Trong ví dụ trường đại học, một đối tượng bảng ghi danh lớp học có thể có ứng xử là bổ sung thêm một sinh viên hay xóa đi tên của một sinh viên khi sinh viên đăng ký học hay bãi bỏ đăng ký.

Sự nhận diện (Identity) đảm bảo rằng mỗi đối tượng là duy nhất – dù trạng thái của nó có thể giống với trạng thái của các đối tượng khác. Ví dụ, khóa học đại số 101 chương 1 và khóa học đại số 101 chương 2 là hai đối tượng trong hệ thống ghi danh trường học. Mặc dù cả hai đều thuộc loại bảng ghi danh, mỗi khóa học vẫn có sự nhận dạng duy nhất của mình.

1.3- Lớp (Class):

Một lớp là một lời miêu tả của một nhóm các đối tượng có chung thuộc tính, chung phương thức (ứng xử), chung các mối quan hệ với các đối tượng khác và chung ngữ nghĩa (semantic). Nói như thế có nghĩa lớp là một khuôn mẫu để tạo ra đối tượng. Mỗi đối tượng là một thực thể của một lớp nào đó và một đối tượng không thể là kết quả thực thể hóa của nhiều hơn một lớp. Chúng ta sử dụng khái niệm lớp để bàn luận về các hệ thống và để phân loại các đối tượng mà chúng ta đã nhận dạng ra trong thế giới thực.

Một lớp tốt sẽ nắm bắt một và chỉ một sự trừu tượng hóa - nó phải có một chủ đề chính. Ví dụ, một lớp vừa có khả năng giữ tất cả các thông tin về một sinh viên và thông tin về tất cả những lớp học mà người sinh viên đó đã trải qua trong nhiều năm trước không phải là một lớp tốt, bởi nó không có chủ đề chính. Lớp này cần phải được chia ra làm hai lớp liên quan đến nhau: lớp sinh viên và lớp lịch sử của sinh viên.

Khi tạo dựng mô hình cũng như thật sự xây dựng các hệ thống doanh nghiệp, các hệ thống thông tin, máy móc hoặc các loại hệ thống khác, chúng ta cần sử dụng các khái niệm của chính phạm vi vấn đề để khiến cho mô hình dễ hiểu và dễ giao tiếp hơn. Nếu chúng ta xây dựng hệ thống cho một công ty bảo hiểm, mô hình cần phải dựa trên các khái niệm của ngành bảo hiểm. Nếu chúng ta xây dựng một hệ thống cho quân đội, thì các khái niệm của thế giới quân sự cần phải được sử dụng khi mô hình hóa hệ thống. Một hệ thống dựa trên các khái niệm chính của một ngành doanh nghiệp nào đó có thể dễ được thiết kế lại cho phù hợp với những quy chế, chiến lược và qui định mới, bởi chúng ta chỉ cần cân bằng và khắc phục sự chênh lệch giữa công việc cũ và công việc mới. Khi các mô hình được xây dựng dựa trên các khái niệm lấy ra từ cuộc đời thực và dựa trên các khái niệm thuộc phạm vi vấn đề, hướng đối tượng sẽ là một phương pháp rất thích hợp bởi nền tảng của phương pháp hướng đối tượng là các lớp, đối tượng và mối quan hệ giữa chúng.

Một lớp là lời miêu tả cho một dạng đối tượng trong bất kỳ một hệ thống nào đó – hệ thống thông tin, hệ thống kỹ thuật, hệ thống nhúng thời gian thực, hệ thống phân tán, hệ thống phần mềm và hệ thống doanh thương. Các vật dụng (artifact) trong một doanh nghiệp, những thông tin cần được lưu trữ, phân tích hoặc các vai trò mà một tác nhân đảm nhận trong một doanh nghiệp thường sẽ trở thành các lớp trong các hệ thống doanh nghiệp và hệ thống thông tin.

Ví dụ về các lớp trong doanh nghiệp và các hệ thống thông tin:

Khách hàng
Bản thương thuyết
Hóa đơn
Món nợ
Tài sản
Bản công bố giá cổ phiếu

Các lớp trong một hệ thống kỹ thuật thường bao gồm các đối tượng kỹ thuật, ví dụ như máy móc được sử dụng trong hệ thống:

Sensor
Màn hình
I/O card
Động cơ
Nút bấm

Lớp điều khiển

Các hệ thống phần mềm thường có các lớp đại diện cho các thực thể phần mềm trong một hệ điều hành:

File

Chương trình chạy được

Trang thiết bị

Icon

Cửa sổ

Thanh kéo

1.4- Biểu đồ lớp (Class diagram):

Một biểu đồ lớp là một dạng mô hình tĩnh. Một biểu đồ lớp miêu tả hướng nhìn tĩnh của một hệ thống bằng các khái niệm lớp và mối quan hệ giữa chúng với nhau. Mặc dù nó cũng có những nét tương tự với một mô hình dữ liệu, nhưng nên nhớ rằng các lớp không phải chỉ thể hiện cấu trúc thông tin mà còn miêu tả cả hình vi. Một trong các mục đích của biểu đồ lớp là tạo nền tảng cho các biểu đồ khác, thể hiện các khía cạnh khác của hệ thống (ví dụ như trạng thái của đối tượng hay cộng tác động giữa các đối tượng, được chỉ ra trong các biểu đồ động). Một lớp trong một biểu đồ lớp có thể được thực thi trực tiếp trong một ngôn ngữ hướng đối tượng có hỗ trợ trực tiếp khái niệm lớp. Một biểu đồ lớp chỉ chỉ ra các lớp, nhưng bên cạnh đó còn có một biến tấu hơi khác đi một chút chỉ ra các đối tượng thật sự là các thực thể của các lớp này (biểu đồ đối tượng).

Để tạo một biểu đồ lớp, đầu tiên ta phải nhận diện và miêu tả các lớp. Một khi đã có một số lượng các lớp, ta sẽ xét đến quan hệ giữa các lớp đó với nhau.

2- Tìm lớp:

Hầu như không có một công thức chung cho việc phát hiện ra các lớp. Đi tìm các lớp là một công việc đòi hỏi trí sáng tạo và cần phải được thực thi với sự trợ giúp của chuyên gia ứng dụng. Vì qui trình phân tích và thiết kế mang tính vòng lặp, nên danh sách các lớp sẽ thay đổi theo thời gian. Tập hợp ban đầu của các lớp tìm ra chưa chắc đã là tập hợp cuối cùng của các lớp sau này sẽ được thực thi và biến đổi thành code. Vì thế, thường người ta hay sử dụng đến khái niệm các *lớp ứng cử viên* (Candidate Class) để miêu tả tập hợp những lớp đầu tiên được tìm ra cho hệ thống.

Như đã nói trong phần 2.10 (Thực hiện Trường hợp sử dụng), trường hợp sử dụng là những lời miêu tả chức năng của hệ thống, còn trách nhiệm thực thi thuộc về các đối tượng cộng tác thực thi chức năng đó. Nói một cách khác, chúng ta đi tìm các lớp là để tiến tới tìm giải pháp cung cấp những ứng xử hướng ngoại đã được xác định trong các trường hợp sử dụng.

Có nhiều phương pháp khác nhau để thực hiện công việc đó. Có phương pháp đề nghị tiến hành phân tích phạm vi bài toán, chỉ ra tất cả các lớp thực thể (thuộc phạm vi bài toán) với mỗi quan hệ của chúng với nhau. Sau đó nhà phát triển sẽ phân tích từng trường hợp sử dụng và phân bổ trách nhiệm cho các lớp trong mô hình phân tích (analysis model), nhiều khi sẽ thay đổi chúng hoặc bổ sung thêm

các lớp mới. Có phương pháp đề nghị nên lấy các trường hợp sử dụng làm nền tảng để tìm các lớp, làm sao trong quá trình phân bổ trách nhiệm thì mô hình phân tích của phạm vi bài toán sẽ từng bước từng bước được thiết lập.

2.1- Phân tích phạm vi bài toán để tìm lớp:

Quá trình phân tích phạm vi bài toán thường được bắt đầu với các *khái niệm then chốt* (Key Abstraction), một công cụ thường được sử dụng để nhận diện và lọc ra các lớp ứng cử viên (Candidate class).

2.1.1- Khái niệm then chốt

Hãy lấy ví dụ một nhà băng ABC, điều đầu tiên ta nghĩ tới là gì? Tiền! Bên cạnh đó, ABC còn phải có những thực thể liên quan tới tiền như sau:

Khách hàng

Sản phẩm (các tài khoản được coi là các sản phẩm của một nhà băng)

Lực lượng nhân viên

Ban quản trị nhà băng

Phòng máy tính trong nhà băng

Những thực thể này được gọi là các khái niệm then chốt cho những gì mà nhà băng có thể có. Khái niệm then chốt hoặc mang tính cấu trúc (structural) hoặc mang tính chức năng (functional). Thực thể mang tính cấu trúc là những thực thể vật lý tương tác với nhà băng, ví dụ khách hàng. Thực thể mang tính chức năng là những chức năng mà nhà băng phải thực hiện, ví dụ duy trì một tài khoản hoặc chuyển tiền từ tài khoản này sang tài khoản khác. Khái niệm then chốt là các thực thể ta để ý đến đầu tiên. Chúng rất quan trọng vì giúp ta:

Định nghĩa ranh giới của vấn đề

Nhấn mạnh đến các thực thể có liên quan đến thiết kế của hệ thống

Loại bỏ thực thể nằm ngoài phạm vi hệ thống

Các khái niệm then chốt thường sẽ trở thành các lớp trong mô hình phân tích

Một khái niệm then chốt tóm lại là một lớp hay đối tượng thuộc chuyên ngành của phạm vi bài toán. Khi trình bày với người sử dụng, chúng có một ánh xạ 1-1 giữa với những thực thể liên quan tới người sử dụng như hóa đơn, séc, giấy đề nghị rút tiền, sổ tiết kiệm, thẻ rút tiền tự động, nhân viên thu ngân, nhân viên nhà băng, các phòng ban,....

Mức độ trừu tượng:

Khi phân tích phạm vi bài toán, cần chú ý rằng mức độ trừu tượng của các khái niệm then chốt là rất quan trọng, bởi mức độ trừu tượng quá cao hay quá thấp đều rất dễ gây nhầm lẫn.

Mức trừu tượng quá cao dẫn tới những định nghĩa quá khái quát về một thực thể, tạo nên một cái nhìn vĩ mô và thường không nhắm vào một mục tiêu cụ thể. Ví dụ trong một nhà băng, ta không thể chọn khái niệm then chốt là "người", bởi nó sẽ dẫn đến lời miêu tả: "Một người đến nhà băng để gửi tiền vào, và số tiền đó được một người khác tiếp nhận." – trong khi một yêu cầu quan trọng ở đây là phải phân biệt giữa nhân viên với khách hàng vì chức năng của họ là khác hẳn nhau.

Tương tự như vậy, mức trừu tượng quá thấp cũng dễ gây hiểu lầm, bởi những thông tin quá vụn vặt chưa thích hợp với thời điểm này. Ví dụ những quyết định dạng:

Form mở tài khoản đòi hỏi tất cả 15 Entry.

Những dữ liệu trên Form này đều phải được căn phải.

Không có nhiều chỗ để ghi địa chỉ của khách hàng trên Form.

nên được để dành cho các giai đoạn sau.

Vài điểm cần chú ý về khái niệm then chốt:

Những thực thể xuất hiện đầu tiên trong óc não chúng ta là những thực thể dễ có khả năng trở thành khái niệm then chốt cho một vấn đề định trước.

Mỗi lần tìm thấy một khái niệm then chốt mới, cần xem xét nó theo cách nhìn của vấn đề, có thể hỏi các câu hỏi sau:

Những chức năng nào có thể được thực hiện đối với thực thể này?

Điều gì khiến những thực thể loại này được tạo ra?

Nếu không có câu trả lời thích hợp, cần phải suy nghĩ lại về thực thể đó.

Mỗi khái niệm then chốt mới cần phải được đặt tên cho thích hợp, miêu tả đúng chức năng của khái niệm.

2.1.2- Nhận dạng lớp và đối tượng

Nắm vững khái niệm lớp, chúng ta có thể tương đối dễ dàng tìm thấy các lớp và đối tượng trong phạm vi vấn đề. Một nguyên tắc thô sơ thường được áp dụng là **danh từ** trong các lời phát biểu bài toán thường là các ứng cử viên để chuyển thành lớp và đối tượng.

Một số gợi ý thực tế cho việc tìm lớp trong phạm vi vấn đề:

Bước đầu tiên là cần phải tập trung nghiên cứu kỹ:

Các danh từ trong những lời phát biểu bài toán

Kiến thức chuyên ngành thuộc phạm vi bài toán

Các Trường hợp sử dụng

Ví dụ trong lời phát biểu "Có một số tài khoản mang lại tiền lãi", ta thấy có hai danh từ là **tài khoản** và **tiền lãi**. Chúng có thể là các lớp tiềm năng cho mô hình nhà băng lẻ.

Thứ hai, chúng ta cần chú ý đến các nhóm vật thể trong hệ thống hiện thời như:

Các thực thể vật lý của hệ thống: những vật thể tương tác với hệ thống, ví dụ khách hàng.

Các vật thể hữu hình: các vật thể vật lý mà ta có thể nhìn và sờ thấy. Ví dụ như công cụ giao thông, sách vở, một con người, một ngôi nhà,... Trong một nhà băng ABC, đó có thể là tập séc, phiếu đề nghị rút tiền, sổ tiết kiệm, các loại Form cần thiết.

Các sự kiện (Events): Một chiếc xe bị hỏng, một cái cửa được mở ra. Trong một nhà băng là sự đáo hạn một tài khoản đầu tư, hiện tượng rút quá nhiều tiền mặt trong một tài khoản bình thường.

Các vai trò (Role): Ví dụ như mẹ, khách hàng, người bán hàng, Trong một nhà băng, vai trò có thể là nhân viên, nhà quản trị, khách hàng,...

Các sự tương tác (Interactions): Ví dụ việc bán hàng là một chuỗi tương tác bao gồm khách hàng, người bán hàng và sản phẩm. Trong một nhà băng, việc mở một tài khoản mới sẽ yêu cầu một chuỗi tương tác giữa nhân viên và khách hàng.

Vị trí (Location): Một đồ vật nào đó hoặc một người nào đó được gán cho một vị trí nào đó. Ví dụ: Ôtô đối với nhà để xe. Trong một nhà băng ta có thể thấy nhân viên thu ngân luôn đứng ở cửa sổ của mình.

Đơn vị tổ chức (Organisation Unit): Ví dụ các phòng ban, phòng trưng bày sản phẩm, các bộ phận. Trong một nhà băng có thể có bộ phận tài khoản bình thường, bộ phận tài khoản tiết kiệm, bộ phận tài khoản đầu tư.

Bên cạnh đó, còn nhiều câu hỏi khác giúp ta nhận dạng lớp. Ví dụ như:

Ta có thông tin cần được lưu trữ hoặc cần được phân tích không? Nếu có thông tin cần phải được lưu trữ, biến đổi, phân tích hoặc xử lý trong một phương thức nào đó thì chắc chắn đó sẽ là ứng cử viên cho lớp. Những thông tin này có thể là một khái niệm luôn cần phải được ghi trong hệ thống hoặc là sự kiện, giao dịch xảy ra tại một thời điểm cụ thể nào đó.

Ta có các hệ thống ngoại vi không? Nếu có, thường chúng cũng đáng được quan tâm tới khi tạo dựng mô hình. Các hệ thống bên ngoài có thể được coi là các lớp chứa hệ thống của chúng ta hoặc tương tác với hệ thống của chúng ta.

Chúng ta có các mẫu, thư viện lớp, thành phần và những thứ khác không? Nếu chúng ta có mẫu, thư viện, thành phần từ các dự án trước (xin được của các bạn đồng nghiệp, mua được từ các nhà cung cấp) thì chúng thường cũng sẽ chứa các ứng cử viên lớp.

Có thiết bị ngoại vi mà hệ thống của chúng ta cần xử lý không? Mỗi thiết bị kỹ thuật được nối với hệ thống của chúng ta thường sẽ trở thành ứng cử viên cho lớp xử lý loại thiết bị ngoại vi này.

Chúng ta có phần công việc tổ chức không? Miêu tả một đơn vị tổ chức là công việc được thực hiện với các lớp, đặc biệt là trong các mô hình doanh nghiệp.

2.1.3- Tổng kết về các nguồn thông tin cho việc tìm lớp:

Nhìn chung, các nguồn thông tin chính cần đặc biệt chú ý khi tìm lớp là:

- Các lời phát biểu yêu cầu
- Các Trường hợp sử dụng
- Sự trợ giúp của các chuyên gia ứng dụng
- Nghiên cứu hệ thống hiện thời

Loạt các lớp đầu tiên được nhận dạng qua đây thường được gọi là các lớp ứng cử viên (Candidate Class). Ngoài ra, nghiên cứu những hệ thống tương tự cũng có thể sẽ mang lại cho ta các lớp ứng cử viên khác:

Khi nghiên cứu hệ thống hiện thời, hãy để ý đến các danh từ và các khái niệm then chốt để nhận ra lớp ứng cử viên. Không nên đưa các lớp đã được nhận diện một lần nữa vào mô hình chỉ bởi vì chúng được nhắc lại ở đâu đó theo một tên gọi khác. Ví dụ, một hệ thống nhà băng có thể coi cùng một khách hàng với nhiều vị trí khác nhau là nhiều khách hàng khác nhau. Cần chú ý khi phân tích

những lời miêu tả như thế để tránh dẫn đến sự trùng lặp trong quá trình nhận diện lớp.

Có nhiều nguồn thông tin mà nhà thiết kế cần phải chú ý tới khi thiết kế lớp và chỉ khi làm như vậy, ta mới có thể tin chắc về khả năng tạo dựng một mô hình tốt. Hình sau tổng kết các nguồn thông tin kể trên.

Các trường hợp sử dụng là nguồn tốt nhất cho việc nhận diện lớp và đối tượng. Cần nghiên cứu kỹ các Trường hợp sử dụng để tìm các thuộc tính (attribute) báo trước sự tồn tại của đối tượng hoặc lớp tiềm năng. Ví dụ nếu Trường hợp sử dụng yêu cầu phải đưa vào một số tài khoản (account-number) thì điều này trở tới sự tồn tại của một đối tượng tài khoản.

Một nguồn khác để nhận ra lớp/đối tượng là các Input và Output của hệ thống. Nếu Input bao gồm tên khách hàng thì đây là tín hiệu cho biết sự tồn tại của một đối tượng khách hàng, bởi nó là một attribute của khách hàng.

Nói chuyện với người sử dụng cũng gợi mở đến các khái niệm then chốt. Thường thì người sử dụng miêu tả hệ thống theo lối cần phải đưa vào những gì và mong chờ kết quả gì. Thông tin đưa vào và kết quả theo lối miêu tả của người sử dụng cần phải được tập hợp lại với nhau để nhận dạng khái niệm then chốt.

2.2- Các lớp ứng cử viên:

Theo các bước kể trên trong phần đầu giai đoạn phân tích, ta đã miêu tả được một số lớp khác nhau. Những lớp này được gọi là các lớp ứng cử viên, chúng thể hiện những lớp có khả năng tồn tại trong một hệ thống cho trước. Mặc dù vậy,

đây vẫn có thể chưa phải là kết quả chung cuộc, một số lớp ứng cử viên có thể sẽ bị loại bỏ trong các bước sau vì không thích hợp.

Giai đoạn đầu khi định nghĩa các lớp ứng cử viên, ta chưa nên cố gắng thanh lọc các lớp, hãy tập trung cáo mục tiêu nghiên cứu bao quát và toàn diện từ nhiều nguồn thông tin khác nhau để không bỏ sót nhiều khía cạnh cần xử lý.

Ví dụ trong nhà một ngân hàng, các lớp ứng cử viên có thể là:

Khách hàng
Các loại tài khoản khác nhau
Sec, sổ tiết kiệm, đơn,
Phiếu yêu cầu mở tài khoản mới
Thẻ ATM
Bản in thông tin về tài khoản
Giấy chứng nhận tài khoản đầu tư
Thẻ xếp hàng (Token), số thứ tự
Nhân viên
Nhân viên thu ngân

2.3- Loại bỏ các lớp ứng cử viên không thích hợp:

Có rất nhiều loại lớp ứng cử viên không thích hợp cần phải được loại bỏ:

Lớp dư, thừa: Khi có hơn một lớp định nghĩa cùng một thực thể, nên giữ lại lớp tốt nhất và loại bỏ những lớp khác. Ví dụ, trong một nhà băng có hai lớp chủ tài khoản và khách hàng. Cả hai lớp biểu hiện cùng một thực thể và vì thế chỉ cần giữ lại một.

Lớp không thích hợp: Lớp định nghĩa ra những thực thể không liên quan đến vấn đề thực tại. Mọi lớp không xuất phát từ phạm vi ứng dụng cần phải được loại bỏ. Ví dụ, lớp của các máy đếm tiền bên casse trong một nhà băng có thể là một ứng cử viên cho khái niệm lớp không thích hợp.

Lớp không rõ ràng: Lớp không có chức năng cụ thể được gọi là các lớp không rõ ràng. Lớp tồn tại và có giá trị sử dụng trong một hệ thống là lớp có một chức năng đã được nhận diện và xác định rõ ràng. Các lớp không rõ ràng cần phải được định nghĩa lại hoặc loại bỏ. Ví dụ quan sát nhiều bộ phận khác nhau trong một nhà băng

ABC. Một trong những bộ phận đã được nhận diện có thể là bộ phận hành chính. Vì phạm vi cho quá trình vi tính hóa của nhà băng hiện thời chưa bao gồm mảng hành chính nên lớp này có thể được coi là một lớp không rõ ràng (vì không có chức năng rõ ràng trong hệ thống cần xây dựng trước mắt).

Tương tự, những thuộc tính và phương thức không rõ ràng cần phải được loại ra khỏi danh sách các lớp ứng cử viên. Chúng không cần phải bị xoá hẳn, nhưng cần được đưa ra ngoài để ta có thể nhìn rõ các lớp cần thiết đã được nhận diện. Các ứng xử đó sau này có thể được gán cho các lớp thích hợp hơn.

Các lớp chỉ là vai trò (Role) đối với một lớp khác: Hãy loại bỏ tất cả các vai trò và giữ lại lớp chính. Ví dụ nhà quản trị, nhân viên thu ngân, người chạy giấy rất có thể chỉ là vai trò của lớp nhân viên. Hãy giữ lại lớp nhân viên và loại bỏ tất cả những lớp khác chỉ là vai trò.

Một lớp không cung cấp ứng xử cần thiết hoặc thuộc tính cần thiết có thể sẽ là lớp không cần thiết. Nhiều khi, có thể có một lớp chẳng cung cấp một thuộc tính hoặc ứng xử nào mà chỉ định nghĩa một tập hợp các mối quan hệ. Những lớp như thế cần phải được nghiên cứu kỹ để xác định sự liên quan với hệ thống. Ví dụ một khách hàng có thể được định nghĩa là khách hàng quan trọng hay khách hàng bình thường tùy theo mối quan hệ mà anh ta có với nhà băng trong tư cách chủ nhân tài khoản.

Tất cả những công cụ xây dựng (Implementation constructs) ví dụ như stack, arrays, link lists, ... cần phải được đưa ra khỏi mô hình phân tích. Chúng sẽ được dùng tới trong giai đoạn xây dựng phần mềm.

Một lớp có tên mang tính động từ có thể đơn giản chỉ là một hàm chứ không phải là một lớp. Ví dụ "rút tiền" không cần phải được coi là một lớp, nó có thể là chức năng của một lớp.

Lớp chỉ có một hàm hoặc chỉ là sự miêu tả việc thực hiện một chức năng nào đó có thể đơn giản chỉ là một hàm, hoặc quá trình trừu tượng hóa dữ liệu (data abstraction) ở đây chưa được thực hiện đầy đủ.

Lớp không có hàm là một thiếu sót trong mô hình. Vấn đề hàm thành phần (phương thức) của lớp này chưa được suy nghĩ thấu đáo.

3- Lớp và đối tượng trong UML:

UML thể hiện lớp bằng hình chữ nhật có 3 phần. Phần thứ nhất chứa tên lớp. Trong phần thứ hai là thuộc tính và các dữ liệu thành phần của lớp và trong phần thứ ba là các phương thức hay hàm thành phần của lớp.

3.1- Tên lớp (lass name):

Tên lớp được in đậm (bold) và căn giữa. Tên lớp phải được dẫn xuất từ phạm vi vấn đề và rõ ràng như có thể. Vì thế nó là danh từ, ví dụ như tài khoản, nhân viên,....

3.2- Thuộc tính (attribute):

Lớp có thuộc tính miêu tả những đặc điểm của đối tượng. Giá trị của thuộc tính thường là những dạng dữ liệu đơn giản được đa phần các ngôn ngữ lập trình hỗ trợ như Integer, Boolean, Floats, Char, ...

Thuộc tính có thể có nhiều mức độ trông thấy được (visibility) khác nhau, miêu tả liệu thuộc tính đó có thể được truy xuất từ các lớp khác, khác với lớp định nghĩa ra nó. Nếu thuộc tính có tính trông thấy là công cộng (public), thì nó có thể được nhìn thấy và sử dụng ngoài lớp đó. Nếu thuộc tính có tính trông thấy là riêng (private), bạn sẽ không thể truy cập nó từ bên ngoài lớp đó. Một tính trông thấy khác là bảo vệ (protected), được sử dụng chung với công cụ khái quát hóa và chuyên biệt hóa. Nó cũng giống như các thuộc tính riêng nhưng được thừa kế bởi các lớp dẫn xuất.

Trong UML, thuộc tính công cộng mang kí hiệu "+" và thuộc tính riêng mang dấu "-".

Giá trị được gán cho thuộc tính có thể là một cách để miêu tả trạng thái của đối tượng. Mỗi lần các giá trị này thay đổi là biểu hiện cho thấy có thể đã xảy ra một sự thay đổi trong trạng thái của đối tượng.

Lưu ý: Mọi đặc điểm của một thực thể là những thông tin cần lưu trữ đều có thể chuyển thành thuộc tính của lớp miêu tả loại thực thể đó.

3.3- Phương thức (methods):

Phương thức định nghĩa các hoạt động mà lớp có thể thực hiện. Tất cả các đối tượng được tạo từ một lớp sẽ có chung thuộc tính và phương thức. Phương thức được sử dụng để xử lý thay đổi các thuộc tính cũng như thực hiện các công việc khác. Phương thức thường được gọi là các hàm (function), nhưng chúng nằm trong một lớp và chỉ có thể được áp dụng cho các đối tượng của lớp này. Một phương thức được miêu tả qua tên, giá trị trả về và danh sách của 0 cho tới

nhieu tham số. Lúc thi hành, phương thức được gọi kèm theo một đối tượng của lớp. Vì nhóm các phương thức miêu tả những dịch vụ mà lớp có thể cung cấp nên chúng được coi là giao diện của lớp này. Giống như thuộc tính, phương thức cũng có tính trông thấy được như công cộng, riêng và bảo vệ.

3.4- Kí hiệu đối tượng:

Đối tượng là thực thể của các lớp nên kí hiệu dùng cho đối tượng cũng là kí hiệu dùng cho lớp.

Hình trên được đọc như sau: CAH là đối tượng của lớp AccountHolder. Các thuộc tính được gán giá trị, đây là các giá trị khi lớp được thực thể hóa. Chú ý rằng kí hiệu đối tượng không chứa phần phương thức.

4- Quan hệ giữa các lớp:

Biểu đồ lớp thể hiện các lớp và các mối quan hệ giữa chúng. Quan hệ giữa các lớp gồm có bốn loại:

Liên hệ (Association)

Khái quát hóa (Generalization)

Phụ thuộc (Dependency)

Nâng cấp (Refinement)

Một liên hệ là một sự nối kết giữa các lớp, cũng có nghĩa là sự nối kết giữa các đối tượng của các lớp này. Trong UML, một liên hệ được định nghĩa là một mối quan hệ miêu tả một tập hợp các nối kết (links), trong khi nối kết được định nghĩa là một sự liên quan về ngữ nghĩa (semantic connection) giữa một nhóm các đối tượng.

Khái quát hóa là mối quan hệ giữa một yếu tố mang tính khái quát cao hơn và một yếu tố mang tính chuyên biệt hơn. Yếu tố mang tính chuyên biệt hơn có thể chứa chỉ các thông tin bổ sung. Một thực thể (một đối tượng là một thực thể của một lớp) của yếu tố mang tính chuyên biệt hơn có thể được sử dụng ở bất cứ nơi nào mà đối tượng mang tính khái quát hóa hơn được phép.

Sự phụ thuộc là một mối quan hệ giữa các yếu tố, gồm một yếu tố mang tính độc lập và một yếu tố mang tính phụ thuộc. Một sự thay đổi trong yếu tố độc lập sẽ ảnh hưởng đến yếu tố phụ thuộc.

Một sự nâng cấp là mối quan hệ giữa hai lời miêu tả của cùng một sự vật, nhưng ở những mức độ trừu tượng hóa khác nhau.

5- Liên hệ (Association)

Một liên hệ là một sự nối kết giữa các lớp, một liên quan về ngữ nghĩa giữa các đối tượng của các lớp tham gia. Liên hệ thường thường mang tính hai chiều, có nghĩa khi một đối tượng này có liên hệ với một đối tượng khác thì cả hai đối tượng này nhận thấy nhau. Một mối liên hệ biểu thị bằng các đối tượng của hai lớp có nối kết với nhau, ví dụ rằng "chúng biết về nhau", "được nối với nhau", "cứ mỗi X lại có một Y",.... Lớp và liên hệ giữa các lớp là những công cụ rất mạnh mẽ cho việc mô hình hóa các hệ thống phức tạp, ví dụ như cấu trúc sản phẩm, cấu trúc văn bản và tất cả các cấu trúc thông tin khác.

Mỗi liên kết được thể hiện trong biểu đồ UML bằng một đường thẳng nối hai lớp (hình 4.18).

5.1- Vai trò trong liên hệ

Một liên hệ có thể có các vai trò (**Roles**). Các vai trò được nối với mỗi lớp bao chứa trong quan hệ. Vai trò của một lớp là chức năng mà nó đảm nhận nhìn từ góc nhìn của lớp kia. Tên vai trò được viết kèm với một mũi tên chỉ từ hướng lớp chủ nhân ra, thể hiện lớp này đóng vai trò như thế nào đối với lớp mà mũi tên chỉ đến.

Trong ví dụ trên: một khách hàng có thể là chủ nhân của một tài khoản và tài khoản được chiếm giữ bởi khách hàng. Đường thẳng thể hiện liên hệ giữa hai lớp.

Một số điểm cần chú ý khi đặt tên vai trò:

Tên vai trò có thể bỏ đi nếu trùng với tên lớp

Tên vai trò phải là duy nhất.

Tên vai trò phải khác với các thuộc tính của lớp.

Tên vai trò phải miêu tả được chức năng mà lớp này đảm nhận trong quan hệ, tức cần phải là các khái niệm lấy ra từ phạm vi vấn đề, giống như tên các lớp.

5.2- Liên hệ một chiều (Uni-Directional Association)

Ta cũng có thể sử dụng mối liên hệ một chiều bằng cách thêm một mũi tên và một đầu của đường thẳng nối kết. Mũi tên chỉ ra rằng sự nối kết chỉ có thể được sử dụng duy nhất theo chiều của mũi tên.

Biểu đồ phần 5.15 thể hiện rằng giữa hai lớp có liên hệ, nhưng không hề có thông tin về số lượng các đối tượng trong quan hệ. Ta không thể biết một khách hàng có thể có bao nhiêu tài khoản và một tài khoản có thể là của chung cho bao nhiêu khách hàng. Trong UML, loại thông tin như thế được gọi là số lượng phần tử (Cardinality) trong quan hệ.

5.3- Số lượng (Cardinality) trong liên hệ:

Biểu đồ trên nói rõ một khách hàng có thể mở một hoặc nhiều tài khoản và một tài khoản có thể thuộc về một cho tới ba khách hàng.

Số lượng được ghi ở phía đầu đường thẳng thể hiện liên hệ, sát vào lớp là miền áp dụng của nó. Phạm vi của số lượng phần tử trong liên hệ có thể từ 0-tới-1 (0..1), 0-tới-nhiều (0..* hay), một-tới-nhiều (1..), hai (2), năm-tới-mười một (5..11). Cũng có thể miêu tả một dãy số ví dụ (1,4,6, 8..12). Giá trị mặc định là 1.

Hình trên là ví dụ cho một biểu đồ lớp tiêu biểu. Biểu đồ giải thích rằng bộ phận dịch vụ tài khoản tiết kiệm của một nhà băng có thể có nhiều tài khoản tiết kiệm nhưng tất cả những tài khoản này đều thuộc về bộ phận đó. Một tài khoản tiết kiệm về phần nó lại có thể có nhiều tài liệu, nhưng những tài liệu này chỉ thuộc về một tài khoản tiết kiệm mà thôi. Một tài khoản tiết kiệm có thể thuộc về từ 1 cho tới nhiều nhất là 3 khách hàng. Mỗi khách hàng có thể có nhiều hơn một tài khoản.

5.4- Phát hiện liên hệ:

Thường sẽ có nhiều mối liên hệ giữa các đối tượng trong một hệ thống. Quyết định liên hệ nào cần phải được thực thi là công việc thuộc giai đoạn thiết kế. Có thể tìm các mối liên hệ qua việc nghiên cứu các lời phát biểu vấn đề, các yêu cầu. Giống như danh từ đã giúp chúng ta tìm lớp, các **động từ** ở đây sẽ giúp ta tìm ra các mối quan hệ.

Một vài lời mách bảo khi tìm liên hệ:

Vị trí về mặt vật lý hoặc sự thay thế, đại diện: Mỗi cụm động từ xác định hay biểu lộ một vị trí đều là một biểu hiện chắc chắn cho liên hệ. Ví dụ: tại địa điểm, ngồi trong, ...

Sự bao chứa: Cụm động từ biểu lộ sự bao chứa, ví dụ như: là thành phần của....

Giao tiếp: Có nhiều cụm động từ biểu lộ sự giao tiếp, ví dụ truyền thông điệp, nói chuyện với, ...

Quyền sở hữu: Ví dụ: thuộc về, của, ...

Thoả mãn một điều kiện: Những cụm từ như: làm việc cho, là chồng/vợ của, quản trị,

5.5- Xử lý các liên hệ không cần thiết:

Sau khi tìm các mối liên hệ, bước tiếp theo đó là phân biệt các liên hệ cần thiết ra khỏi các liên hệ không cần thiết. Liên hệ không cần thiết có thể bao gồm những liên hệ bao chứa các lớp ứng cử viên đã bị loại trừ hoặc các liên hệ không liên quan đến hệ thống. Có những liên hệ được tạo ra nhằm mục đích tăng hiệu quả. Những liên hệ như thế là ví dụ tiêu biểu của các chi tiết thực thi và không liên quan tới giai đoạn này.

Cần chú ý phân biệt giữa hành động và mối liên hệ. Người ta thường có xu hướng miêu tả hành động như là liên hệ, bởi cả liên hệ lẫn hành động đều được dẫn xuất từ những cụm từ mang tính động từ trong bản miêu tả yêu cầu. Các hành động đã được thể hiện sai thành liên hệ cũng cần phải được loại bỏ. Khi làm việc này, có thể áp dụng một nguyên tắc: liên hệ là nối kết mang tính tĩnh giữa các đối tượng, trong khi hành động chỉ là thao tác xảy ra một lần. Hành động vì vậy nên được coi là Phương thức đối với một đối tượng chứ không phải quan hệ giữa các lớp.

Ví dụ với "Ban quản trị nhà băng đuổi việc một nhân viên", động từ "đuổi việc" thể hiện hành động. Trong khi đó với "Một nhân viên làm việc cho hãng" thì động từ "làm việc" miêu tả liên hệ giữa hai lớp nhân viên và hãng.

Trong khi cố gắng loại bỏ các liên hệ dư thừa, bạn sẽ thấy có một số liên hệ dư thừa đã "lèn vào" mô hình của chúng ta trong giai đoạn thiết kế. Hình sau chỉ ra một số loại liên hệ dư thừa cần đặc biệt chú trọng.

5.6- Nâng cấp các mối liên hệ:

Một khi các mối liên hệ cần thiết đã được nhận dạng, bước tiếp theo là nghiên cứu kỹ mô hình và nâng cấp các mối liên hệ đó.

Động tác nâng cấp đầu tiên là xem xét lại tên liên hệ, tên vai trò, đặt lại cho đúng với bản chất quan hệ mà chúng thể hiện. Mỗi liên hệ cần phải được suy xét kỹ về phương diện số lượng thành phần tham gia (Cardinality). Sự hạn định (Qualification) cho liên hệ đóng một vai trò quan trọng ở đây, bổ sung yếu tố hạn định có thể giúp làm giảm số lượng. Nếu cần thiết, hãy bổ sung các liên hệ còn

thiếu. Nghiên cứu kỹ các thuộc tính, xem liệu trong số chúng có thuộc tính nào thật ra thể hiện liên hệ. Nếu có, hãy chuyển chúng thành liên hệ. Bổ sung các thông tin và điều kiện cần thiết cũng như xem xét các mối liên hệ trong mô hình tổng thể để xác định các dạng quan hệ giữa chúng với nhau.

5.6.1- Liên hệ và yếu tố hạn định (Qualifier):

Một liên hệ được hạn định liên hệ hai lớp và một yếu tố hạn định (Qualifier) với nhau. Yếu tố hạn định là một thuộc tính hạn chế số lượng thành phần tham gia trong một mối liên hệ. Có thể hạn định các mối liên hệ một-tới-nhiều và nhiều-tới-nhiều. Yếu tố hạn định giúp phân biệt trong nhóm đối tượng của đầu nhiều của liên hệ.

Ví dụ một thư mục có nhiều tập tin. Một tập tin chỉ thuộc về một thư mục mà thôi. Trong một thư mục xác định, tên của tập tin sẽ xác định duy nhất tập tin mang tên đó. Thư mục và Tập tin là hai lớp, và tên tập tin ở đây đóng vai trò yếu tố hạn định. Một thư mục và một tên tập tin xác định một tập tin. Yếu tố hạn định ở đây đã chuyển một mối liên hệ một-tới-nhiều thành liên hệ một-tới-một.

5.6.2- Liên hệ VÀ (AND Association)

Nhà băng nợ đưa ra quy định: khách hàng khi muốn mở một tài khoản ATM phải là chủ nhân của ít nhất một tài khoản đầu tư. Trong một trường hợp như thế, mỗi liên hệ **VÀ** (AND) sẽ được thể hiện như sau:

Biểu đồ trên cho thấy một khách hàng có thể có nhiều hơn một tài khoản đầu tư có thời hạn và chỉ một tài khoản ATM. Trong biểu đồ có một mối liên hệ VÀ ngầm được áp dụng giữa nhóm tài khoản đầu tư và tài khoản ATM mà một khách hàng có thể có.

5.6.3- Liên hệ HOẶC (OR Association)

Ví dụ tại một hãng bảo hiểm nợ, cá nhân cũng công ty đều có thể ký hợp đồng bảo hiểm, nhưng cá nhân và công ty không được phép có *cùng loại* hợp đồng bảo hiểm như nhau. Trong một trường hợp như thế, giải pháp là sử dụng liên hệ HOẶC (OR Association). Một liên hệ HOẶC là một sự hạn chế đối với một nhóm hai hay nhiều liên hệ, xác định rằng đối tượng của một lớp này tại một thời điểm chỉ có thể tham gia vào nhiều nhất một trong các mối liên hệ đó.

5.6.4- Liên hệ được sắp xếp (Ordered Association)

Các mối nối kết (link) giữa các đối tượng có một trật tự ngầm định. Giá trị mặc định của trật tự này là ngẫu nhiên. Một liên hệ có trật tự rõ ràng có thể được hiểu là một liên hệ với **trật tự sắp xếp** (sort order) trong nhóm các nối kết, nó sẽ được thể hiện như sau:

Nhãn **{ordered}** được ghi gần lớp có đối tượng được sắp xếp. Biểu đồ trên được đọc là các tài khoản tiết kiệm được sắp xếp theo khách hàng.

5.6.5- Liên hệ tam nguyên (Ternary Association)

Có thể có nhiều hơn hai lớp nối kết với nhau trong một liên hệ tam nguyên.

Biểu đồ trên được đọc như sau: Một khách hàng có thể quan hệ với bộ phận đầu tư và một bộ phận đầu tư có thể có một hoặc nhiều khách hàng. Một giấy chứng nhận tài khoản đầu tư sẽ xuất hiện qua quan hệ giữa khách hàng và bộ phận đầu tư.

5.6.6- Lớp liên hệ (Association Class)

Một lớp có thể được đính kèm theo một liên hệ, trong trường hợp này nó sẽ được gọi là một lớp liên hệ. Một lớp liên hệ không được nối tới bất kỳ một lớp nào của mỗi liên hệ, mà tới chính bản thân mỗi liên hệ. Cũng giống như một lớp bình thường, lớp liên hệ có thể có thuộc tính, Phương thức và các quan hệ khác. Lớp liên hệ được sử dụng để bổ sung thêm thông tin cho nối kết (link), ví dụ như thời điểm nối kết được thiết lập. Mỗi nối kết của liên hệ gắn liền với một đối tượng của lớp liên hệ.

Ví dụ sau miêu tả một hệ thống thang máy. Bộ phận điều khiển chỉ huy bốn thang máy. Cho mỗi nối kết giữa nhóm thang máy và bộ phận điều khiển có một hàng xếp (queue). Mỗi hàng lưu trữ những yêu cầu kể cả từ phía bộ phận điều khiển lẫn từ phía thang máy (những nút bấm bên trong thang). Khi bộ phận điều khiển chọn một thang máy để thực hiện một lời yêu cầu đến từ một hành khách đứng ngoài thang máy (một hành khách trên hành lang), nó sẽ đọc các hàng và chọn thang máy nào có hàng yêu cầu ngắn nhất.

5.6.7- Liên hệ đệ quy (Recursive Association)

Có thể liên kết một lớp với bản thân nó trong một mối liên hệ. Mỗi liên hệ ở đây vẫn thể hiện một sự liên quan ngữ nghĩa, nhưng các đối tượng được nối kết đều thuộc chung một lớp. Một liên hệ của một lớp với chính bản thân nó được gọi là một liên hệ đệ quy, và là nền tảng cho rất nhiều mô hình phức tạp, sử dụng ví dụ để miêu tả các cấu trúc sản phẩm. Hình 5.25 chỉ ra một ví dụ của liên hệ đệ quy và hình 5.26 là một biểu đồ đối tượng cho biểu đồ lớp trong hình 5.25.

6- Quan hệ kết tập (Aggregation)

6.1- Khái niệm kết tập:

Kết tập là một trường hợp đặc biệt của liên hệ. Kết tập biểu thị rằng quan hệ giữa các lớp dựa trên nền tảng của nguyên tắc "một tổng thể được tạo thành bởi các bộ phận". Nó được sử dụng khi chúng ta muốn tạo nên một thực thể mới bằng cách tập hợp các thực thể tồn tại với nhau. Một ví dụ tiêu biểu của kết tập là chiếc xe ô tô gồm có bốn bánh xe, một động cơ, một khung gầm, một hộp số, v.v....

Quá trình ghép các bộ phận lại với nhau để tạo nên thực thể cần thiết được gọi là sự kết tập. Trong quá trình tìm lớp, kết tập sẽ được chú ý tới khi gặp các loại động từ "được tạo bởi", "gồm có", Quan hệ kết tập không có tên riêng. Tên ngầm chứa trong nó là "bao gồm các thành phần".

6.2- Kí hiệu kết tập:

Kí hiệu UML cho kết tập là đường thẳng với hình thoi (diamond) đặt sát lớp biểu thị sự kết tập (tổng thể).

Một lớp tài khoản được tạo bởi các lớp chi tiết về khách hàng, các lệnh giao dịch đối với tài khoản cũng như các quy định của nhà băng.

Quan hệ trên có thể được trình bày như sau:

Mỗi thành phần tạo nên kết tập (tổng thể) được gọi là một bộ phận (aggregates). Mỗi bộ phận về phần nó lại có thể được tạo bởi các bộ phận khác.

Trong trường hợp tài khoản kể trên, một trong các bộ phận của nó là các chi tiết về khách hàng. Các chi tiết về khách hàng lại bao gồm danh sách chủ tài khoản, danh sách địa chỉ, các quy định về kỳ hạn cũng như các chi tiết khác khi mở tài khoản.



6.3- Kết tập và liên hệ:

Khái niệm kết tập nảy sinh trong tình huống một thực thể bao gồm nhiều thành phần khác nhau. Liên hệ giữa các lớp mặt khác là mối quan hệ giữa các thực thể.

Quan sát hình sau:

Một tài khoản được tạo bởi các chi tiết về khách hàng, các lệnh giao dịch đối với tài khoản cũng như các quy định của nhà băng. Khách hàng không phải là bộ phận của tài khoản, nhưng có quan hệ với tài khoản.

Nhìn chung, nếu các lớp được nối kết với nhau một cách chặt chẽ qua quan hệ "toàn thể – bộ phận" thì người ta có thể coi quan hệ là kết tập. Không có lời hướng dẫn chắc chắn và rõ ràng cho việc bao giờ nên dùng kết tập và bao giờ nên dùng liên hệ. Một lối tiệm cận nhất quán đi kèm với những kiến thức sâu sắc về phạm vi vấn đề sẽ giúp nhà phân tích chọn giải pháp đúng đắn.

7- Khái quát hóa và chuyên biệt hóa (Generalization & Specialization)

Hãy quan sát cấu trúc lớp trong biểu đồ sau:

Trong hình trên, tài khoản là khái niệm chung của các loại tài khoản khác nhau và chứa những đặc tả cần thiết cho tất cả các loại tài khoản. Ví dụ như nó có thể chứa số tài khoản và tên chủ tài khoản. Ta có thể có hai loại tài khoản đặc biệt suy ra từ dạng tài khoản chung này, một loại mang tính kỳ hạn và một loại mang tính giao dịch. Yếu tố chia cách hai lớp này với nhau là các quy định chuyên ngành hay đúng hơn là phương thức hoạt động của hai loại tài khoản.

Tương tự như vậy, tài khoản đầu tư trung hạn và dài hạn lại là những khái niệm chuyên biệt của khái niệm tài khoản có kỳ hạn. Mặt khác, tài khoản bình thường và tài khoản tiết kiệm là những trường hợp đặc biệt của loại tài khoản giao dịch.

Loại cấu trúc lớp như thế được gọi là một cấu trúc hình cây hoặc cấu trúc phân cấp. Khi chúng ta dịch chuyển từ điểm xuất phát của cây xuống dưới, chúng ta sẽ gặp các khái niệm càng ngày càng được chuyên biệt hóa nhiều hơn. Theo con đường đi từ tài khoản đến tài khoản tiết kiệm, ta sẽ phải đi qua lớp tài khoản giao dịch. Lớp này tiếp tục phân loại các lớp chuyên biệt hóa cao hơn, tùy thuộc vào chức năng của chúng.

◆ 7.1- Kí hiệu khái quát hóa và chuyên biệt hóa

Trong biểu đồ trên, các lớp trong một cấu trúc cây được nối với nhau bằng một mũi tên rỗng, chỉ từ lớp chuyên biệt hơn tới lớp khái quát hơn.

Quá trình bắt đầu với một lớp khái quát để sản xuất ra các lớp mang tính chuyên biệt cao hơn được gọi là quá trình **chuyên biệt hoá (Specialization)**

Chuyên biệt hóa: là quá trình tinh chế một lớp thành những lớp chuyên biệt hơn. Chuyên biệt hóa bổ sung thêm chi tiết và đặc tả cho lớp kết quả. Lớp mang tính khái quát được gọi là **lớp cha (superclass)**, kết quả chuyên biệt hóa là việc tạo ra các **lớp con (Subclass)**.

Mặt khác, nếu chúng ta đi dọc cấu trúc cây từ dưới lên, ta sẽ gặp các lớp ngày càng mang tính khái quát cao hơn - Ví dụ từ lớp tài khoản tiết kiệm lên tới lớp tài khoản. Con đường bắt đầu từ một lớp chuyên biệt và khiến nó ngày càng mang tính khái quát cao hơn được gọi là quá trình **khái quát hóa (Generalization)**. Lớp chuyên biệt ở đây được gọi là lớp con, trong ví dụ trên là tài khoản tiết kiệm, trong khi lớp khái quát kết quả được gọi là lớp cha.

Chuyên biệt hóa và khái quát hóa là hai con đường khác nhau để xem xét cùng một mối quan hệ.

Một lớp là lớp con của một lớp này có thể đóng vai trò là một lớp cha của lớp khác.

◆ 7.2- Yếu tố phân biệt (Discriminator)

Để tạo một cấu trúc phân cấp, cần phải có một số thuộc tính làm nền tảng cho quá trình chuyên biệt hóa. Thuộc tính đó được gọi là **yếu tố phân biệt** (Discriminator).

Với mỗi giá trị có thể gán cho yếu tố phân biệt trong lớp cha, ta sẽ có một lớp con tương ứng.

Trong hình trên, yếu tố phân biệt trong lớp tài khoản là "loại tài khoản". Chúng ta giả thiết rằng chỉ có hai loại tài khoản, một mang tính kỳ hạn và một mang tính giao dịch. Theo đó, ta phải tạo ra hai lớp con, một cho các tài khoản mang tính kỳ hạn và một cho các tài khoản mang tính giao dịch.

Trong mô hình đối tượng, không nhất thiết phải nêu bật yếu tố phân biệt. Yếu tố phân biệt luôn có mặt trong một cấu trúc phân cấp lớp cha/ con, dù có được nhấn mạnh trong mô hình đối tượng hay không. Mặc dầu vậy, để đảm bảo cho một mô hình được định nghĩa rõ ràng, trình bày yếu tố phân biệt vẫn luôn là công việc nên thực hiện.

7.2.1- Lớp trừu tượng

Quan sát cấu trúc trong hình trên, ta thấy lớp tài khoản sẽ không bao giờ được thực thể hóa, có nghĩa là hệ thống sẽ không bao giờ tạo ra các đối tượng thuộc lớp này. Nguyên nhân là vì lớp tài khoản mang tính khái quát cao đến mức độ việc khởi tạo lớp này sẽ không có một ý nghĩa nào đáng kể. Lớp tài khoản mặc dù vậy vẫn đóng một vai trò quan trọng trong việc khái quát hóa các thuộc tính sẽ được cần đến trong các lớp dẫn xuất từ nó. Những loại lớp như thế được dùng để cung cấp một cây cấu trúc lớp và không có sự tồn tại đầy đủ ý nghĩa trong một mô hình thật sự ngoài đời, chúng được gọi là **lớp trừu tượng (abstract class)**.

7.2.2- Tạo lớp trừu tượng

Các lớp trừu tượng là kết quả của quá trình khái quát hóa. Hãy quan sát ví dụ cấu trúc lớp sau đây. Lớp tài khoản đứng đầu cây cấu trúc và được gọi là lớp căn bản. Lớp căn bản của một cây cấu trúc chứa những thuộc tính đã được khái quát hóa và có thể được áp dụng cho mọi lớp dẫn xuất từ nó. Trong quá trình khái quát hóa, các thuộc tính được dùng chung trong các lớp chuyên biệt được đưa lên lớp cha. Lớp cha về cuối được tạo bởi các thuộc tính chung của tất cả các lớp dẫn xuất từ nó. Những lớp cha dạng như vậy trong rất nhiều trường hợp sẽ mang tính khái quát tuyệt đối và sẽ không theo đuổi mục đích khởi tạo, chúng có lối ứng xử giống như một thùng chứa (container) cho tất cả các thuộc tính chung của các lớp dẫn xuất. Những lớp như thế trong trường hợp chung thường là kết quả ánh

xạ của những danh từ trừu tượng, là hệ quả của phương pháp sử dụng các danh từ để nhận diện lớp.

Biểu đồ trên cho ta một ví dụ về khái quát hóa và các thuộc tính chung, nó chỉ ra nhiều lớp chuyên biệt. Chú ý rằng cứ theo mỗi mức chuyên biệt hóa lại có thêm các thuộc tính được bổ sung thêm cho các lớp, khiến chúng mang tính chuyên biệt cao hơn so với các lớp cha ở mức trừu tượng bên trên. Ví dụ lớp tài khoản có thuộc tính là số tài khoản và tên khách hàng. Đây là những thuộc tính hết sức chung chung. Tất cả các lớp dẫn xuất từ nó, dù là trực tiếp hay gián tiếp (ở các mức độ trừu tượng thấp hơn nữa), đều có quyền sử dụng các thuộc tính đó của lớp tài khoản. Các lớp tài khoản có kỳ hạn và tài khoản giao dịch là hai lớp chuyên biệt dẫn xuất từ lớp tài khoản. Chúng có những thuộc tính chuyên biệt riêng của chúng - ví dụ mức thời gian (duration) đối với lớp tài khoản có kỳ hạn và mức tiền tối thiểu đối với lớp tài khoản giao dịch – bên cạnh hai thuộc tính số tài khoản và tên khách hàng mà chúng thừa kế từ lớp tài khoản. Cũng tương tự như thế với tài khoản đầu tư ngắn hạn và tài khoản đầu tư trung hạn là các loại lớp thuộc tài khoản có kỳ hạn, tài khoản tiết kiệm và tài khoản bình thường là các loại lớp thuộc lớp tài khoản giao dịch.

7.2.3- Lớp cụ thể (concrete class)

Lớp cụ thể là những lớp có thể thực thể hóa. Như đã nói từ trước, các lớp cụ thể khi thực thể hóa được gọi là các đối tượng. Trong ví dụ trên, các lớp tài khoản đầu tư ngắn hạn và tài khoản đầu tư dài hạn có thể được thực thể hóa thành đối tượng. Tương tự đối với tài khoản tiết kiệm và tài khoản bình thường.

7.2.4- Tổng kết về phát triển cây cấu trúc

Cơ chế dùng chung thuộc tính và thủ tục sử dụng nguyên tắc khái quát hóa được gọi là **tính thừa kế (inheritance)**. Sử dụng tính thừa kế để tinh chế (refine) các lớp sẽ dẫn tới việc phát triển một cây cấu trúc. Nên phát hiện những ứng xử (behaviour) chung trong một loạt lớp rồi thể hiện nó thành một lớp cha. Sự khác biệt trong ứng xử của cùng một lớp sẽ dẫn tới việc tạo ra các lớp con.

Khi phát triển cây cấu trúc, hãy quan sát ứng xử của các lớp. Trong trường hợp có một liên hệ tồn tại từ một lớp cụ thể đến tất cả các lớp con của một lớp cha, nên dịch chuyển liên hệ này lên lớp cha.

Nếu tồn tại một liên hệ giữa một lớp nào đó và một lớp cha, hãy chuyên biệt hóa và nâng cao cấu trúc để xác định xem liệu liên hệ này có được áp dụng cho tất cả

các lớp con của lớp cha nó hay không. Nếu có thì gán nó vào lớp cha, nếu không thì dịch xuống cho những lớp con phù hợp.

Trong khi tiến hành khái quát hóa, trọng tâm công việc là xác định các ứng xử chung trong một nhóm nhiều lớp chuyên biệt bậc trung. Khi đã xây dựng được một thủ tục hoặc một thuộc tính chung, nên kiểm tra lại xem chúng có thật sự là yếu tố chung của tất cả các lớp chuyên biệt trong phạm vi này. Khái quát hóa được áp dụng chỉ khi chúng ta có một tập hợp các lớp định nghĩa một loại đối tượng riêng biệt và có một số lượng lớn các ứng xử chung. Trọng tâm ở đây là tạo nên lớp cha chứa các ứng xử chung đó.

Khi chuyên biệt hóa, ta đi tìm các sự khác biệt trong ứng xử để tạo các lớp con thích ứng. Có nghĩa là ta xem xét một lớp tồn tại, kiểm tra xem có phải tất cả các ứng xử của nó đều có khả năng áp dụng cho mọi đối tượng. Nếu không, ta lọc ra ứng xử không phải lúc nào cũng cần thiết và chia trường hợp nó ra thành các lớp con. Trọng tâm của chuyên biệt hóa là tạo các lớp con.

Với cơ chế thừa kế, một lớp con sẽ kế thừa mọi thuộc tính à thủ tục của tất cả các lớp cha của nó.

Hình sau làm rõ việc tạo cấu trúc lớp sử dụng tính khái quát.

Thường xảy ra trường hợp tất cả các lớp con cùng tham gia vào một liên hệ hoặc kết tập. Trong trường hợp này nên tạo lớp cha định nghĩa liên hệ /kết tập đó. Hình sau giải thích thêm điểm này:

8- Quan hệ phụ thuộc và nâng cấp (Dependency & Refinement)

Bên cạnh liên hệ và khái quát hóa, UML còn định nghĩa hai loại quan hệ khác. **Quan hệ phụ thuộc (Dependency)** là một sự liên quan ngữ nghĩa giữa hai phần tử mô hình, một mang tính độc lập và một mang tính phụ thuộc. Mọi sự thay đổi trong phần tử độc lập sẽ ảnh hưởng đến phần tử phụ thuộc. Phần tử mô hình ở đây có thể là một lớp, một gói (package), một trường hợp sử dụng, .v.v... Có thể nêu một vài ví dụ cho sự phụ thuộc như: một lớp lấy tham số là đối tượng của một lớp khác, một lớp truy nhập một đối tượng toàn cục của một lớp khác, một lớp gọi một thủ tục thuộc một lớp khác. Trong tất cả các trường hợp trên đều có một sự phụ thuộc của một lớp này vào một lớp kia, mặc dù chúng không có liên hệ rõ ràng với nhau.

Quan hệ phụ thuộc được thể hiện bằng đường thẳng gạch rời (dashed line) với mũi tên (và có thể thêm một nhãn) giữa các phần tử mô hình. Nếu sử dụng nhãn thì nó sẽ là một khuôn mẫu (stereotype), xác định loại phụ thuộc. Hình sau chỉ ra một sự phụ thuộc dạng "friend", có nghĩa rằng một phần tử mô hình nhận được quyền truy cập đặc biệt tới cấu trúc nội bộ của phần tử thứ hai (thậm chí tới cả những phần mang tính nhìn thấy là private).

Nâng cấp (Refinement) là một quan hệ giữa hai lời miêu tả của cùng một sự vật, nhưng ở những mức độ trừu tượng hóa khác nhau. Nâng cấp có thể là mối quan hệ giữa một loại đối tượng và lớp thực hiện nó. Các nâng cấp thường gặp khác là quan hệ giữa một lớp phân tích (trong mô hình phân tích) và một lớp thiết kế (trong mô hình thiết kế) đều mô hình hóa cùng một thứ, quan hệ giữa một lời miêu tả có mức trừu tượng hóa cao và một lời miêu tả có mức trừu tượng hóa thấp (ví dụ một bức tranh khái quát của một sự cộng tác động và một biểu đồ chi tiết của cùng cộng tác đó). Quan hệ nâng cấp còn được sử dụng để mô hình hóa nhiều mức thực thi của cùng một thứ (một thực thi đơn giản và một thực thi phức tạp hơn, hiệu quả hơn).

Quan hệ nâng cấp được thể hiện bằng đường thẳng gạch rời (dashed line) với mũi tên rỗng.

Quan hệ nâng cấp được sử dụng trong việc phối hợp mô hình. Trong các dự án lớn, mọi mô hình đều cần phải được phối hợp với nhau. Phối hợp mô hình được sử dụng nhằm mục đích:

- Chỉ ra mối liên quan giữa các mô hình ở nhiều mức độ trừu tượng khác nhau.

- Chỉ ra mối liên quan giữa các mô hình ở nhiều giai đoạn khác nhau (phân tích yêu cầu, phân tích, thiết kế, thực thi,...).

- Hỗ trợ việc quản trị cấu hình.

- Hỗ trợ việc theo dõi trong mô hình.

9- Nâng cấp mô hình qua các vòng lặp kế tiếp

Cho tới thời điểm này, chúng ta đi qua các bước công việc phân tích căn bản và tạo nên phiên bản đầu tiên của mô hình đối tượng. Mô hình này cần phải được lấy làm mục tiêu cho các vòng lặp nâng cấp tiếp theo.

Công việc nâng cấp có thể được thực hiện bằng cách đưa mô hình qua tất cả các giai đoạn phát triển mô hình đối tượng một lần nữa. Lần này, những kiến thức thu được trong vòng phát triển đầu sẽ tỏ ra rất hữu dụng. Khi nâng cấp mô hình cần chú ý đến các bước sau:

- a) Nghiên cứu các lớp để tìm các thuộc tính và thủ tục không đồng dạng (dissimilar). Nếu có, xẻ lớp thành các thành phần để tạo tính đồng nhất (harmony) trong lớp. Ví dụ với một lớp đảm nhận hai vai trò khác nhau, hãy xẻ lớp thành các lớp kết quả với những thủ tục được xác định rõ ràng.
- b) Nếu phát hiện thấy một chức năng không hướng tới một lớp đích nào thì đó là triệu chứng thiếu lớp. Hãy bổ sung lớp thiếu và đưa thủ tục kể trên vào lớp đó.
- c) Khái quát hóa là còn chưa đủ độ nếu có các liên hệ trùng lặp (nhiều liên hệ cùng định nghĩa một quan hệ). Trong trường hợp này, cần tạo lớp cha để kết hợp các mối liên hệ đó.
- d) Nếu một vai trò mang một ý nghĩa đặc biệt quan trọng đối với hệ thống thì thường nó cần một lớp riêng. Một lựa chọn khác là biến liên hệ định nghĩa vai trò này thành một lớp liên hệ.
- e) Nếu một lớp thiếu cả thuộc tính lẫn thủ tục và / hoặc liên hệ thì rất có thể đây là một lớp không cần thiết. Hãy loại bỏ những lớp đó nếu có thể.
- f) Hãy rà soát toàn bộ hệ thống để tìm những vai trò giữa các lớp còn chưa được thể hiện. Nếu có, đây là triệu chứng thiếu liên hệ.
- g) Nếu có một liên hệ giữa các đối tượng nhưng lại chẳng được thủ tục nào sử dụng tới thì rất có thể đây là một liên hệ không cần thiết. Ví dụ ta đã xác định một liên hệ giữa nhân viên thu ngân và khách hàng nhưng lại không có thủ tục nào được định nghĩa giữa hai người. Trong trường hợp này, rất có thể liên hệ đó là không cần thiết.

Một số mách bảo thực tế:

Nghiên cứu để hiểu thấu đáo vấn đề cần giải quyết:

Khi xây dựng mô hình đối tượng, không nên bắt đầu bằng cách viết ra các cấu trúc lớp, các mối liên hệ cũng như những mối quan hệ thừa kế lộ rõ trên bề mặt và đập thẳng vào mắt chúng ta. Hãy dành thời gian nghiên cứu kỹ bản chất vấn đề. Mô hình đối tượng phải được thiết kế để phù hợp với giải pháp cho vấn đề mà chúng ta nhắm tới.

Cẩn thận khi chọn tên:

Tên cần được chọn một cách cẩn thận bởi nó chứng nhận sự tồn tại các thực thể. Tên cần phải chính xác, ngắn gọn, tránh gây bần cãi. Tên phải thể hiện tổng thể đối tượng chứ không chỉ nhằm tới một khía cạnh nào đó của đối tượng.

Bất cứ nơi nào có thể, hãy chọn những tên nào bao chứa các danh từ chuyên ngành quen thuộc đối với người sử dụng. Những tên tạo ra những hình xa vời đối với người sử dụng, hoặc các thực thể được đặt tên một cách tồi tệ rất dễ gây ra nhầm lẫn.

Cần giữ cho mô hình đối tượng được đơn giản:

Hãy kháng cự lại xu hướng tạo ra các mô hình phức tạp, chúng chỉ mang lại sự nhầm lẫn, bối rối. Trong vòng đầu của quy trình mô hình hóa đối tượng, hãy xác định các mối liên hệ căn bản và gạt ra ngoài các chi tiết, việc xem xét tới các số lượng thành phần tham gia (Cardinality) trong quan hệ được để dành cho giai đoạn sau; rất có thể là ở vòng thứ hai. Tốt nhất là các chi tiết phản ánh số lượng các thành phần tham gia trong quan hệ chỉ được bổ sung thêm vào trong vòng thứ hai hoặc vòng thứ ba của công việc mô hình hóa đối tượng. Thường thường, người ta thấy những phiên bản đầu tiên của mô hình thường chỉ chứa các mối liên hệ với số lượng là từ 0-tới-0; 0-tới-1, 1- tới-1; 1-tới-nhiều.

Nên sử dụng các mối liên hệ hạn định bất cứ khi nào có thể.

Tránh khái quát hóa quá nhiều. Thường chỉ nên hạn chế ở ba tầng khái quát.

Hãy nghiên cứu thật kỹ các mối liên hệ 1-tới-nhiều. Chúng thường có thể được chuyển thành các quan hệ 1-tới-0 hoặc 1-tới-1.

Tất cả các mô hình cần phải được lấy làm đối tượng cho việc tiếp tục nâng cấp. Nếu không thực hiện những vòng nâng cấp sau đó, rất có thể mô hình của chúng ta sẽ thiếu hoàn chỉnh.

Động tác để cho những người khác xem xét lại mô hình là rất quan trọng. Thường sự liên quan quá cận kề với mô hình sẽ khiến chúng ta mù lòa, không nhận những ra khiếm khuyết của nó. Một cái nhìn vô tư trong trường hợp này là rất cần thiết.

Không nên mô hình hóa các mối liên hệ thành thuộc tính. Nếu điều này xảy ra, ta thường có thể nhận thấy qua triệu chứng là mô hình thiếu liên hệ. Thêm vào đó, đã có lúc ta bỏ qua sự cần thiết của một yếu tố hạn định.

Việc viết tài liệu cho mô hình là vô cùng quan trọng. Các tài liệu cần phải nắm bắt thấu đáo những nguyên nhân nằm đằng sau mô hình và trình bày chúng chính xác như có thể.

10- Chất lượng mô hình

Làm sao để biết được mô hình là tốt hay chưa tốt? Một ngôn ngữ mô hình hóa có thể cung cấp ngữ pháp và ngữ nghĩa cho ta làm việc, nhưng nó không cho ta biết liệu một mô hình vừa được tạo dựng nên là tốt hay không. Yếu tố này mở ra một vấn đề quan trọng trong việc xác định chất lượng mô hình. Điều chủ chốt khi chúng ta thiết kế mô hình là thứ chúng ta muốn nói về hiện thực. Mô hình mang lại sự diễn giải cho những gì mà chúng ta nghiên cứu (hiện thực, một viễn cảnh...).

Trong một mô hình, yếu tố quan trọng bậc nhất là phải nắm bắt được bản chất của vấn đề. Trong một hệ thống tài chính, chúng ta thường mô hình hóa các hóa đơn chứ không phải các món nợ. Trong đa phần doanh nghiệp, bản thân hóa đơn không thật sự có tầm quan trọng đến như vậy, yếu tố quan trọng ở đây là các món nợ. Một hóa đơn chỉ là một sự thể hiện của một món nợ, nhưng ta cần phải mô hình hóa làm sao để phản ánh điều đó. Một khái niệm khác là một tài khoản ở nhà băng. Trong những năm 70 và 80 đã có rất nhiều mô hình thể hiện tài khoản nhà băng. Khách hàng (chủ nhân của tài khoản tại nhà băng) được coi là một thành phần của tài khoản này (một tài khoản nhà băng được mô hình hóa như là một lớp hoặc là một thực thể và một khách hàng là một thuộc tính). Khó khăn đầu tiên xảy ra là nhà băng không thể xử lý tài khoản có nhiều chủ. Vấn đề thứ hai là nhà băng không thể tạo ra các chiến lược marketing nhắm tới những khách hàng không có tài khoản trong nhà băng chỉ bởi vì họ không có địa chỉ.

Vì vậy, một trong những khía cạnh của chất lượng mô hình là tính thích hợp của mô hình đó. Một mô hình thích hợp phải nắm bắt các khía cạnh quan trọng của đối tượng nghiên cứu. Những khía cạnh khác trong việc đánh giá chất lượng là mô hình phải dễ giao tiếp, phải có một mục tiêu cụ thể, dễ bảo quản, mang tính vững bền và có khả năng tích hợp. Nhiều mô hình của cùng một hệ thống nhưng có các mục đích khác nhau (hoặc là hướng nhìn khác nhau) phải có khả năng tích hợp được với nhau.

Dù là sử dụng phương pháp nào hoặc ngôn ngữ mô hình hóa nào, ta vẫn còn phải đối mặt với các vấn đề khác. Khi tạo dựng mô hình, chúng ta trở thành một phần của doanh nghiệp, có nghĩa là chúng ta cần phải quan sát hiệu ứng sự can thiệp của chúng ta vào doanh nghiệp. Yếu tố quan trọng là cần phải xử lý tất cả các khía cạnh của sự can thiệp đó ví dụ như về chính sách, văn hóa, cấu trúc xã hội và năng suất. Nếu không làm được điều này, rất có thể ta không có khả năng

phát hiện và nắm bắt tất cả những đòi hỏi cần thiết từ phía khách hàng (cần chú ý rằng những phát biểu yêu cầu được đưa ra không phải bao giờ cũng chính xác là những gì khách hàng thực sự cần). Hãy đặc biệt chú ý đến các vấn đề với chính sách nội bộ, các mẫu hình xã hội, các cấu trúc không chính thức và các thể lực bao quanh khách hàng.

◆ 10.1- Thế nào là một mô hình tốt?

Một mô hình sẽ là một mô hình tốt nếu ta có khả năng giao tiếp với nó, nếu nó phù hợp với các mục đích của nó và nếu chúng ta đã nắm bắt được những điểm cốt yếu của vấn đề. Một mô hình tốt đòi hỏi thời gian xây dựng; bình thường ra nó được tạo bởi một nhóm phát triển, được thành lập với một mục đích cụ thể. Một trong những mục đích này có thể là huy động toàn bộ lực lượng để phát hiện ra các yêu cầu của một cơ quan. Một mục đích khác rất có thể là mô hình hóa một đặc tả yêu cầu, thực hiện một giai đoạn phân tích, hay vẽ một bản thiết kế kỹ thuật cho một hệ thống thông tin. Khi các cá nhân khác nhau được tập hợp thành nhóm, động tác này cần phải được thực hiện tập trung vào mục tiêu định trước. Các nhóm để mô hình hóa một doanh nghiệp hoặc là một hệ thống thông tin rất có thể được tạo bởi khách hàng, chuyên gia mô hình hóa và chuyên gia ứng dụng.

◆ 10.2- Ta có thể giao tiếp với mô hình?

Tại sao mô hình lại phải là thứ dễ giao tiếp? Tất cả các dự án, dù lớn hay nhỏ, đều cần phải được giao tiếp. Con người ta nói chuyện với nhau. Họ đọc các tài liệu của nhau và thảo luận các nội dung của chúng. Sáng kiến khởi thủy nằm đằng sau bất kỳ một mô hình nào cũng là để tạo ra khả năng giao tiếp với chúng. Nếu chúng ta tạo ra các mô hình mà không ai đọc nổi, hiểu nổi, thì đó là việc làm vô ý nghĩa. Mô hình chẳng phải được tạo ra bởi người dẫn đầu một phương pháp hoặc người dẫn đầu một dự án ra lệnh. Mô hình được tạo ra để phục vụ cho việc giao tiếp và tập hợp các cố gắng của chúng ta để đạt đến năng suất, hiệu quả và chất lượng cao như có thể.

◆ 10.3- Mô hình có phù hợp với mục đích của nó không?

Một mô hình hình cần phải có một mục đích rõ ràng, sao cho ai dùng nó cũng nhận được ra. Tất cả các mô hình đều có mục đích, nhưng thường mục đích này là ngầm ẩn, và điều này khiến cho việc sử dụng và hiểu nó trở nên khó khăn. Các mô hình phân tích và mô hình thiết kế có thể là mô hình của cùng một hệ thống, nhưng chúng vẫn là những mô hình khác nhau và tập trung vào các chủ đề khác nhau (hay là chi tiết khác nhau). Cần phải xác định rõ ràng mục đích cho mỗi mô hình để có thể kiểm tra và phê duyệt nó. Nếu không có mục đích rõ ràng,

chúng ta ví dụ rất có thể sẽ thẩm tra một mô hình hình phân tích như thế nó là một mô hình thiết kế.

◆ **10.4- Nắm bắt những điểm trọng yếu**

Nhiều mô hình chỉ bao gồm các tài liệu của doanh nghiệp – ví dụ như các hóa đơn, những thông tin nhận được, các hợp đồng bảo hiểm. Nếu mô hình chỉ là sự bao gồm các tài liệu thì điều gì sẽ xảy ra nếu doanh nghiệp thay đổi? Đây là một vấn đề rất quan trọng trong thực tế. Chúng ta cần thiết phải nắm bắt bản chất của doanh nghiệp (tạo nên phần nhân) và mô hình xoay quanh các khái niệm thiết yếu đó để có khả năng xử lý các thay đổi một cách thích hợp. Hãy mô hình hóa phần nhân của doanh nghiệp và sau đó mới đến một mô hình diễn giải phần nhân đó. Một khi phần nhân đã được mô hình hóa, những thay đổi nho nhỏ trong doanh nghiệp có thể được xử lý qua việc sửa đổi các lớp diễn giải các loại đối tượng thuộc phần nhân (ví dụ như các hóa đơn là một sự diễn giải của các món nợ).

◆ **10.5- Phối hợp các mô hình**

Các mô hình khác nhau của cùng một hệ thống phải có khả năng được kết hợp và liên quan đến nhau. Một trong các khía cạnh của phối hợp mô hình là sự tích hợp. Tích hợp có nghĩa là một nhóm các mô hình cùng chung mục đích và thể hiện cùng một thứ (mặc dù chúng có thể có nhiều hướng nhìn khác nhau, ví dụ như mô hình động, mô hình chức năng, mô hình tĩnh), thì chúng phải có khả năng được ráp lại với nhau mà không làm nảy sinh mâu thuẫn.

Quan hệ giữa các mô hình ở những mức độ trừu tượng khác nhau là một khía cạnh quan trọng khác. Nó là một trong những chìa khóa dẫn đến khả năng có thể theo dõi bước phát triển của các phần tử khác nhau, phục vụ cho công nghệ lập trình. Quan hệ giữa các mức độ trừu tượng khác nhau có thể được thể hiện bằng quan hệ nâng cấp trong UML. Điều đó có nghĩa là các mô hình sẽ được phối hợp tại mỗi một mức độ trừu tượng cũng như được phối hợp giữa các mức độ trừu tượng khác nhau.

◆ **10.6- Độ phức tạp của mô hình**

Ngay cả khi các mô hình của chúng ta dễ dàng giao tiếp, có một mục đích rõ ràng, nắm bắt được những điểm trọng yếu trong phạm vi vấn đề và có thể được phối hợp với nhau, ta vẫn có thể gặp khó khăn nếu mô hình quá phức tạp. Những mô hình cực kỳ phức tạp sẽ khó nghiên cứu, khó thẩm tra, khó phê duyệt và khó bảo trì. Sáng kiến tốt là hãy bắt đầu với một mô hình đơn giản, và sau đó chi tiết hóa nhiều hơn bằng cách sử dụng việc phối hợp mô hình. Nếu bản chất phạm vi vấn đề của chúng ta là phức tạp, hãy xẻ mô hình thành nhiều mô hình khác nhau

(sử dụng các tiểu mô hình – tức là các gói) và cố gắng để qui trình này có thể kiểm soát được tình huống.

11- Tóm tắt về mô hình đối tượng

Khi tạo mô hình là chúng ta diễn giải các chi tiết về những gì mà chúng ta nghiên cứu, thế nhưng một yếu tố rất quan trọng là mô hình phải nắm bắt được những điểm trọng yếu của đối tượng nghiên cứu. Một đối tượng là một thứ gì đó mà chúng ta có thể nói về và có thể xử lý trong một số phương thức nào đó. Một đối tượng tồn tại trong thế giới thực (hoặc nói cho chính xác hơn là trong sự hiểu biết của chúng ta về thế giới thực). Một đối tượng có thể là một thành phần của một hệ thống nào đó trong thế giới – một chiếc máy, một tổ chức, một doanh nghiệp. Một lớp là lời miêu tả từ 0, 1 hoặc nhiều đối tượng với cùng lối ứng xử. Lớp và đối tượng được sử dụng để bàn luận về các hệ thống.

Khi chúng ta mô hình hóa, chúng ta sử dụng một ngôn ngữ mô hình hóa ví dụ như UML, cung cấp cho chúng ta ngữ pháp và ngữ nghĩa để tạo dựng mô hình. Ngôn ngữ mô hình hóa mặc dù vậy không thể cho chúng ta biết liệu chúng ta đã tạo ra một mô hình tốt hay không. Chất lượng mô hình cần phải được chú ý riêng biệt, điều đó có nghĩa là tất cả các mô hình cần phải có một mục đích rõ ràng và chính xác và chúng phải nắm bắt được bản chất của đối tượng nghiên cứu. Tất cả các mô hình cần phải được làm sao để dễ giao tiếp, dễ thẩm tra, phê duyệt và bảo trì.

UML cung cấp mô hình tĩnh, động và theo chức năng. Mô hình tĩnh được thể hiện qua các biểu đồ lớp, bao gồm các lớp và mối quan hệ giữa chúng. Quan hệ có thể là liên hệ, khái quát hoá, phụ thuộc hoặc là nâng cấp. Một mối quan hệ liên hệ là một sự nối kết giữa các lớp, có nghĩa là sự nối kết giữa các đối tượng của các lớp này. Khái quát hoá là quan hệ giữa một phần tử mang tính khái quát hơn và một phần tử mang tính chuyên biệt hơn. Phần tử mang tính chuyên biệt hơn có thể chỉ chứa các thông tin bổ sung. Một thực thể (một đối tượng là một thực thể của một lớp) của phần tử chuyên biệt hơn có thể được sử dụng bất cứ nơi nào mà thực thể của phần tử khái quát hơn được cho phép. Phụ thuộc là mối quan hệ giữa hai phần tử, một mang tính độc lập và một mang tính phụ thuộc. Mỗi thay đổi trong phần tử độc lập sẽ gây tác động đến phần tử phụ thuộc. Một quan hệ nâng cấp là một quan hệ giữa hai lời miêu tả của cùng một thứ nhưng ở những mức độ trừu tượng khác nhau.

Phần câu hỏi

Hỏi: Khi tạo dựng mô hình, cần sử dụng các khái niệm của chính phạm vi vấn đề để mô hình dễ hiểu và dễ giao tiếp.

Đáp: Đúng

Hỏi: Các lớp chỉ thể hiện cấu trúc thông tin?

Đáp: sai, các lớp không phải chỉ thể hiện cấu trúc thông tin mà còn mô tả cả hành vi.

Hỏi: Các khái niệm then chốt thường sẽ trở thành các lớp trong mô hình phân tích?

Đáp: Đúng

Hỏi: Thường các danh từ trong các lời phát biểu bài toán sẽ là ứng cử viên để chuyển thành lớp và đối tượng?

Đáp: Đúng

Hỏi: Quan hệ kết hợp (Association) giữa các lớp định nghĩa các mối liên quan có thể tồn tại giữa các đối tượng?

Đáp: Đúng, ví dụ một mối quan hệ kết hợp là một sự nối kết giữa các lớp, có nghĩa là sự nối kết giữa các đối tượng của các lớp này.

Hỏi: Kết tập biểu thị rằng quan hệ giữa các lớp dựa trên nền tảng của nguyên tắc "một tổng thể được tạo thành bởi các bộ phận"

Đáp: Đúng, nó được sử dụng khi chúng ta muốn tạo nên một thực thể mới bằng cách tập hợp các thực thể tồn tại với nhau

Hỏi: Khái quát hoá được sử dụng để tạo các lớp con?

Đáp: Sai, khái quát hoá là quá trình bắt đầu từ một lớp chuyên biệt và khiến nó ngày càng mang tính khái quát cao hơn (lớp cha)

Hỏi: Chuyên biệt hoá bổ sung thêm chi tiết và đặc tả cho lớp kết quả?

Đáp: Đúng, chuyên biệt hoá là quá trình tinh chế một lớp thành những lớp chuyên biệt hơn (lớp con)

☐ ☐ ☐

Chương 6: MÔ HÌNH ĐỘNG

1- SỰ CẦN THIẾT CÓ MÔ HÌNH ĐỘNG (DYNAMIC MODEL)

Mô hình đối tượng và quá trình phát triển nó là trọng tâm của những cuộc thảo luận trong chương trước. Mô hình đối tượng định nghĩa hệ thống theo khái niệm các thành phần tĩnh. Mô hình đối tượng miêu tả ứng xử mang tính cấu trúc và chức năng của các lớp. Mặc dầu vậy, để mô hình hóa sự hoạt động thật sự của một hệ thống và trình bày một hướng nhìn đối với hệ thống trong thời gian hệ thống hoạt động, chúng ta cần tới **mô hình động (dynamic model)**.

Trong UML, mô hình động đề cập tới các trạng thái khác nhau trong vòng đời của một đối tượng thuộc hệ thống. Phương thức ứng xử của một hệ thống tại một thời điểm cụ thể sẽ được miêu tả bằng các điều kiện khác nhau ấn định cho sự hoạt động của nó.

Một yếu tố hết sức quan trọng là cần phải hiểu cho được hệ thống sẽ đáp lại những kích thích từ phía bên ngoài ra sao, có nghĩa là chúng ta cần phải xác định và nghiên cứu chuỗi các thủ tục sẽ là hệ quả của một sự kích thích từ ngoài. Cho việc này, ta cần tới mô hình động bởi trọng tâm của mô hình này là lối ứng xử phụ thuộc vào thời gian của các đối tượng trong hệ thống.

Chúng ta cần tới mô hình động bởi chúng ta cần thể hiện sự thay đổi xảy ra trong hệ thống dọc theo thời gian chạy. Công cụ miêu tả mô hình động là không thể thiếu ví dụ trong trường hợp các đối tượng trải qua nhiều giai đoạn khác nhau trong thời gian hệ thống hoạt động. Điều đó có nghĩa là mặc dù đối tượng được tạo ra một lần, nhưng các thuộc tính của chúng chỉ dần dần từng bước nhận được giá trị. Ví dụ như một tài khoản đầu tư có kỳ hạn được tạo ra, nhưng tổng số tiền lãi cộng dồn của nó chỉ được tăng lên dần dần theo thời gian.

Các mô hình động cũng là yếu tố hết sức cần thiết để miêu tả ứng xử của một đối tượng khi đưa ra các yêu cầu hoặc thực thi các tác vụ. Cả tác vụ lẫn dịch vụ, theo định nghĩa, đều là các hoạt động động và vì thế mà chỉ có thể được biểu diễn qua một mô hình động.

2- CÁC THÀNH PHẦN CỦA MÔ HÌNH ĐỘNG

Đối tượng trong các hệ thống giao tiếp với nhau, chúng gửi thông điệp (message) đến nhau. Ví dụ một đối tượng khách hàng là John gửi một thông điệp mua hàng đến người bán hàng là Bill để làm một việc gì đó. Một thông điệp thường là một lệnh gọi thủ tục mà một đối tượng này gọi qua một đối tượng kia. Các đối tượng giao tiếp với nhau ra sao và hiệu ứng của sự giao tiếp như thế được gọi là **khía cạnh động** của một hệ thống, ý nghĩa của khái niệm này là câu hỏi: các đối tượng cộng tác với nhau qua giao tiếp như thế nào và các đối tượng trong một hệ

thống thay đổi trạng thái ra sao trong thời gian hệ thống hoạt động. Sự giao tiếp trong một nhóm các đối tượng nhằm tạo ra một số các lệnh gọi hàm được gọi là **tương tác (interaction)**, tương tác có thể được thể hiện qua ba loại biểu đồ: biểu đồ tuần tự (sequence Diagram), biểu đồ cộng tác (collaboration Diagram) và biểu đồ hoạt động (activity Diagram).

Trong chương này, chúng ta sẽ đề cập tới bốn loại biểu đồ động của UML:

🔗 **Biểu đồ trạng thái:** miêu tả một đối tượng có thể có những trạng thái nào trong vòng đời của nó, ứng xử trong các trạng thái đó cũng như các sự kiện nào gây ra sự chuyển đổi trạng thái, ví dụ, một tờ hóa đơn có thể được trả tiền (trạng thái đã trả tiền) hoặc là chưa được trả tiền (trạng thái chưa trả tiền).

🔗 **Biểu đồ tuần tự:** miêu tả các đối tượng tương tác và giao tiếp với nhau ra sao. Tiêu điểm trong các biểu đồ tuần tự là thời gian. Các biểu đồ tuần tự chỉ ra chuỗi của các thông điệp được gửi và nhận giữa một nhóm các đối tượng, nhằm mục đích thực hiện một số chức năng.

🔗 **Biểu đồ cộng tác:** cũng miêu tả các đối tượng tương tác với nhau ra sao, nhưng trọng điểm trong một biểu đồ cộng tác là sự kiện. Tập trung vào sự kiện có nghĩa là chú ý đặc biệt đến mối quan hệ (nối kết) giữa các đối tượng, và vì thế mà phải thể hiện chúng một cách rõ ràng trong biểu đồ.

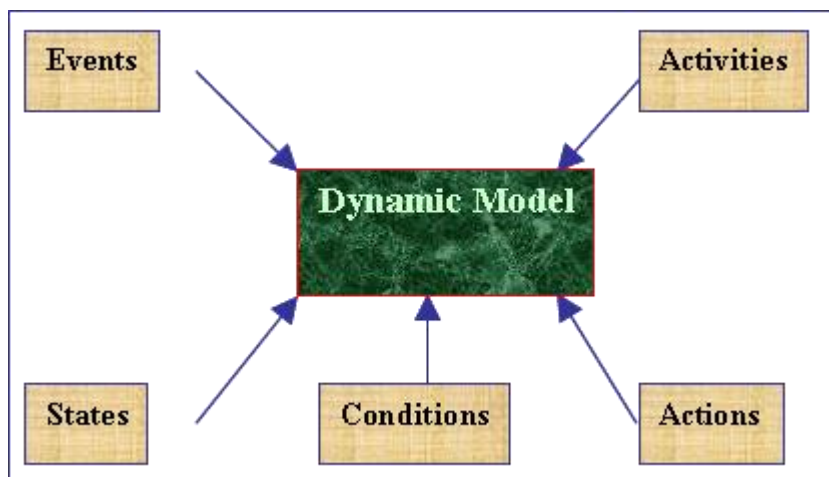
🔗 **Biểu đồ hoạt động:** là một con đường khác để chỉ ra tương tác, nhưng chúng tập trung vào công việc. Khi các đối tượng tương tác với nhau, các đối tượng cũng thực hiện các tác vụ, tức là các hoạt động. Những hoạt động này cùng thứ tự của chúng được miêu tả trong biểu đồ hoạt động.

Vì biểu đồ tuần tự, biểu đồ cộng tác lẫn biểu đồ hoạt động đều chỉ ra tương tác nên thường bạn sẽ phải chọn nên sử dụng biểu đồ nào khi lập tài liệu cho một tương tác. Quyết định của bạn sẽ phụ thuộc vào việc khía cạnh nào được coi là quan trọng nhất.

Ngoài cấu trúc tĩnh và ứng xử động, hướng nhìn chức năng cũng có thể được sử dụng để miêu tả hệ thống. Hướng nhìn chức năng thể hiện các chức năng mà hệ thống sẽ cung cấp. Trường hợp sử dụng chính là các lời miêu tả hệ thống theo chức năng; chúng miêu tả các tác nhân có thể sử dụng hệ thống ra sao. Như đã đề cập từ trước, trường hợp sử dụng bình thường ra được mô hình hóa trong những giai đoạn đầu tiên của quá trình phân tích, nhằm mục đích miêu tả xem tác nhân có thể muốn sử dụng hệ thống như thế nào. Mô hình trường hợp sử

dụng chỉ nên nắm bắt duy nhất khía cạnh tác nhân sử dụng hệ thống, không nên đề cập khía cạnh hệ thống được xây dựng bên trong ra sao. Lớp và các tương tác trong hệ thống thực hiện trường hợp sử dụng. Tương tác được miêu tả bởi các biểu đồ tuần tự, biểu đồ cộng tác và hoặc/và biểu đồ hoạt động, tức là có một sự nối kết giữa hướng nhìn chức năng và hướng nhìn động của hệ thống. Các lớp được sử dụng trong việc thực thi các trường hợp sử dụng được mô hình hóa và miêu tả qua các biểu đồ lớp và biểu đồ trạng thái (một biểu đồ trạng thái sẽ được đính kèm cho một lớp, một hệ thống con hoặc là một hệ thống). Trường hợp sử dụng và các mối quan hệ của chúng đến tương tác đã được miêu tả trong chương 3 (trường hợp sử dụng).

Nhìn chung, một mô hình động miêu tả năm khía cạnh căn bản khác nhau:



Hình 6.1- Các thành phần của mô hình động

Các thành phần kể trên sẽ được đề cập chi tiết hơn trong các phần sau.

Ngoài ra, một mô hình động cũng còn được sử dụng để xác định các nguyên tắc chuyên ngành (business rule) cần phải được áp dụng trong mô hình. Nó cũng được sử dụng để ấn định xem các nguyên tắc đó được đưa vào những vị trí nào trong mô hình.

Một vài ví dụ cho những nguyên tắc chuyên ngành cần phải được thể hiện trong mô hình động:

- Một khách hàng không được quyền rút tiền ra nếu không có đủ mức tiền trong tài khoản.
- Những món tiền đầu tư có kỳ hạn không thể chuyển sang một tên khác trước khi đáo hạn.
- Giới hạn cao nhất trong một lần rút tiền ra bằng thẻ ATM là 500 USD.

3- ƯU ĐIỂM CỦA MÔ HÌNH ĐỘNG:

Bất cứ khi nào có những ứng xử động cần phải được nghiên cứu hoặc thể hiện, chúng ta sẽ phải dùng đến mô hình động.

Mô hình động đóng một vai trò vô cùng quan trọng trong những trường hợp như:

- Các hệ thống mang tính tương tác cao
- Hệ thống có sử dụng các trang thiết bị ngoại vi có thể gọi nên các ứng xử của hệ thống.

Mô hình động không tỏ ra thật sự hữu hiệu trong trường hợp của các hệ thống tĩnh. Ví dụ một hệ thống chỉ nhằm mục đích nhập dữ liệu để lưu trữ vào một ngân hàng dữ liệu.

Một mô hình động tập trung vào các chuỗi tương tác (biểu đồ cộng tác) và vào yếu tố thời gian của các sự kiện (biểu đồ tuần tự). Một mô hình động có thể được sử dụng cho mục đích thể hiện rõ ràng theo thời gian hoạt động của hệ thống nếu trong thời gian này có những đối tượng:

- Được tạo ra
- Bị xóa đi
- Được lưu trữ
- Bị hủy

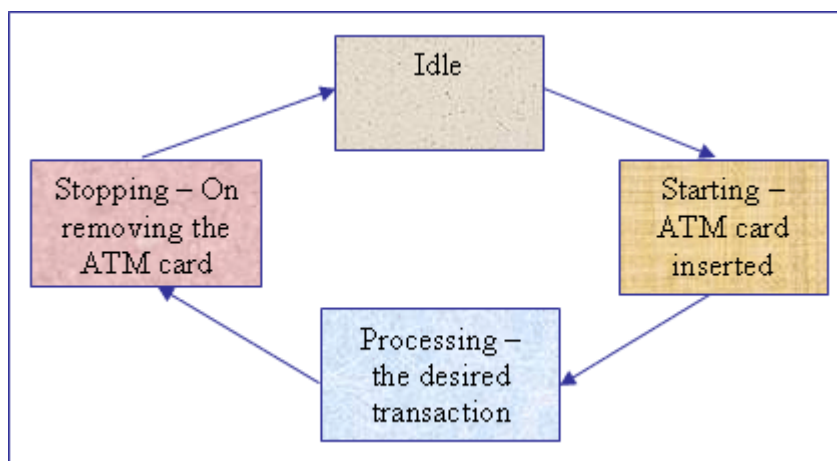
Hãy quan sát trường hợp rút tiền mặt và tương tác của khách hàng đối với nhà băng:

- Khách hàng điền tất cả các chi tiết cần thiết vào giấy yêu cầu rút tiền mặt.
- Khách hàng đưa giấy yêu cầu đó cho một nhân viên phát thẻ xếp hàng.
- Nhân viên phát thẻ ghi số của giấy yêu cầu rút tiền vào danh sách.
- Động tác ghi số của giấy yêu cầu rút tiền được thực hiện tuần tự, tương ứng với những số thẻ tuần tự được phát ra.
- Một tấm thẻ xếp hàng (token) được trao cho khách hàng.
- Khách hàng đi vào hàng xếp, chờ nhân viên bên casse gọi đúng số thẻ của mình.
- Song song với quá trình chờ của khách hàng, giấy yêu cầu rút tiền của anh ta trải qua nhiều giai đoạn trong nội bộ nhà băng.

- Chữ ký của khách hàng trên giấy yêu cầu rút tiền được thẩm tra.
- Giấy yêu cầu được xem xét về phương diện số tài khoản và mức tiền trong tài khoản.
- Nếu một trong hai điều kiện trên không được thỏa mãn, quá trình rút tiền mặt sẽ bị chặn lại hoặc là được sửa đổi và tiếp tục.
- Khi cả hai điều kiện nêu trên được thỏa mãn, giấy yêu cầu rút tiền mặt sẽ được đưa đến cho nhân viên ngồi bên casse, nơi khách hàng sẽ được gọi tới tuần tự dựa theo số thẻ xếp hàng.
- Nhân viên bên casse đưa tiền mặt cho khách hàng.

Lỗi ứng xử trong việc rút tiền mặt là mang tính động. Suốt quá trình rút tiền mặt, tương tác và trình tự của quá trình phụ thuộc vào một số các điều kiện xác định. Loại ứng xử này không thể được thể hiện qua mô hình đối tượng, đây là trường hợp ta cần đến mô hình động.

Mô hình động cũng tỏ ra hữu dụng trong trường hợp có những trang thiết bị trải qua tuần tự các bước trong một vòng lặp và tiến trình phụ thuộc vào một số điều kiện nhất định. Ví dụ một đối tượng mô hình hóa lỗi ứng xử của một máy rút tiền mặt tự động (ATM). Máy ATM lần lượt đi qua các bước của một vòng lặp mang tính thủ tục (chức năng), bắt đầu từ việc một thẻ ATM được đút vào trong máy, xử lý các yêu cầu do khách hàng đưa ra, dừng lại và chờ yêu cầu giao dịch khác, rồi sau đó quay trở lại trạng thái ban đầu (đứng yên) sau khi thẻ ATM đã được rút ra ngoài.



Hình 6.2- Mô hình động của máy rút tiền ATM

4- SỰ KIỆN VÀ THÔNG ĐIỆP (EVENT & MESSAGE)

4.1- Sự kiện (Event):

Một trong những thành phần quan trọng bậc nhất của một đối tượng là sự kiện. Một sự kiện là một sự kích thích được gửi từ đối tượng này sang đối tượng khác.

Một sự kiện là một việc sẽ xảy ra và có thể gây ra một hành động nào đó. Ví dụ như khi bạn bấm lên nút *Play* trên máy CD-Player, nó sẽ bắt đầu chơi nhạc (giả sử rằng CD-Player có điện, trong máy có đĩa CD và nói chung là dàn CD-Player hoạt động tốt). Sự kiện ở đây là bạn nhấn lên nút *Play*, và hành động ở đây là bắt đầu chơi nhạc. Nếu có một sự nối kết được định nghĩa rõ ràng giữa sự kiện và hành động, người ta gọi nó là **quan hệ nhân quả (Causality)**. Trong công nghệ phần mềm, chúng ta thường chỉ mô hình hóa các hệ thống mang tính nhân quả, nơi sự kiện và hành động được nối kết với nhau. Một phản ví dụ của quan hệ nhân quả: bạn lái xe trên xa lộ với tốc độ quá nhanh, cảnh sát ngăn xe lại. Đây không phải là nhân quả bởi hành động ngăn bạn lại của cảnh sát không chắc chắn bao giờ cũng xảy ra; vì thế mà không có một sự nối kết được định nghĩa rõ ràng giữa sự kiện (lái xe quá nhanh) và hành động (ngăn xe). Trong mô hình hóa, vậy là ta quan tâm đến sự kiện theo nghĩa là bất kỳ hành động nào khiến hệ thống phản ứng theo một cách nào đó.

Quan sát ví dụ một nhà băng lẻ, ta có một vài ví dụ về sự kiện như sau:

- Điền một tờ giấy yêu cầu rút tiền.
- Sự đáo hạn một tài khoản đầu tư có kỳ hạn.
- Kết thúc một hợp đồng trước kỳ hạn.
- Điền một giấy yêu cầu mở tài khoản.

UML biết đến tất cả bốn loại sự kiện:

- Một điều kiện trở thành được thỏa mãn (trở thành đúng)
- Nhận được một tín hiệu ngoại từ một đối tượng khác
- Nhận được một lời gọi thủ tục từ một đối tượng khác (hay từ chính đối tượng đó).
- Một khoảng thời gian xác định trước trôi qua.

Xin chú ý rằng cả các lỗi xảy ra cũng là sự kiện và có thể mang tính hữu dụng rất lớn đối với mô hình.

❗4.1.1- Sự kiện độc lập và sự kiện phụ thuộc

Các sự kiện có thể mang tính độc lập hay liên quan đến nhau. Có một số sự kiện, theo bản chất, phải đi trước hoặc là xảy ra sau các sự kiện khác. Ví dụ:

- Điền các chi tiết trong một tờ yêu cầu rút tiền mặt sẽ dẫn tới việc nhận được một số thẻ xếp hàng.
- Sự đáo hạn của một tài khoản đầu tư có kỳ hạn sẽ dẫn đến động tác gia hạn hoặc rút tiền mặt.
- Điền các chi tiết trong một giấy yêu cầu mở tài khoản sẽ dẫn tới việc phải nộp một khoản tiền tối thiểu (theo quy định) vào tài khoản.

Các sự kiện độc lập là những sự kiện không được nối kết với nhau trong bất kỳ một phương diện nào. Ví dụ:

- Rút tiền mặt và đưa tiền vào tài khoản là các sự kiện độc lập với nhau.
- Mở một tài khoản đầu tư có kỳ hạn và mở một tài khoản giao dịch là độc lập với nhau.
- Kết thúc trước kỳ hạn một tài khoản đầu tư và việc mở một tài khoản đầu tư có kỳ hạn khác là độc lập với nhau.

Các sự kiện độc lập còn có thể được gọi là các sự kiện song song hay đồng thời. Bởi chúng không phụ thuộc vào nhau, nên các sự kiện này có thể xảy ra tại cùng một thời điểm.

Trong nhiều trường hợp, một sự kiện riêng lẻ trong phạm vi vấn đề sẽ được chuyển tải thành nhiều sự kiện trong hệ thống. Ví dụ: đưa giấy yêu cầu rút tiền mặt cho nhân viên phát thẻ xếp hàng sẽ có kết quả là một loạt các sự kiện nối tiếp.

Có những tình huống nơi một sự kiện riêng lẻ sẽ được nhận bởi nhiều đối tượng khác nhau và khiến cho chúng phản ứng thích hợp. Ví dụ như một lời đề nghị ngân một tờ séc có thể đồng thời được gửi đến cho nhân viên thu ngân và nhân viên kiểm tra séc.

❗4.1.2-Sự kiện nội (internal) và sự kiện ngoại (external):

Sự kiện nội là các sự kiện xảy ra trong nội bộ hệ thống. Đây là các sự kiện do một đối tượng này gây ra đối với đối tượng khác. Ví dụ, tính toán tiền lãi cho một tài khoản đầu tư có kỳ hạn sẽ được nội bộ hệ thống thực hiện, tuân theo một đối tượng quan sát ngày tháng.

Sự kiện ngoại là những sự kiện được kích nên từ phía bên ngoài biên giới của hệ thống, ví dụ như sự kết thúc trước kỳ hạn một tài khoản đầu tư.

❗4.1.3- Sự kiện và lớp sự kiện:

Lớp sự kiện đối với sự kiện cũng như lớp đối với đối tượng bình thường. Lời định nghĩa xác định một loại sự kiện được gọi là một lớp sự kiện.

Lớp sự kiện ngoài ra còn có thể được phân loại:

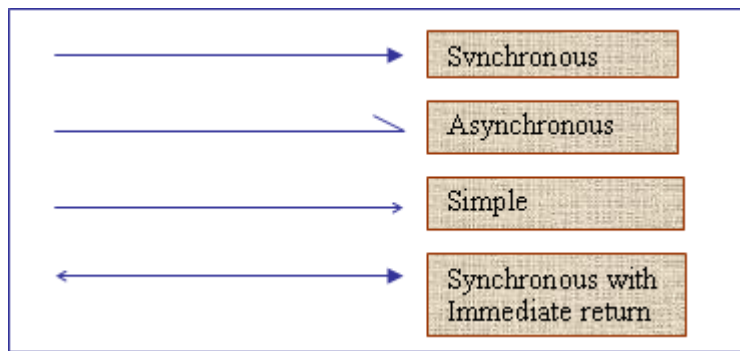
- *Các tín hiệu đơn giản*: Lớp sự kiện trong trường hợp này sẽ được thực thể hóa để chỉ ra một sự kiện hoặc là một tín hiệu của một sự kiện.
- *Các sự kiện chuyển tải dữ liệu*: thường thì một sự kiện có khả năng và chuyển tải dữ liệu. Tất cả các sự kiện cần phải "biết đến" các đối tượng sẽ nhận được sự kiện này. Thông tin về người nhận sự kiện được gọi là thông tin nhận diện. Nói một cách khác, yếu tố nhận diện xác định các đối tượng sẽ nhận sự kiện. Bên cạnh đó, còn có thể có các dữ liệu bổ sung thuộc về các đối tượng khác, không nhất thiết phải là đối tượng gửi hay nhận sự kiện.

Về mặt nguyên tắc, các sự kiện thuộc dạng phát tin (Broadcast) sẽ được truyền đến cho tất cả các đối tượng. Nếu sự kiện này là không quan trọng đối với đối tượng nào đó trong trạng thái hiện thời của nó thì đối tượng sẽ bỏ qua sự kiện.

4.2- Thông điệp (Message):

Trong lập trình hướng đối tượng, một tương tác giữa hai đối tượng được thực thi dưới dạng thông điệp được gửi từ đối tượng này sang đối tượng khác. Trong ngữ cảnh này, yếu tố quan trọng là không nên hiểu danh từ "thông điệp" quá chính xác theo nghĩa văn học bình thường. Một thông điệp ở đây thường được thực hiện qua một lệnh gọi thủ tục đơn giản (một đối tượng này gọi một thủ tục của một đối tượng khác); khi thủ tục đã được thực hiện xong, quyền điều khiển được trao trở về cho đối tượng gọi thủ tục cùng với giá trị trả về. Một thông điệp mặt khác cũng có thể là một thông điệp thực thụ được gửi qua một số cơ chế giao tiếp nào đó, hoặc là qua mạng hoặc là nội bộ trong một máy tính, đây là điều thường xảy ra trong các hệ thống thời gian thực. Thông điệp được thể hiện trong tất cả các loại biểu đồ động (tuần tự, cộng tác, hoạt động và trạng thái) theo ý nghĩa là sự giao tiếp giữa các đối tượng. Một thông điệp được vẽ là một đường thẳng với mũi tên nối giữa đối tượng gửi và đối tượng nhận thông điệp. Loại mũi tên sẽ chỉ rõ loại thông điệp.

Hình 6.3 chỉ rõ các loại thông điệp được sử dụng trong UML.



Hình 6.3- Các ký hiệu của các kiểu thông điệp

🔗 **Thông điệp đơn giản (simple):** Chỉ miêu tả đơn giản chiều điều khiển. Nó chỉ ra quyền điều khiển được trao từ đối tượng này sang cho đối tượng khác mà không kèm thêm lời miêu tả bất kỳ một chi tiết nào về sự giao tiếp đó. Loại thông điệp này được sử dụng khi người ta không biết các chi tiết về giao tiếp hoặc coi chúng là không quan trọng đối với biểu đồ.

🔗 **Thông điệp đồng bộ (synchronous):** thường được thực thi là một lệnh gọi thủ tục. Thủ tục xử lý thông điệp này phải được hoàn tất (bao gồm bất kỳ những thông điệp nào được lồng vào trong, được gửi như là một thành phần của sự xử lý) trước khi đối tượng gọi tiếp tục thực thi. Quá trình trở về có thể được chỉ ra dưới dạng thông điệp đơn giản.

🔗 **Thông điệp không đồng bộ (asynchronous):** đây là dạng điều khiển trình tự không đồng bộ, nơi không có một sự trở về đối với đối tượng gọi và nơi đối tượng gửi thông điệp tiếp tục quá trình thực thi của mình sau khi đã gửi thông điệp đi, không chờ cho tới khi nó được xử lý xong. Loại thông điệp này thường được sử dụng trong các hệ thống thời gian thực, nơi các đối tượng thực thi đồng thời.

Thông điệp đơn giản và thông điệp đồng bộ có thể được kết hợp với nhau trong chỉ một đường thẳng chỉ thông điệp với mũi tên chỉ thông điệp đồng bộ ở một phía và mũi tên chỉ thông điệp đơn giản ở phía kia. Điều này chỉ rõ rằng sự trả về được xảy ra hầu như ngay lập tức sau lệnh gọi hàm.

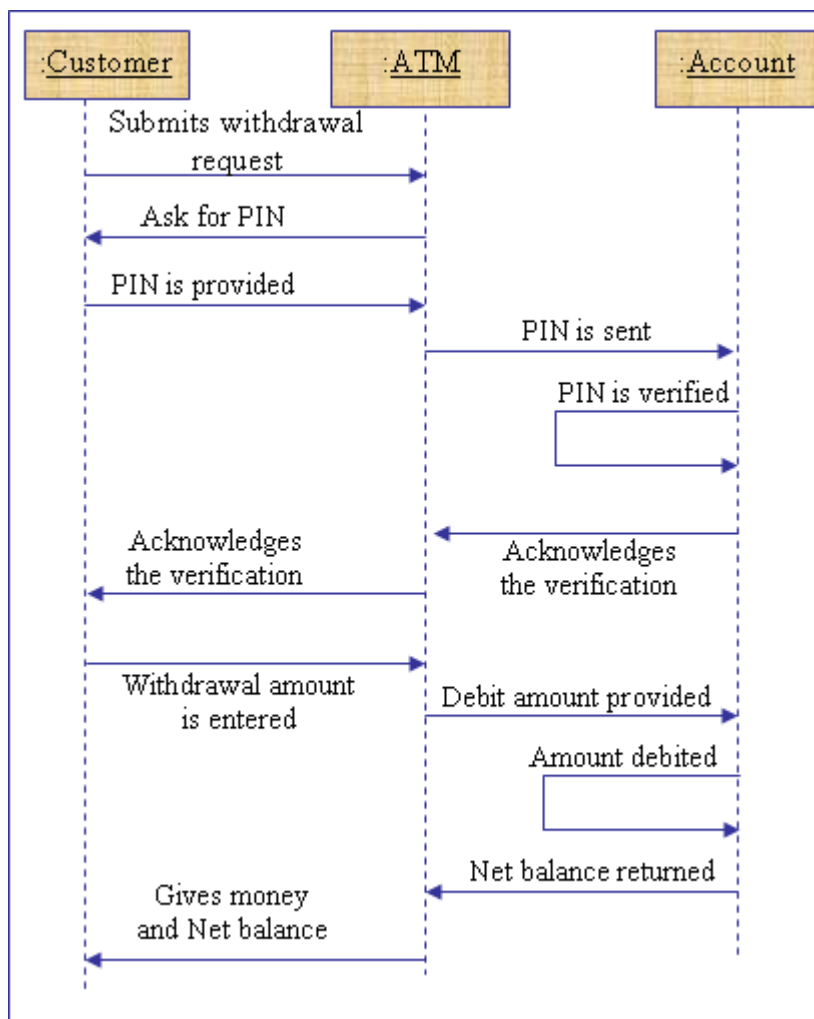
5- BIỂU ĐỒ TUẦN TỰ (SEQUENCE DIAGRAM)

Biểu đồ tuần tự minh họa các đối tượng tương tác với nhau ra sao. Chúng tập trung vào các chuỗi thông điệp, có nghĩa là các thông điệp được gửi và nhận giữa một loạt các đối tượng như thế nào. Biểu đồ tuần tự có hai trục: trục nằm dọc chỉ thời gian, trục nằm ngang chỉ ra một tập hợp các đối tượng. Một biểu đồ tuần tự

cũng nêu bật sự tương tác trong một cảnh kịch (scenario) – một sự tương tác sẽ xảy ra tại một thời điểm nào đó trong quá trình thực thi của hệ thống.

Từ các hình chữ nhật biểu diễn đối tượng có các đường gạch rời (dashed line) thẳng đứng biểu thị đường đời đối tượng, tức là sự tồn tại của đối tượng trong chuỗi tương tác. Trong khoảng thời gian này, đối tượng được thực thể hóa, sẵn sàng để gửi và nhận thông điệp. Quá trình giao tiếp giữa các đối tượng được thể hiện bằng các đường thẳng thông điệp nằm ngang nối các đường đời đối tượng. Mỗi tên ở đầu đường thẳng sẽ chỉ ra loại thông điệp này mang tính đồng bộ, không đồng bộ hay đơn giản. Để đọc biểu đồ tuần tự, hãy bắt đầu từ phía bên trên của biểu đồ rồi chạy dọc xuống và quan sát sự trao đổi thông điệp giữa các đối tượng xảy ra dọc theo tiến trình thời gian.

Ví dụ hãy quan sát một cảnh kịch rút tiền mặt tại một máy ATM của một nhà băng là:



Hình 6.4- Biểu đồ cảnh kịch rút tiền mặt tại máy ATM

Biểu đồ trên có thể được diễn giải theo trình tự thời gian như sau:

- Có ba lớp tham gia cảnh kịch này: khách hàng, máy ATM và tài khoản.
- Khách hàng đưa yêu cầu rút tiền vào máy ATM
- Đối tượng máy ATM yêu cầu khách hàng cung cấp mã số
- Mã số được gửi cho hệ thống để kiểm tra tài khoản
- Đối tượng tài khoản kiểm tra mã số và báo kết quả kiểm tra đến cho ATM
- ATM gửi kết quả kiểm tra này đến khách hàng
- Khách hàng nhập số tiền cần rút.
- ATM gửi số tiền cần rút đến cho tài khoản
- Đối tượng tài khoản trừ số tiền đó vào mức tiền trong tài khoản. Tại thời điểm này, chúng ta thấy có một mũi tên quay trở lại chỉ vào đối tượng tài khoản. Ý nghĩa của nó là đối tượng tài khoản xử lý yêu cầu này trong nội bộ đối tượng và không gửi sự kiện đó ra ngoài.
- Đối tượng tài khoản trả về mức tiền mới trong tài khoản cho máy ATM.
- Đối tượng ATM trả về mức tiền mới trong tài khoản cho khách hàng và dĩ nhiên, cả lượng tiền khách hàng đã yêu cầu được rút.

Đối tượng tài khoản chỉ bắt đầu được sinh ra khi đối tượng ATM cần tới nó để kiểm tra mã số và đối tượng tài khoản tiếp tục sống cho tới khi giao dịch được hoàn tất. Sau đó, nó chết đi. Bởi khách hàng có thể muốn tiếp tục thực hiện các giao dịch khác nên đối tượng khách hàng và đối tượng máy ATM vẫn tiếp tục tồn tại, điều này được chỉ ra qua việc các đường đời đối tượng được kéo vượt quá đường thẳng thể hiện sự kiện cuối cùng trong chuỗi tương tác.

Loại tương tác này là rất hữu dụng trong một hệ thống có một số lượng nhỏ các đối tượng với một số lượng lớn các sự kiện xảy ra giữa chúng. Mặc dù vậy, khi số lượng các đối tượng trong một hệ thống tăng lên thì mô hình này sẽ không còn mấy thích hợp.

Để có thể vẽ biểu đồ tuần tự, đầu tiên hãy xác định các đối tượng liên quan và thể hiện các sự kiện xảy ra giữa chúng.

Khi vẽ biểu đồ tuần tự, cần chú ý:

- Sự kiện được biểu diễn bằng các đường thẳng nằm ngang.
- Đối tượng bằng các đường nằm dọc.

- Thời gian được thể hiện bằng đường thẳng nằm dọc bắt đầu từ trên biểu đồ. Điều đó có nghĩa là các sự kiện cần phải được thể hiện theo đúng thứ tự mà chúng xảy ra, vẽ từ trên xuống dưới.

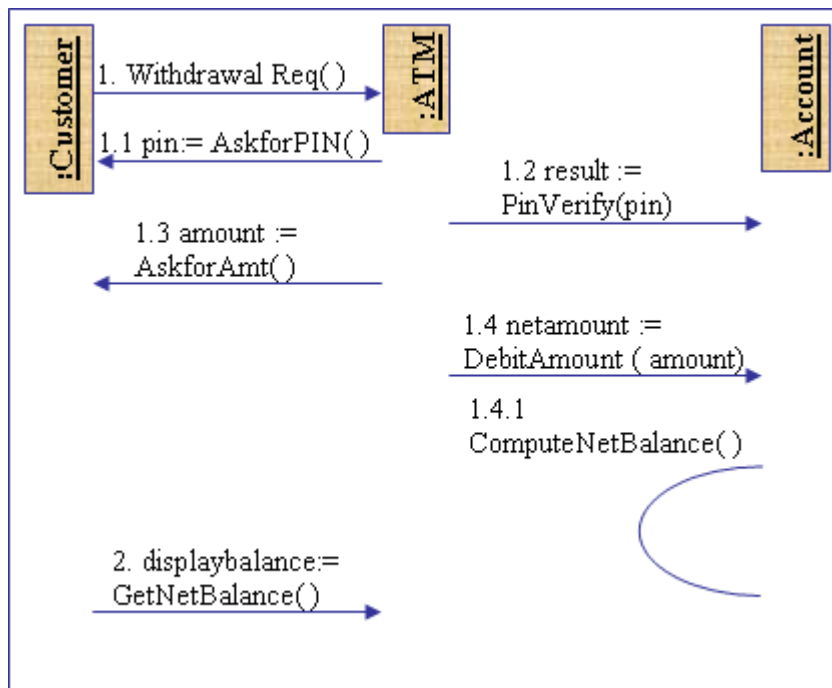
6- BIỂU ĐỒ CỘNG TÁC (COLLABORATION DIAGRAM)

Một biểu đồ cộng tác miêu tả tương tác giữa các đối tượng cũng giống như biểu đồ tuần tự, nhưng nó tập trung trước hết vào các sự kiện, tức là tập trung chủ yếu vào sự tương tác giữa các đối tượng.

Trong một biểu đồ cộng tác, các đối tượng được biểu diễn bằng kí hiệu lớp. Thứ tự trong biểu đồ cộng tác được thể hiện bằng cách đánh số các thông điệp. Kỹ thuật đánh số được coi là hơi có phần khó hiểu hơn so với kỹ thuật mũi tên sử dụng trong biểu đồ tuần tự. Nhưng ưu điểm của biểu đồ cộng tác là nó có thể chỉ ra các chi tiết về các lệnh gọi hàm (thủ tục), yếu tố được né tránh trong biểu đồ tuần tự.

Biểu đồ sau đây là một ví dụ cho một biểu đồ cộng tác, được chuẩn bị cũng cho một cảnh kịch rút tiền mặt như trong biểu đồ tuần tự của phần trước. Hãy quan sát các thứ tự số trong biểu đồ. Đầu tiên thủ tục `WithdrawalReq()` được gọi từ lớp khách hàng. Đó là lệnh gọi số 1. Bước tiếp theo trong tuần tự là hàm `AskForPin()`, số 1.1, được gọi từ lớp ATM. Thông điệp trong biểu đồ được viết dưới dạng `pin:=AskForPin()`, thể hiện rằng "giá trị trả về" của hàm này chính là mã số mà lớp khách hàng sẽ cung cấp.

Hình cung bên lớp tài khoản biểu thị rằng hàm `ComputeNetBalance()` được gọi trong nội bộ lớp tài khoản và nó xử lý cục bộ. Thường thì nó sẽ là một thủ tục riêng (private) của lớp.



Hình 6.5- Một biểu đồ cộng tác của kịch bản rút tiền ở máy ATM

7- BIỂU ĐỒ TRẠNG THÁI (STATE DIAGRAM)

Biểu đồ trạng thái nắm bắt vòng đời của các đối tượng, các hệ thống con (Subsystem) và các hệ thống. Chúng cho ta biết các trạng thái mà một đối tượng có thể có và các sự kiện (các thông điệp nhận được, các khoảng thời gian đã qua đi, các lỗi xảy ra, các điều kiện được thỏa mãn) sẽ ảnh hưởng đến những trạng thái đó như thế nào dọc theo tiến trình thời gian. Biểu đồ trạng thái có thể đính kèm với tất cả các lớp có những trạng thái được nhận diện rõ ràng và có lỗi ứng xử phức tạp. Biểu đồ trạng thái xác định ứng xử và miêu tả nó sẽ khác biệt ra sao phụ thuộc vào trạng thái, ngoài ra nó cũng còn miêu tả rõ những sự kiện nào sẽ thay đổi trạng thái của các đối tượng của một lớp.

7.1- Trạng thái và sự biến đổi trạng thái (State transition)

Tất cả các đối tượng đều có trạng thái; trạng thái là một kết quả của các hoạt động trước đó đã được đối tượng thực hiện và nó thường được xác định qua giá trị của các thuộc tính cũng như các nối kết của đối tượng với các đối tượng khác. Một lớp có thể có một thuộc tính đặc biệt xác định trạng thái, hoặc trạng thái cũng có thể được xác định qua giá trị của các thuộc tính "bình thường" trong đối tượng. Ví dụ về các trạng thái của đối tượng:

- Hóa đơn (đối tượng) đã được trả tiền (trạng thái).
- Chiếc xe ô tô (đối tượng) đang đứng yên (trạng thái).
- Động cơ (đối tượng) đang chạy (trạng thái).

- Jen (đối tượng) đang đóng vai trò người bán hàng (trạng thái).
- Kate (đối tượng) đã lấy chồng (trạng thái).

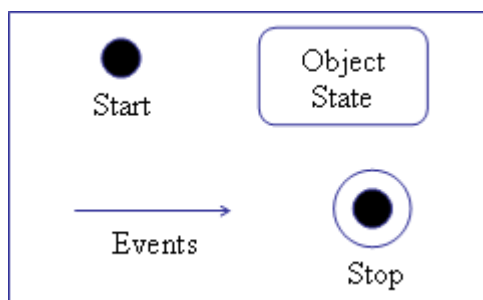
Một đối tượng sẽ thay đổi trạng thái khi có một việc nào đó xảy ra, thứ được gọi là sự kiện; ví dụ có ai đó trả tiền cho hóa đơn, bật động cơ xe ô tô hay là lấy chồng lấy vợ. Khía cạnh động có hai chiều không gian: tương tác và sự biến đổi trạng thái nội bộ. Tương tác miêu tả lối ứng xử đối ngoại của các đối tượng và chỉ ra đối tượng này sẽ tương tác với các đối tượng khác ra sao (qua việc gửi thông điệp, nối kết hoặc chấm dứt nối kết). Sự biến đổi trạng thái nội bộ miêu tả một đối tượng sẽ thay đổi các trạng thái ra sao – ví dụ giá trị các thuộc tính nội bộ của nó sẽ thay đổi như thế nào. Biểu đồ trạng thái được sử dụng để miêu tả việc bản thân đối tượng phản ứng ra sao trước các sự kiện và chúng thay đổi các trạng thái nội bộ của chúng như thế nào, ví dụ, một hóa đơn sẽ chuyển từ trạng thái chưa trả tiền sang trạng thái đã trả tiền khi có ai đó trả tiền cho nó. Khi một hóa đơn được tạo ra, đầu tiên nó bước vào trạng thái chưa được trả tiền.

7.2- Biểu đồ trạng thái

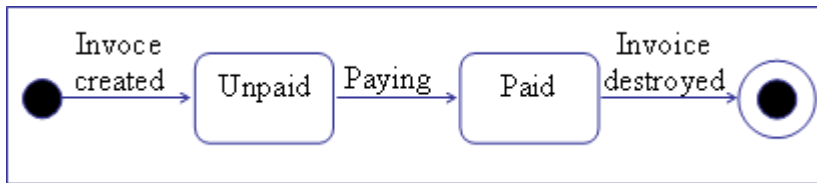
Biểu đồ trạng thái thể hiện những khía cạnh mà ta quan tâm tới khi xem xét trạng thái của một đối tượng:

- Trạng thái ban đầu
- Một số trạng thái ở giữa
- Một hoặc nhiều trạng thái kết thúc
- Sự biến đổi giữa các trạng thái
- Những sự kiện gây nên sự biến đổi từ một trạng thái này sang trạng thái khác

Hình sau sẽ chỉ ra các ký hiệu UML thể hiện trạng thái bắt đầu và trạng thái kết thúc, sự kiện cũng như các trạng thái của một đối tượng.

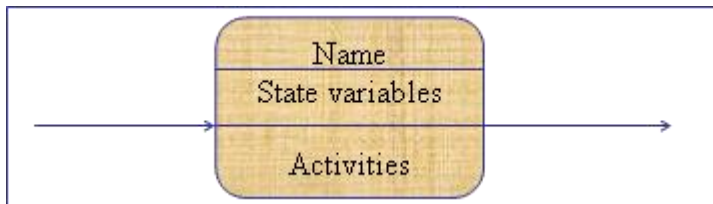


Hình 6.6- Các ký hiệu UML thể hiện bắt đầu, kết thúc, sự kiện và trạng thái của một đối tượng.



Hình 6.7- Biểu đồ trạng thái thực hiện hoá đơn.

Một trạng thái có thể có ba thành phần, như được chỉ trong hình sau:



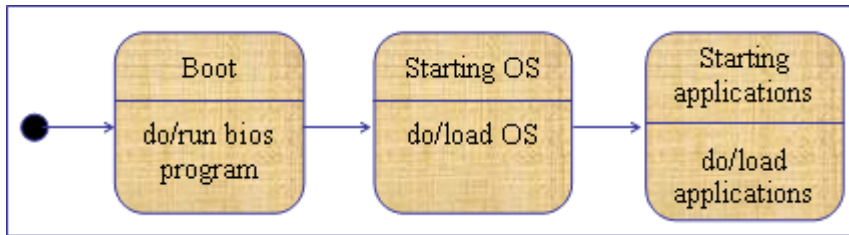
Hình 6.8- Các ngăn Tên, Biến trạng thái và hành động

Phần thứ nhất chỉ ra tên của trạng thái, ví dụ như chờ, đã được trả tiền hay đang chuyển động. Phần thứ hai (không bắt buộc) dành cho các biến trạng thái. Đây là những thuộc tính của lớp được thể hiện qua biểu đồ trạng thái; nhiều khi các biến tạm thời cũng tỏ ra rất hữu dụng trong trạng thái, ví dụ như các loại biến đếm (counter). Phần thứ ba (không bắt buộc) là phần dành cho hoạt động, nơi các sự kiện và các hành động có thể được liệt kê. Có ba loại sự kiện chuẩn hóa có thể được sử dụng cho phần hành động: *entry* (đi vào), *exit* (đi ra), và *do* (thực hiện). Loại *sự kiện đi vào* được sử dụng để xác định các hành động khởi nhập trạng thái, ví dụ gán giá trị cho một thuộc tính hoặc gửi đi một thông điệp. *Sự kiện đi ra* có thể được sử dụng để xác định hành động khi rời bỏ trạng thái. *Sự kiện thực hiện* được sử dụng để xác định hành động cần phải được thực hiện trong trạng thái, ví dụ như gửi một thông điệp, chờ, hay tính toán. Ba loại sự kiện chuẩn này không thể được sử dụng cho các mục đích khác.

Một sự biến đổi trạng thái thường có một sự kiện đi kèm với nó, nhưng không bắt buộc. Nếu có một sự kiện đi kèm, sự thay đổi trạng thái sẽ được thực hiện khi sự kiện kia xảy ra. Một hành động loại *thực hiện* trong trạng thái có thể là một quá trình đang tiếp diễn (ví dụ chờ, điều khiển các thủ tục,...) phải được thực hiện trong khi đối tượng vẫn ở nguyên trong trạng thái này. Một hành động *thực hiện* có thể bị ngắt bởi các sự kiện từ ngoài, có nghĩa là một sự kiện kiện gây nên một sự biến đổi trạng thái có thể ngưng ngắt một hành động thực hiện mang tính nội bộ đang tiếp diễn.

Trong trường hợp một sự biến đổi trạng thái không có sự kiện đi kèm thì trạng thái sẽ thay đổi khi hành động nội bộ trong trạng thái đã được thực hiện xong (hành động nội bộ kiểu đi vào, đi ra, thực hiện hay các hành động do người sử dụng định nghĩa). Theo đó, khi tất cả các hành động thuộc trạng thái đã được

thực hiện xong, một sự thay đổi trạng thái sẽ tự động xảy ra mà không cần sự kiện từ ngoài.



Hình 6.9- Biến đổi trạng thái không có sự kiện từ ngoài. Sự thay đổi trạng thái xảy ra khi các hoạt động trong mỗi trạng thái được thực hiện xong.

7.3- Nhận biết trạng thái và sự kiện

Quá trình phát hiện sự kiện và trạng thái về mặt bản chất bao gồm việc hỏi một số các câu hỏi thích hợp:

- Một đối tượng có thể có những trạng thái nào?: Hãy liệt kê ra tất cả những trạng thái mà một đối tượng có thể có trong vòng đời của nó.
- Những sự kiện nào có thể xảy ra?: Bởi sự kiện gây ra việc thay đổi trạng thái nên nhận ra các sự kiện là một bước quan trọng để nhận diện trạng thái.
- Trạng thái mới sẽ là gì?: Sau khi nhận diện sự kiện, hãy xác định trạng thái khi sự kiện này xảy ra và trạng thái sau khi sự kiện này xảy ra.
- Có những thủ tục nào sẽ được thực thi?: Hãy để ý đến các thủ tục ảnh hưởng đến trạng thái của một đối tượng.
- Chuỗi tương tác giữa các đối tượng là gì?: Tương tác giữa các đối tượng cũng có thể ảnh hưởng đến trạng thái của đối tượng.
- Qui định nào sẽ được áp dụng cho các phản ứng của các đối tượng với nhau?: Các qui định kiểm tra phản ứng đối với một sự kiện sẽ xác định rõ hơn các trạng thái.
- Những sự kiện và sự chuyển tải nào là không thể xảy ra?: Nhiều khi có một số sự kiện hoặc sự thay đổi trạng thái không thể xảy ra. Ví dụ như bán một chiếc ô tô đã được bán rồi.
- Cái gì khiến cho một đối tượng được tạo ra?: Đối tượng được tạo ra để trả lời cho một sự kiện. Ví dụ như một sinh viên ghi danh cho một khóa học.

- Cái gì khiến cho một đối tượng bị hủy?: Đối tượng sẽ bị hủy đi khi chúng không được cần tới nữa. Ví dụ khi một sinh viên kết thúc một khóa học.
- Cái gì khiến cho đối tượng cần phải được tái phân loại (reclassified)? : Những loại sự kiện như một nhân viên được tăng chức thành nhà quản trị sẽ khiến cho động tác tái phân loại của nhân viên đó được thực hiện.

7.4- Một số lời mách bảo cho việc tạo dựng biểu đồ trạng thái

- Chuyển biểu đồ tuần tự thành biểu đồ trạng thái.
- Xác định các vòng lặp (loop)
- Bổ sung thêm các điều kiện biên và các điều kiện đặc biệt
- Trộn lẫn các cảnh kịch khác vào trong biểu đồ trạng thái.

Một khi mô hình đã được tạo nên, hãy nêu ra các câu hỏi và kiểm tra xem mô hình có khả năng cung cấp tất cả các câu trả lời. Qui trình sau đây cần phải được nhắc lại cho mỗi đối tượng.

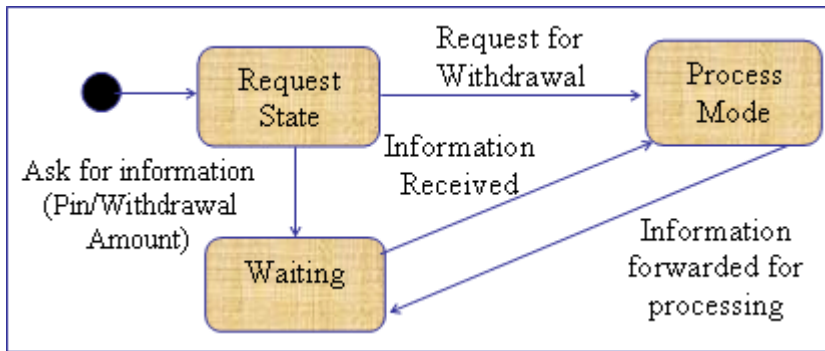
7.4.1- Chuyển biểu đồ tuần tự thành biểu đồ trạng thái

Hãy dõi theo một chuỗi các sự kiện được miêu tả trong biểu đồ, chuỗi này phải mang tính tiêu biểu cho các tương tác trong hệ thống. Hãy quan sát các sự kiện ảnh hưởng đến đối tượng mà ta đang nghiên cứu.

Hãy sắp xếp các sự kiện thành một đường dẫn, dán nhãn input (hoặc entry) và output (exit) cho các sự kiện. Khoảng cách giữa hai sự kiện này sẽ là một trạng thái.

Nếu cảnh kịch có thể được nhắc đi nhắc lại rất nhiều lần (vô giới hạn), hãy nối đường dẫn từ trạng thái cuối cùng đến trạng thái đầu tiên.

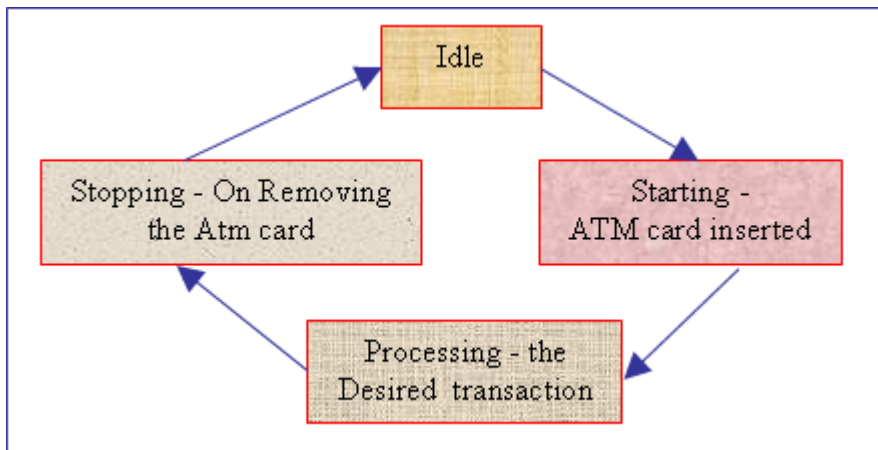
Biểu đồ sau đây chỉ ra biểu đồ trạng thái của một lớp máy ATM, được chiết suất từ biểu đồ tuần tự hoặc biểu đồ cộng tác đã được trình bày trong các phần trước.



Hình 6.10- Chuyển một biểu đồ tuần tự sang biểu đồ trạng thái

7.4.2- Nhận ra các vòng lặp (loop)

Một chuỗi sự kiện có thể được nhắc đi nhắc lại vô số lần được gọi là vòng lặp (loop).



Hình 6.11- Biểu đồ lặp

Chú ý:

- Trong một vòng lặp, chuỗi các sự kiện được nhắc đi nhắc lại cần phải đồng nhất với nhau. Nếu có một chuỗi các sự kiện khác chuỗi khác thì trường hợp đó không có vòng lặp.
- Lý tưởng nhất là một trạng thái trong vòng lặp sẽ có sự kiện kết thúc. Đây là yếu tố quan trọng, nếu không thì vòng lặp sẽ không bao giờ kết thúc.

7.4.3- Bổ sung thêm các điều kiện biên và các điều kiện đặc biệt

Sau khi đã hoàn tất biểu đồ trạng thái cho mọi đối tượng cần thiết trong hệ thống, đã đến lúc chúng ta cần kiểm tra, đối chứng chúng với điều kiện biên và các điều kiện đặc biệt khác, những điều kiện rất có thể đã chưa được quan tâm đủ độ trong thời gian tạo dựng biểu đồ trạng thái. Điều kiện biên là những điều

kiện thao tác trên giá trị, đây là những giá trị nằm bên ranh giới của một điều kiện để quyết định về trạng thái của đối tượng, ví dụ như quy định về kỳ hạn của một tài khoản là 30 ngày thì ngày thứ 31 đối với tài khoản này sẽ là một điều kiện biên. Các điều kiện đặc biệt là những điều kiện ngoại lệ, ví dụ ngày thứ 30 của tháng 2 năm 2000 (nếu có một điều kiện thật sự như vậy tồn tại ngoài đời thực).

7.4.4- Trộn lẫn các cảnh kịch khác vào trong biểu đồ trạng thái

Một khi biểu đồ trạng thái cho một đối tượng đã sẵn sàng, chúng ta cần phải trộn những chuỗi sự kiện có ảnh hưởng đến đối tượng này vào trong biểu đồ trạng thái. Điều này có nghĩa là chúng ta cần phải quan sát các hiệu ứng gián tiếp của các sự kiện khác đối với đối tượng đang là chủ đề chính của biểu đồ trạng thái. Đây là việc quan trọng, bởi các đối tượng trong một hệ thống tương tác với nhau và vì các đối tượng khác cũng có khả năng gây nên sự kiện cho một đối tượng định trước, nên lối ứng xử này cũng cần phải được thể hiện trong biểu đồ trạng thái.

Điểm bắt đầu cho công việc này là:

- Ấn định một điểm bắt đầu chung cho tất cả các chuỗi sự kiện bổ sung.
- Xác định điểm nơi các ứng xử bắt đầu khác biệt với những ứng xử đã được mô hình hóa trong biểu đồ trạng thái.

Bổ sung thêm sự các biến đổi mới từ trạng thái này, trong tư cách một đường dẫn thay thế. Cần để ý đến những đường dẫn có vẻ ngoài đồng nhất nhưng thật ra có khác biệt trong một tình huống nhất định nào đó.

Hãy chú ý đến các sự kiện xảy ra trong những tình huống bất tiện. Mỗi sự kiện do khách hàng hay người sử dụng gây nên đều có thể sa vào trạng thái của các sự kiện bất tiện. Hệ thống không nắm quyền điều khiển đối với người sử dụng và người sử dụng có thể quyết định để làm nảy ra một sự kiện tại một thời điểm tiện lợi đối với anh ta. Ví dụ như khách hàng có thể quyết định kết thúc trước kỳ hạn một tài khoản đầu tư.

Một trường hợp khác cũng cần phải được xử lý là sự kiện do người sử dụng gây nên không thể xảy ra. Có một loạt các lý do (người sử dụng thiếu tập trung, buồn nản, lơ đãng...) khiến cho sự kiện loại này không xảy ra. Cả trường hợp này cũng phải được xử lý thấu đáo. Ví dụ một khách hàng thất bại trong việc báo cho nhà băng biết những mệnh lệnh của anh ta về kỳ hạn của tài khoản, một tấm séc được viết ra nhưng lại không có khả năng giải tỏa mức tiền cần thiết.

Nhìn theo phương diện các biểu đồ trạng thái như là một thành phần của mô hình động, cần chú ý những điểm sau:

- Biểu đồ trạng thái chỉ cần được tạo dựng nên cho các lớp đối tượng có ứng xử động quan trọng.
- Hãy thăm tra biểu đồ trạng thái theo khía cạnh tính nhất quán đối với những sự kiện dùng chung để cho toàn bộ mô hình động được đúng đắn.
- Dùng các trường hợp sử dụng để hỗ trợ cho quá trình tạo dựng biểu đồ trạng thái.
- Khi định nghĩa một trạng thái, hãy chỉ để ý đến những thuộc tính liên quan.

8- BIỂU ĐỒ HOẠT ĐỘNG (ACTIVITY DIAGRAM)

Biểu đồ hoạt động nắm bắt hành động và các kết quả của chúng. Biểu đồ hoạt động tập trung vào công việc được thực hiện trong khi thực thi một thủ tục (hàm), các hoạt động trong một lần thực thi một trường hợp sử dụng hoặc trong một đối tượng. Biểu đồ hoạt động là một biến thể của biểu đồ trạng thái và có một mục tiêu tương đối khác, đó là nắm bắt hành động (công việc và những hoạt động phải được thực hiện) cũng như kết quả của chúng theo sự biến đổi trạng thái. Các trạng thái trong biểu đồ hoạt động (được gọi là các trạng thái hành động) sẽ chuyển sang giai đoạn kế tiếp khi hành động trong trạng thái này đã được thực hiện xong (mà không xác định bất kỳ một sự kiện nào theo như nội dung của biểu đồ trạng thái). Một sự điểm phân biệt khác giữa biểu đồ hoạt động và biểu đồ trạng thái là các hành động của nó được định vị trong các **luồng (swimlane)**. Một luồng sẽ gom nhóm các hoạt động, chú ý tới khái niệm người chịu trách nhiệm cho chúng hoặc chúng nằm ở đâu trong một tổ chức. Một biểu đồ hoạt động là một phương pháp bổ sung cho việc miêu tả tương tác, đi kèm với trách nhiệm thể hiện rõ các hành động xảy ra như thế nào, chúng làm gì (thay đổi trạng thái đối tượng), chúng xảy ra khi nào (chuỗi hành động), và chúng xảy ra ở đâu (luồng hành động).

Biểu đồ hoạt động có thể được sử dụng cho nhiều mục đích khác nhau, ví dụ như:

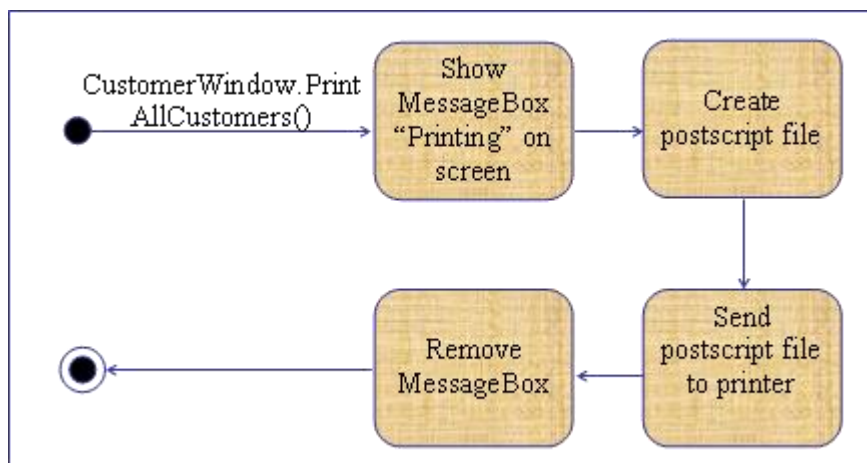
- Để nắm bắt công việc (hành động) sẽ phải được thực thi khi một thủ tục được thực hiện. Đây là tác dụng thường gặp nhất và quan trọng nhất của biểu đồ hoạt động.
- Để nắm bắt công việc nội bộ trong một đối tượng.

- Để chỉ ra một nhóm hành động liên quan có thể được thực thi ra sao, và chúng sẽ ảnh hưởng đến những đối tượng nằm xung quanh chúng như thế nào.
- Để chỉ ra một trường hợp sử dụng có thể được thực thể hóa như thế nào, theo khái niệm hành động và các sự biến đổi trạng thái của đối tượng.
- Để chỉ ra một doanh nghiệp hoạt động như thế nào theo các khái niệm công nhân (tác nhân), qui trình nghiệp vụ (workflow), hoặc tổ chức và đối tượng (các khía cạnh vật lý cũng như tri thức được sử dụng trong doanh nghiệp).

Biểu đồ hoạt động có thể được coi là một loại Flow chart. Điểm khác biệt là Flow Chart bình thường ra chỉ được áp dụng đối với các qui trình tuần tự, biểu đồ hoạt động có thể xử lý cả các qui trình song song.

Hành động và sự thay đổi trạng thái

Một hành động được thực hiện để sản sinh ra một kết quả. Việc thực thi của thủ tục có thể được miêu tả dưới dạng một tập hợp của các hành động liên quan, sau này chúng sẽ được chuyển thành các dòng code thật sự. Theo như định nghĩa ở phần trước, một biểu đồ hoạt động chỉ ra các hành động cùng mối quan hệ giữa chúng và có thể có một điểm bắt đầu và một điểm kết thúc. Biểu đồ hoạt động sử dụng cũng cùng những ký hiệu như trong biểu đồ trạng thái bình thường.



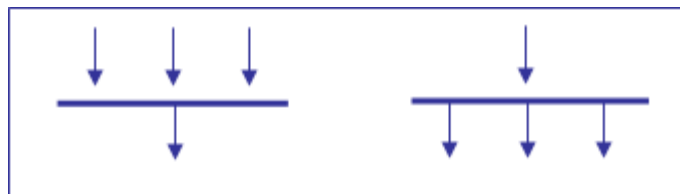
Hình 6.12- Khi một người gọi tác vụ PrintAllCustomer (trong lớp CustomerWindow), các hành động khởi động. hành động đầu tiên là hiện một hộp thông báo lên màn hình; hành động thứ hai là tạo một tập tin postscript; hành động thứ ba là gửi file postscript đến máy in; và hành động thứ tư là xóa hộp thông báo trên màn hình. Các hành động được chuyển tiếp tự động; chúng xảy ra ngay khi hành động trong giai đoạn nguồn được thực hiện.

Các sự thay đổi có thể được bảo vệ bởi các điều kiện cạnh giữ (Guard-condition), các điều kiện này phải được thỏa mãn thì sự thay đổi mới nổ ra. Một ký hiệu hình thoi được sử dụng để thể hiện một quyết định. Ký hiệu quyết định có thể có một hoặc nhiều sự thay đổi đi vào và một hoặc nhiều sự thay đổi đi ra được dán nhãn đi kèm các điều kiện bảo vệ. Bình thường ra, một trong số các sự thay đổi đi ra bao giờ cũng được thỏa mãn (là đúng).

Một sự thay đổi được chia thành hai hay nhiều sự thay đổi khác sẽ dẫn đến các hành động xảy ra song song. Các hành động được thực hiện đồng thời, mặc dù chúng cũng có thể được thực hiện lần lượt từng cái một. Yếu tố quan trọng ở đây là tất cả các thay đổi đồng thời phải được thực hiện trước khi chúng được thống nhất lại với nhau (nếu có). Một đường thẳng nằm ngang kẻ đậm (còn được gọi là thanh đồng bộ hóa – Synchronisation Bar) chỉ rằng một sự thay đổi được chia thành nhiều nhánh khác nhau và chỉ ra một sự chia sẻ thành các hành động song song. Cũng đường thẳng đó được sử dụng để chỉ ra sự thống nhất các nhánh.

Kí hiệu UML cho các thành phần **căn bản của biểu đồ hoạt động**:

- *Hoạt động (Activity)*: là một qui trình được định nghĩa rõ ràng, có thể được thực thi qua một hàm hoặc một nhóm đối tượng. Hoạt động được thể hiện bằng hình chữ nhật bo tròn cạnh.
- *Thanh đồng bộ hóa (Synchronisation bar)*: chúng cho phép ta mở ra hoặc là đóng lại các nhánh chạy song song nội bộ trong tiến trình.



Hình 6.13- Thanh đồng bộ hóa

- *Điều kiện cạnh giữ (Guard Condition)*: các biểu thức logic có giá trị hoặc đúng hoặc sai. Điều kiện cạnh giữ được thể hiện trong ngoặc vuông, ví dụ:

[Customer existing].

9- VÒNG ĐỜI ĐỐI TƯỢNG (OBJECT LIFECYCLE)

Vòng đời mà một đối tượng đi qua phụ thuộc vào loại đối tượng. Có hai loại vòng đời khác nhau đối với một đối tượng: vòng đời sinh ra rồi chết đi; và vòng đời vòng lặp.

◆ 9.1- Vòng đời sinh ra và chết đi:

Trong một vòng đời sinh ra rồi chết đi:

- Sẽ có một hay nhiều trạng thái nơi đối tượng bắt đầu tồn tại. Những trạng thái này được gọi là trạng thái tạo ra đối tượng.
- Sẽ có một hay nhiều trạng thái đóng tư cách là điểm kết thúc cho vòng đời của một đối tượng. Những trạng thái này được gọi là trạng thái kết thúc.

Có hai loại trạng thái kết thúc. Một dạng trạng thái kết thúc là loại nơi đối tượng bị hủy và không tiếp tục tồn tại nữa. Loại thứ hai là dạng trạng thái kết thúc mà sau đó đối tượng sẽ được lưu trữ lại hoặc chuyển sang trạng thái im lặng. Đối tượng tiếp tục tồn tại nhưng không tiếp tục thể hiện ứng xử động.

Sau trạng thái khởi tạo và trước trạng thái kết thúc, đối tượng có thể đi qua một hoặc là nhiều trạng thái trung gian. Tại mỗi một thời điểm, đối tượng phải ở một trạng thái hiện thời.

Không có một điểm nào sau trạng thái khởi tạo và trước trạng thái kết thúc mà đối tượng lại không có trạng thái.

Ví dụ cho đối tượng có vòng đời sinh ra và chết đi: khách hàng, tài khoản.

◆ 9.2- Vòng đời lặp

Khác với trường hợp sinh ra và chết đi, trong vòng đời vòng lặp:

- Đối tượng được coi là đã luôn luôn tồn tại ở đây và sẽ còn mãi mãi tiếp tục tồn tại.
- Không có trạng thái khởi tạo cũng không có trạng thái kết thúc.

Mặc dù thật ra đối tượng đã được tạo ra tại một thời điểm nào đó và cũng sẽ thật sự bị hủy diệt tại một thời điểm nào đó, nhưng nó vẫn được coi là luôn luôn tồn tại và có mặt. Thường thì những đối tượng tỏ ra có một vòng đời vòng lặp sẽ được tạo ra tại thời điểm cài đặt hệ thống và sẽ chết đi khi hệ thống kết thúc.

Một đối tượng với vòng đời vòng lặp sẽ có một hoặc là nhiều trạng thái "ngủ yên". Đó là những trạng thái nơi đối tượng nằm chờ một sự kiện xảy ra. Bên cạnh đó, đối tượng cũng luôn luôn ở trạng thái hiện thời.

Ví dụ cho đối tượng có vòng đời lặp lại: máy rút tiền tự động (ATM), nhân viên thu ngân.

10- XEM XÉT LẠI MÔ HÌNH ĐỘNG

◆ 10.1- Thẩm vấn biểu đồ trạng thái

Sau khi đã hoàn tất các thành phần căn bản của mô hình động như các biểu đồ tuần tự, biểu đồ cộng tác, biểu đồ trạng thái và biểu đồ hoạt động, nhóm phát triển có thể phác thảo biểu đồ thành phần và biểu đồ triển khai. Biểu đồ triển khai có thể được coi là biểu đồ cuối cùng trong mô hình động. Tới thời điểm này, có thể coi là ta đã hoàn tất một phiên bản của mô hình động.

Phần quan trọng nhất trong mô hình này là biểu đồ trạng thái. Hãy tìm câu trả lời cho một loạt các câu hỏi để xác định xem biểu đồ trạng thái đã đúng đắn và có một mức độ chi tiết thích hợp hay chưa. Công việc này cần nhằm tới hai mục đích:

- Kiểm tra tính trọn vẹn của mô hình
- Đảm bảo mọi điều kiện gây lỗi đã được xử lý

Trong giai đoạn này, có thể sẽ có các cảnh kịch (scenario) mới xuất hiện và gia nhập phạm vi quan sát của chúng ta, nếu trước đó có một số trạng thái chưa được xử lý. Những tình huống loại này là loại vấn đề có thể được giải quyết, song có thể né tránh qua việc xác định thật đầy đủ các sự kiện và trạng thái.

10.2- Phối hợp sự kiện

Bước cuối cùng là một vòng kiểm tra bổ sung nhằm đảm bảo tính đúng đắn của mô hình động:

- Kiểm tra để đảm bảo mỗi thông điệp đều có đối tượng gửi và đối tượng nhận. Trong một số trường hợp, số liệu chính xác của những đối tượng nhận sự kiện có thể không được biết tới, nhưng chúng ta phải đảm bảo rằng chúng ta biết những lớp nào sẽ xử lý những sự kiện này.
- Hãy nghiên cứu mô hình theo khía cạnh trạng thái, tìm ra những trạng thái không có trạng thái dẫn trước và không có trạng thái tiếp theo. Những trạng thái này rất có thể là trạng thái khởi đầu hoặc trạng thái kết thúc. Mặc dù vậy, nếu trạng thái đó không thuộc về một trong hai loại trạng thái kia, rất có thể đây là một triệu chứng cho thấy mô hình còn thiếu điều gì đó.
- Nhìn chung, tất cả các trạng thái thường đều có trạng thái dẫn trước và trạng thái tiếp sau.
- Hãy lần theo hiệu ứng của các sự kiện đi vào (entry) để đảm bảo là chúng tương thích với các trường hợp sử dụng nơi chúng xuất phát. Để làm điều này, hãy lần theo một sự kiện từ một đối tượng này đến đối tượng khác, kiểm tra xem mỗi sự kiện có phù hợp với

trường hợp sử dụng hay không. Trong trường hợp có mâu thuẫn, hãy sửa lại biểu đồ trạng thái hoặc trường hợp sử dụng để đảm bảo sự nhất quán.

- Kiểm tra lại những lỗi đồng bộ, có thể chúng là kết quả của một sự kiện không chờ đợi.

10.3- Bao giờ thì sử dụng biểu đồ nào

Không cần phải vẽ tất cả các loại biểu đồ động cho tất cả các loại hệ thống. Mặc dù vậy, trong một số trường hợp khác nhau chúng ta nhất thiết phải cần đến một số loại biểu đồ động nhất định. Sau đây là một vài lời mách bảo có thể giúp giải thích một vài điều còn chưa thông tỏ về việc sử dụng các loại biểu đồ động.

Biểu đồ tuần tự và biểu đồ cộng tác được vẽ khi chúng ta muốn xem xét ứng xử động của nhiều đối tượng/ lớp trong nội bộ một cảnh kịch của một trường hợp sử dụng. Biểu đồ tuần tự và biểu đồ cộng tác rất hữu dụng trong việc chỉ ra sự cộng tác giữa các đối tượng, nhưng chúng lại không hữu dụng khi muốn miêu tả ứng xử chính xác của một đối tượng.

Biểu đồ trạng thái được sử dụng để thể hiện ứng xử chính xác của một đối tượng.

Biểu đồ hoạt động được sử dụng để thể hiện lối ứng xử xuyên suốt nhiều trường hợp sử dụng hoặc các tiểu trình xảy ra song song của một lần thực thi.

Biểu đồ thành phần và biểu đồ triển khai được sử dụng để chỉ ra mối quan hệ vật lý giữa phần mềm và các thành phần phần cứng trong hệ thống.

10.4- Lớp con và biểu đồ trạng thái

Tất cả các lớp con đều thừa kế cả thuộc tính cũng như các thủ tục của lớp cha. Vì vậy, một lớp con cũng sẽ thừa kế cả mô hình động của lớp cha.

Ngoài biểu đồ trạng thái được thừa kế, lớp con cũng có biểu đồ trạng thái riêng của nó. Biểu đồ trạng thái của một lớp cha sẽ được mở rộng để bao chứa lối ứng xử chuyên biệt của lớp con.

Biểu đồ trạng thái của lớp con và biểu đồ trạng thái của lớp cha phải được bảo trì riêng biệt và độc lập. Biểu đồ trạng thái của lớp con cần phải được định nghĩa sử dụng các thuộc tính của lớp con chứ không phải chỉ bằng các thuộc tính của lớp cha. Mặt khác, vẫn có một sự móc nối ngoài ý muốn của lớp cha đến với lớp con thông qua các thuộc tính mà chúng sử dụng chung, ví dụ chỉ nên xem xét biểu đồ trạng thái cho các tài khoản có kỳ hạn theo phương diện sự thay đổi của chính các thuộc tính của chúng, chứ không phải là thuộc tính của lớp cha. Ta phải thực hiện như vậy để né tránh trường hợp trộn lẫn thuộc tính của lớp con và lớp cha.

Việc tuân thủ quy tắc kể trên trong quá trình vẽ biểu đồ trạng thái cho một lớp con sẽ đảm bảo tính mô đun cho động tác mở rộng của bạn.

11- PHỐI HỢP MÔ HÌNH ĐỐI TƯỢNG VÀ MÔ HÌNH ĐỘNG

Khi kết hợp giữa các mô hình đối tượng và mô hình động, mỗi sự kiện trong mô hình động cần phải tương thích với một thủ tục trong mô hình đối tượng. Từ đó suy ra, mỗi sự thay đổi về mặt trạng thái trong mô hình động cần phải phù hợp với một thủ tục của đối tượng. Hành động phụ thuộc vào trạng thái của đối tượng và vào sự kiện.

Mối quan hệ giữa mô hình đối tượng và mô hình động có thể được miêu tả như sau:

- Mô hình đối tượng là cơ cấu (framework) cho mô hình động.
- Mô hình động xác định các chuỗi thay đổi được phép xảy ra đối với các đối tượng trong mô hình đối tượng.
- Mô hình động bị hạn chế chỉ trong những đối tượng có mặt trong mô hình đối tượng cũng như cấu trúc của chúng.
- Không thể có một mô hình động cho một đối tượng không tồn tại trong mô hình đối tượng. Có một mối quan hệ 1-1 giữa mô hình đối tượng và mô hình động.
- Mô hình động chính là mô hình đối tượng cộng thêm với phần ứng xử "sống".
- Mô hình đối tượng miêu tả sự khác biệt giữa các đối tượng như là sự khác biệt giữa các lớp. Khi một đối tượng ứng xử khác một đối tượng khác thì mỗi đối tượng trong số đó sẽ có một lớp riêng.
- Mặc dù vậy, trong mô hình động, sự khác biệt trong ứng xử động sẽ được mô hình hóa thành các trạng thái khác nhau của cùng một lớp.

12- TÓM TẮT VỀ MÔ HÌNH ĐỘNG

Tất cả các hệ thống đều có cấu trúc tĩnh và có ứng xử động. Cấu trúc có thể được miêu tả qua các phần tử mô hình tĩnh, ví dụ như lớp, quan hệ giữa các lớp, nút mạng và thành phần. Khái niệm ứng xử miêu tả các phần tử mô hình trong nội bộ cấu trúc sẽ tương tác với nhau dọc theo tiến trình thời gian ra sao. Đó thường là những tương tác được xác định trước và có thể được mô hình hóa. Mô hình hóa ứng xử động của một hệ thống gọi là mô hình động, được UML hỗ trợ. Có tất cả

bốn loại biểu đồ khác nhau, mỗi loại với một mục đích khác nhau: biểu đồ trạng thái, biểu đồ tuần tự, biểu đồ cộng tác và biểu đồ hoạt động.

Biểu đồ trạng thái được sử dụng để miêu tả lối ứng xử cũng như các trạng thái nội bộ trong một lớp (nó cũng có thể được sử dụng cho các hệ thống con hoặc cho toàn bộ hệ thống). Nó tập trung vào khía cạnh các đối tượng theo tiến trình thời gian sẽ thay đổi các trạng thái của chúng ra sao tùy theo những sự kiện xảy ra, lối ứng xử cũng như các hành động được thực hiện trong các trạng thái, và bao giờ thì sự thay đổi trạng thái xảy ra. Một sự kiện có thể nổ ra khi một điều kiện trở thành được thỏa mãn, khi nhận một tín hiệu hoặc lệnh gọi thủ tục, hoặc là khi một khoảng thời gian định trước qua đi.


Biểu đồ tuần tự được sử dụng để miêu tả một nhóm các đối tượng sẽ tương tác với nhau trong một cảnh kịch riêng biệt như thế nào. Nó tập trung vào chuỗi thông điệp, tức là câu hỏi các thông điệp được gửi và nhận giữa một nhóm các đối tượng như thế nào. Biểu đồ tuần tự có hai trục; trục dọc chỉ thời gian và trục nằm ngang chỉ ra các đối tượng tham gia cảnh kịch. Khía cạnh quan trọng nhất của một biểu đồ tuần tự là thời gian.

Biểu đồ cộng tác được sử dụng để miêu tả các đối tượng tương tác với nhau trong không gian bộ nhớ (space), có nghĩa là bên cạnh các tương tác động, nó còn miêu tả rõ ràng các đối tượng được nối kết với nhau như thế nào. Trong biểu đồ cộng tác không có trục cho thời gian; thay vào đó, các thông điệp sẽ được đánh số để tạo chuỗi.

Biểu đồ hoạt động được sử dụng để miêu tả sự việc xảy ra ra sao, công việc được thực hiện như thế nào. Biểu đồ hoạt động cũng có thể được sử dụng cho các thủ tục, các lớp, các trường hợp sử dụng, và cũng có thể được sử dụng để chỉ ra các quy trình nghiệp vụ (workflow).

PHẦN CÂU HỎI

 **Hỏi:** Thế nào là một vòng lặp?

 **Đáp:** Một chuỗi sự kiện có thể được nhắc đi, nhắc lại vô số lần được gọi là vòng lặp (loop).

 **Hỏi:** Mô hình động chính là mô hình đối tượng cộng thêm phần ứng xử động của hệ thống

 **Đáp:** Đúng

 **Hỏi:** Các sự kiện độc lập cũng có thể là các sự kiện song song

 **Đáp:** Đúng

❏ **Hỏi:** Một đối tượng không nhất thiết phải có trạng thái.

▶ **Đáp:** Sai, mọi đối tượng đều có trạng thái

❏ **Hỏi:** Một lớp có thể có trạng thái ban đầu và trạng thái kết thúc.

▶ **Đáp:** Sai, một đối tượng có thể có trạng thái ban đầu và trạng thái kết thúc.

❏ **Hỏi:** Một vòng đời (chu trình) vòng lặp của đối tượng không có trạng thái khởi tạo cũng không có trạng thái kết thúc

▶ **Đáp:** Đúng, đối tượng được coi là đã luôn luôn tồn tại ở đây và sẽ còn mãi mãi tiếp tục tồn tại.

□ □ □



An Introduction to the Unified Modeling Language UML™

ICONIX is pleased to offer a one-day course titled "An Introduction to the Unified Modeling Language (UML)".

"An Introduction to the Unified Modeling Language (UML)" was developed by Kendall Scott, co-author of the award-winning book *UML Distilled*, as well as *Use Case Driven Object Modeling with UML*, and author of the forthcoming *UML for the Rest of Us*. The course is designed to help you quickly get up to speed on the essential aspects of the UML and the most relevant parts of the associated Unified Process.

This one-day course is designed to provide a quick but insightful introduction to the standard visual modeling language, touching upon all of the key aspects of the UML and its relationship with the use-case-driven, architecture-centric, and iterative and incremental Unified Process. The course also provides pointers about how to go about learning more about the UML, from the published work of the 'three amigos' as well as the instructor's own UML material.

The course is presented in six parts:

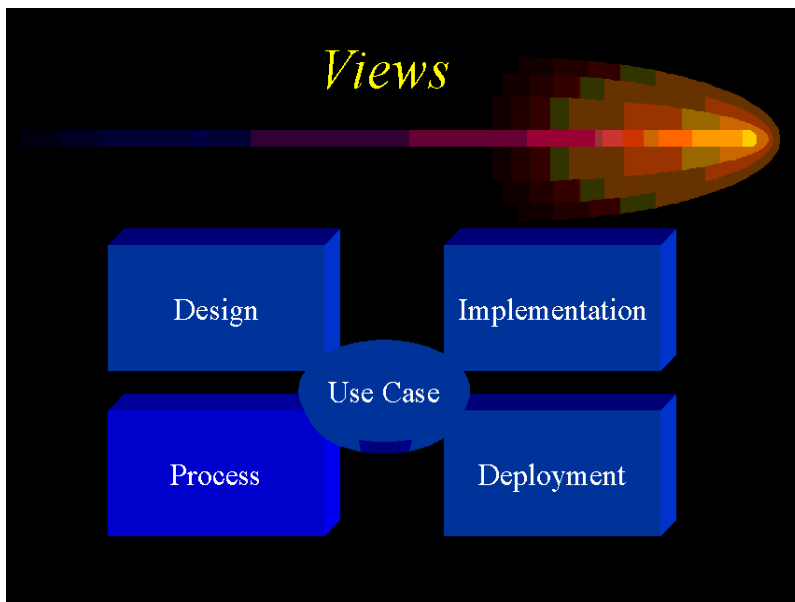
- **The Overview addresses the "mission statement" of the UML, how the language evolved from the work of the "three amigos," and principles of modeling.**

Reasons to Model

- to communicate the desired structure and behavior of the system
- to visualize and control the system's architecture
- to better understand the system and expose opportunities for simplification and reuse
- to manage risk

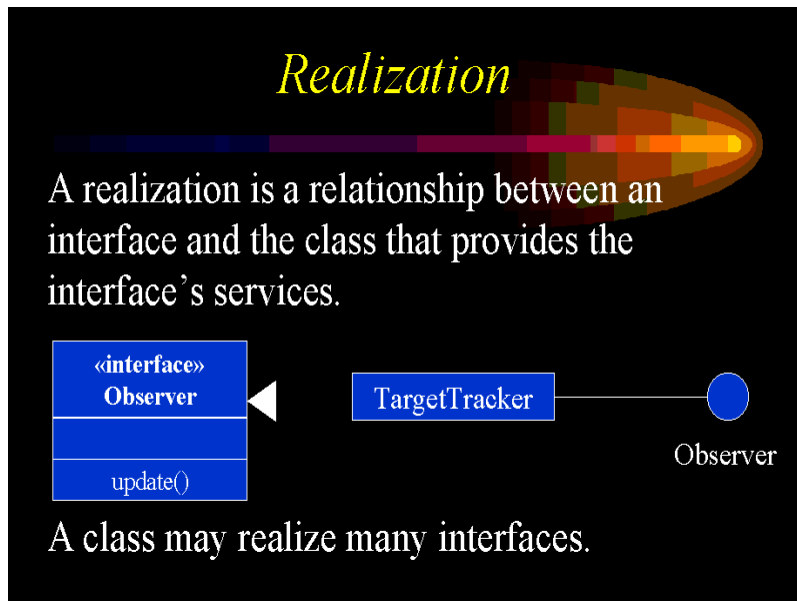
● **Views, Phases, and Diagrams** addresses the five architectural views around which the UML is centered, the four Unified Process phases to which the UML relate, and the nine diagrams at the heart of the UML.

Views

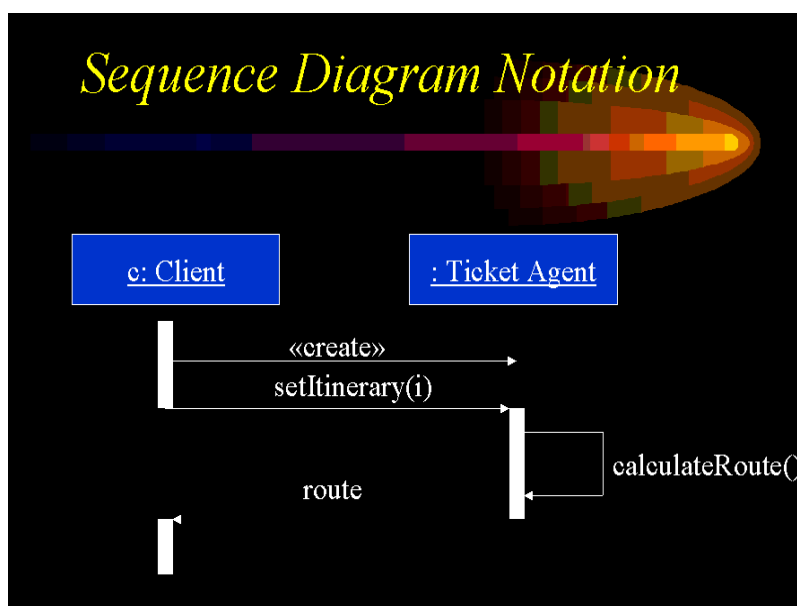


● **Structural (Static) Diagrams** addresses the four UML diagrams

that focus on the structural aspects of a system being modeled, as well as the non-standard but popular package diagram.

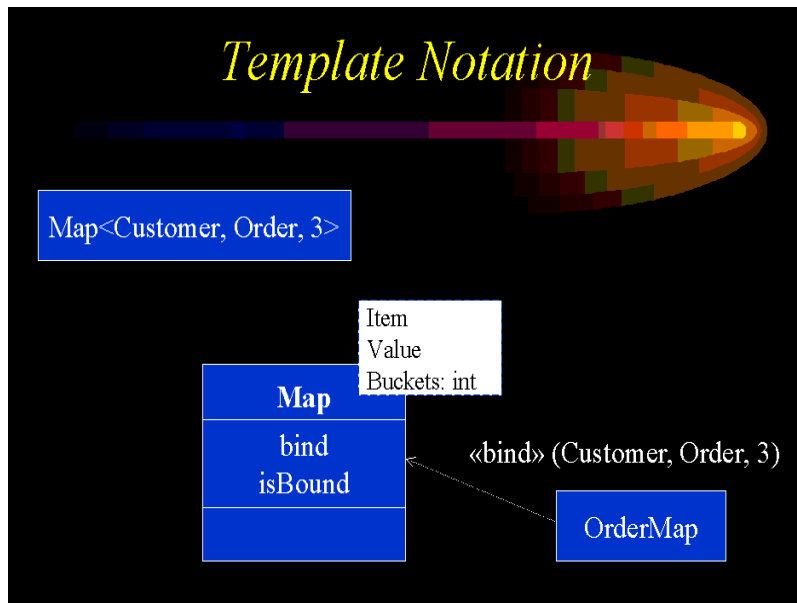


● **Behavioral (Dynamic) Diagrams** addresses the five UML diagrams that focus on the behavioral aspects of a system being modeled.

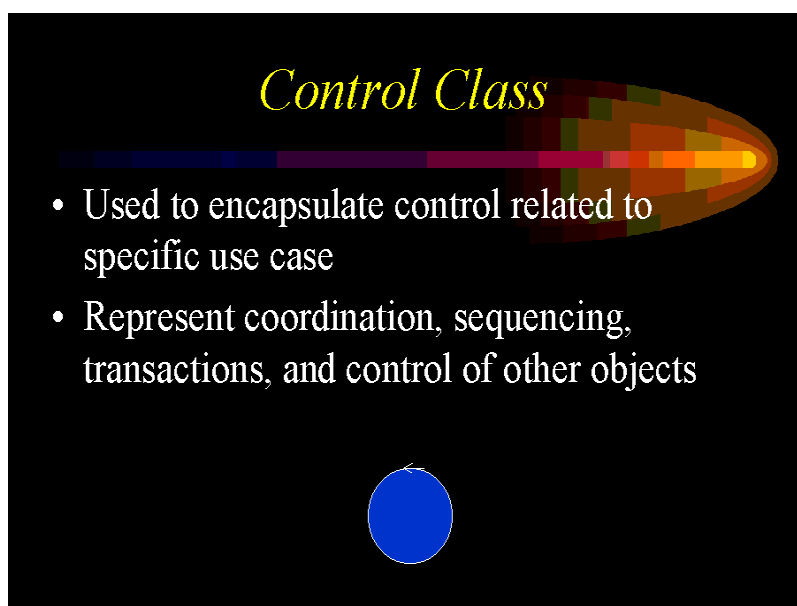


● **Odds and Ends** addresses areas of the UML, such as templates,

processes and threads, and collaborations, which cross the conceptual boundaries that the diagrams establish.



● **Process-Specific Extensions** addresses the class stereotypes that are required for robustness analysis (which is discussed extensively [Use Case Driven Object Modeling with UML](#)), and other Unified Process-related extensions associated with testing, use cases, and requirements.



"An Introduction to the Unified Modeling Language (UML)" is suitable for anyone who is interested in learning about the UML. No knowledge of object orientation is assumed; however, the instructor can customize the course as needed to make it more suitable for students who do have that knowledge. The flavor of the course that involves work with Rose assumes no experience working with visual modeling tools.

***UML is a trademark of Object Management Group, Inc. in the U.S. and other countries.**

ICONIX Software Engineering, Inc./2800 28th Street, Suite 320/Santa Monica, CA 90405/Tel (310)458-0092/Fax (310)396-3454/email: marketing@iconixsw.com



Please note that this site is no longer being updated. [Please click here to return to www.omg.org](http://www.omg.org).



What Is OMG-UML and Why Is It Important?

As the strategic value of software increases for many companies, the industry looks for techniques to automate the production of software and to improve quality and reduce cost and time-to-market. These techniques include component technology, visual programming, patterns and frameworks. Businesses also seek techniques to manage the complexity of systems as they increase in scope and scale. In particular, they recognize the need to solve recurring architectural problems, such as physical distribution, concurrency, replication, security, load balancing and fault tolerance. Additionally, the development for the World Wide Web, while making some things simpler, has exacerbated these architectural problems. The Unified Modeling Language (UML) was designed to respond to these needs.

The Unified Modeling Language (UML) is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems.

The UML was developed by Rational Software and its partners. It is the successor to the modeling languages found in the Booch, OOSE/Jacobson, OMT and other methods. Many companies are incorporating the UML as a standard into their development processes and products, which cover disciplines such as business modeling, requirements management, analysis & design, programming and testing.

The Importance Of Modeling
Developing a model for an industrial-strength software system prior to its construction or renovation is as essential as having a blueprint for a building.

Good models are essential for communication among project teams and to assure architectural soundness. As the complexity of systems increases, so does the importance of good modeling techniques. There are many additional factors of a project's success, but having a rigorous *modeling language* standard is essential.

A modeling language must include:

- Model elements — fundamental modeling concepts and semantics.
- Notation — visual rendering of model elements.
- Guidelines — idioms of usage within the trade.

In the face of increasingly complex systems, visualization and modeling become essential. The UML is a well-defined and widely accepted response to that need. It is the visual modeling language of choice for building object-oriented and component-based systems.

Goals of the UML

The primary goals in the design of the UML were as follows:

- 1) Provide users with a ready-to-use, expressive visual modeling language so they can develop and exchange meaningful models.
- 2) Provide extensibility and specialization mechanisms to extend the core concepts.
- 3) Be independent of particular programming languages and development processes.
- 4) Provide a formal basis for understanding the modeling language.
- 5) Encourage the growth of the OO tools market.
- 6) Support higher-level development concepts such as collaborations, frameworks, patterns and components.
- 7) Integrate best practices.

Scope of the OMG-UML

The Unified Modeling Language (UML) is a language for specifying, constructing, visualizing, and documenting the artifacts of a software-intensive system. First and foremost, the Unified Modeling Language fuses the concepts of Booch, OMT and OOSE. The result is a single, common, and widely usable modeling language for users of these and other methods.

Second, the Unified Modeling Language pushes the envelope of what can be done with existing methods. As an example, the UML authors targeted the modeling of concurrent, distributed systems to assure that the UML adequately addresses these domains.

Third, the Unified Modeling Language focuses on a standard modeling language, not a standard process. Although the UML must be applied in the context of a process, experience has shown that different organizations and problem domains require different processes. (For example, the development process for shrink-wrapped software is an interesting one, but building shrink-wrapped software is vastly different from building hard-real-time avionics systems upon which lives depend.) Therefore, the efforts concentrated first on a common metamodel (which unifies semantics) and second on a common notation (which provides a human rendering of these semantics). The UML authors promote a development process that is *use-case driven*, *architecture centric*, and *iterative and incremental*.

Outside The Scope of the UML

While the UML aims to simplify and standardize modeling it is not an all encompassing language. This gives it the flexibility to be used to design a variety of systems over a wide spectrum of industries. Some major areas outside of the scope of the UML include:

Programming Languages

The UML, a visual *modeling* language, is not intended to be a visual *programming* language, in the sense of having all the necessary visual and semantic support to replace programming languages. The UML is a language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system, but it does draw the line as you move toward code. The UML does have a tight mapping to a family of OO languages, so that you can get the best of both worlds

Tools

Standardizing a language is necessarily the foundation for tools and process. The primary goal of the OMG RFP was to enable tool interoperability. However, tools and their interoperability are very dependent on a solid semantic and notation definition, such as the UML provides. The UML defines a *semantic* metamodel, not a tool *interface*, *storage*, or *run-time* model, although these should be fairly close to one another.

Process

Many organizations will use the UML as a common language for their project artifacts, but they will use the same UML diagram types in the context of different processes. The UML is intentionally process independent, and defining a standard process was not a goal of the UML or OMG's RFP.

Origin of UML and How It Became An OMG Standard

Identifiable object-oriented modeling languages began to appear between mid-1970 and the late 1980s as various methodologists experimented with different approaches to object-oriented analysis and design. The number of identified modeling languages increased from less than 10 to more than 50 during the period between 1989-1994. Many users of OO methods had trouble finding complete satisfaction in any one modeling language, fueling the "method wars." By the mid-1990s, new iterations of these methods began to appear and these methods began to incorporate each other's techniques, and a few clearly prominent methods emerged.

The development of UML began in late 1994 when Grady Booch and Jim Rumbaugh of Rational Software Corporation began their work on unifying the Booch and OMT (Object Modeling Technique) methods. In the Fall of 1995, Ivar Jacobson and his Objectory company joined Rational and this unification effort, merging in the OOSE (Object-Oriented Software Engineering) method.

As the primary authors of the Booch, OMT, and OOSE methods, Grady Booch, Jim Rumbaugh, and Ivar Jacobson were motivated to create a unified modeling language for three reasons. First, these methods were already evolving toward each other independently. It made sense to continue that evolution together rather than apart, eliminating the potential for any unnecessary and gratuitous differences that would further confuse users. Second, by unifying the semantics and notation, they could bring some stability to the object-oriented marketplace, allowing projects to settle on one mature modeling language and letting tool builders focus on delivering more useful features. Third, they expected that their collaboration would yield improvements in all three earlier methods, helping them to capture lessons learned and to address problems that none of their methods previously handled well.

As they began their unification, they established four goals to focus their efforts:

- Enable the modeling of systems (and not just software) using object-oriented concepts.
- Establish an explicit coupling to conceptual as well as executable artifacts.
- Address the issues of scale inherent in complex, mission-critical systems.

- Create a modeling language usable by both humans and machines.

The efforts of Booch, Rumbaugh, and Jacobson resulted in the release of the UML 0.9 and 0.91 documents in June and October of 1996. During 1996, the UML authors invited and received feedback from the general community. They incorporated this feedback, but it was clear that additional focused attention was still required.

While Rational was bringing UML together, efforts were being made on achieving the broader goal of an industry standard modeling language. In early 1995, Ivar Jacobson (then Chief Technology Officer of Objectory) and Richard Soley (then Chief Technology Officer of OMG) decided to push harder to achieve standardization in the methods marketplace. In June 1995, an OMG-hosted meeting of all major methodologists (or their representatives) resulted in the first worldwide agreement to seek methodology standards, under the aegis of the OMG process.

During 1996, it became clear that several organizations saw UML as strategic to their business. A Request for Proposal (RFP) issued by the Object Management Group (OMG) provided the catalyst for these organizations to join forces around producing a joint RFP response. Rational established the UML Partners consortium with several organizations willing to dedicate resources to work toward a strong UML 1.0 definition. Those contributing most to the UML 1.0 definition included: Digital Equipment Corp., HP, i-Logix, IntelliCorp, IBM, ICON Computing, MCI Systemhouse, Microsoft, Oracle, Rational Software, TI, and Unisys. This collaboration produced UML 1.0, a modeling language that was well defined, expressive, powerful, and generally applicable. This was submitted to the OMG in January 1997 as an initial RFP response.

In January 1997 IBM, ObjecTime, Platinum Technology, Ptech, Taskon, Reich Technologies and Softeam also submitted separate RFP responses to the OMG. These companies joined the UML partners to contribute their ideas, and together the partners produced the revised UML 1.1 response. The focus of the UML 1.1 release was to improve the clarity of the UML 1.0 semantics and to incorporate contributions from the new partners. It was submitted to the OMG for their consideration and adopted in the fall of 1997.

UML Present and Future

The UML is nonproprietary and open to all. It addresses the needs of user and scientific communities, as established by experience with the underlying methods on which it is based. Many methodologists, organizations, and tool vendors have committed to use it. Since the UML builds upon similar semantics and notation

from Booch, OMT, OOSE, and other leading methods and has incorporated input from the UML partners and feedback from the general public, widespread adoption of the UML should be straightforward.

There are two aspects of "unified" that the UML achieves: First, it effectively ends many of the differences, often inconsequential, between the modeling languages of previous methods. Secondly, and perhaps more importantly, it unifies the perspectives among many different kinds of systems (business versus software), development phases (requirements analysis, design and implementation), and internal concepts.

Although the UML defines a precise language, it is not a barrier to future improvements in modeling concepts. We have addressed many leading-edge techniques, but expect additional techniques to influence future versions of the UML. Many advanced techniques can be defined using UML as a base. The UML can be extended without redefining the UML core.

The UML, in its current form, is expected to be the basis for many tools, including those for visual modeling, simulation and development environments. As interesting tool integrations are developed, implementation standards based on the UML will become increasingly available.

The	Meta	Object	Facility
------------	-------------	---------------	-----------------

The main purpose of the OMG MOF is to provide a set of CORBA interfaces that can be used to define and manipulate a set of interoperable metamodels. The MOF is a key building block in the construction of CORBA-based distributed development environments.

The Meta Object Facility represents the integration of work currently underway by the OMG members in the areas of object repositories, object modeling tools, and meta data management in distributed object environments. The MOF specification uses the Unified Modeling Language (UML) notation. The facility interface and semantics incorporate some of the advanced meta data management concepts that have been implemented in the commercial object repositories, development tools, and object framework products developed by the co-submitters.

The specification enhances meta data management and meta data interoperability in distributed object environments in general and in distributed development environments in particular. While the initial work addresses meta data interoperability in object analysis and design domain, it is anticipated that the MOF will be rich enough to support additional domains. Examples include metamodels that cover the application development life cycle as well as

additional domains such as data warehouse management and business object management. OMG is expected to issue new RFPs to cover these additional domains.

Copyright © 1997-2001 Object Management Group, Inc.
All Rights Reserved.

For questions about the WEBSITE, please contact
webeditor@omg.org

For TECHNICAL questions, please contact
webtech@omg.org

Object Oriented Analysis and Design Using UML

A Whitepaper by Mark Collins-Cope of Ratio Group.

Introduction

You're proficient in C++, Java or another OO language, you're designing class hierarchies, using inheritance, and manipulating complex pointer relationships to store the necessary links between your classes. You've probably drawn blobs (representing classes in some way) on whiteboards, with connecting lines to indicate the relationships between classes (inheritance or other). Perhaps you're feeling the need for a more formal notation to express your designs - using something that is language independent, and that enables you to consider the important aspects of design leaving the detail for later.

Alternatively, perhaps you're a Project Manager, looking to formalise the OO design process a little to make sure you're getting the most from your move to C++/Java or a similar language.

In this paper, I take a look at the UML (Unified Modelling Language) notation for Object Oriented Analysis and Design - the emerging standard designed by Booch, Rumbaugh and Jacobson, each of whom previously had their own notations published independently.

The starting point is Object Modelling, a technique that enables you to focus on class structure, inheritance, etc., whilst avoiding language specifics such as pointer dereferencing.

Object Modelling In UML

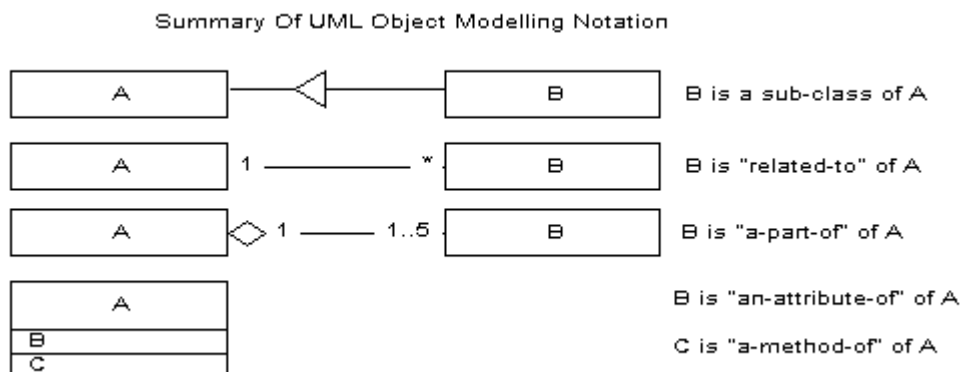


Figure 1 - Subset of UML Object Modelling Notation - A Summary

Object Modelling is the central technique in UML. It is a language independent notation allowing the specification of classes, their data or attributes(private) and methods (public), inheritance, and other more general relationships between classes. The notation itself is fairly simple to grasp (see figure 1), however this hides the somewhat more subtle thought processes underlying a good model.

The best way to understand the notation is to look at an example. The following Object Model shows a simple Banking System, containing classes for Head-Office, Branch, Accounts held at that Branch, and the Customers who the Accounts belong to:

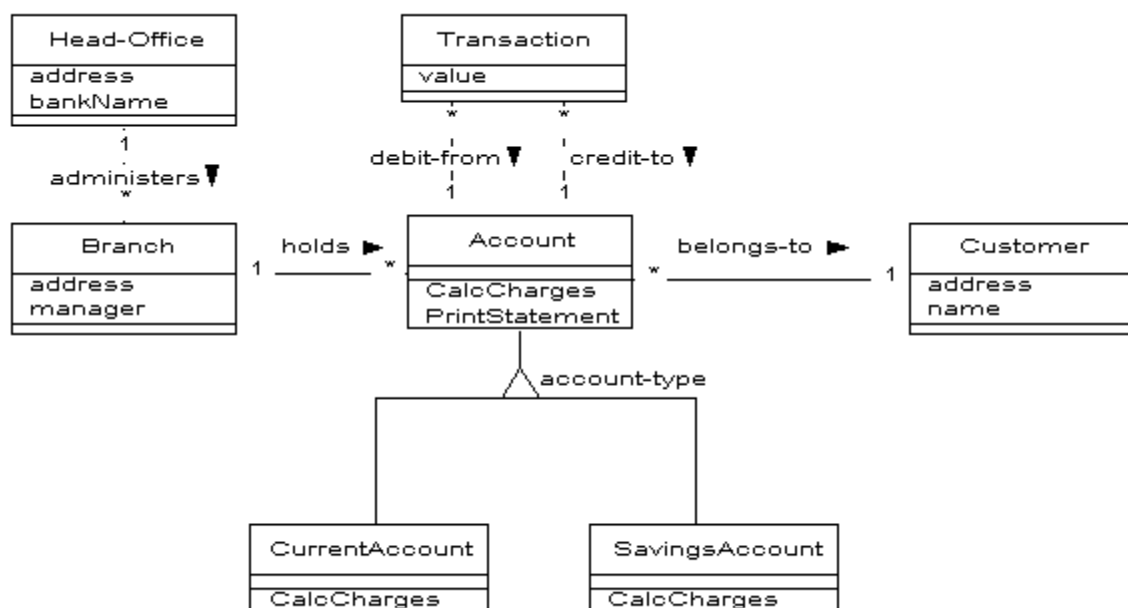


Figure 2 - A Simple Banking System Object Model

Examining this Object Model in more detail, we can see the following information about our class structure:

- A **Head-Office** class (containing "bankName" and "address" fields, otherwise known as attributes) "administers" an (unspecified) number of **Branch** classes; whilst a **Branch** is "administered-by" exactly one **Head-Office** (the little black arrows indicates the direction in which the name given to a relationship should be read). On the diagram this relationship is represented by the line from the **Head-Office** class to the **Branch** class which is labelled "administers". The "1" at the **Head-Office** end of the line shows that exactly one **Head-Office** is associated with each **Branch** (as you would expect). The "*" at the **Branch** end of the line shows that a **Head-Office** "administers" many **Branches** - again as you would expect.

- Similarly, a Branch class (which contains "manager" and "address" attributes) "holds" many Account classes; whilst each Account class "is-held-by" exactly one Branch. We can also see that we have determined that an Account class has a "CalcCharges" method (also known as operations or member functions) defined. This method, when invoked, will look at the detail stored within the Account object, and apply the appropriate (undoubtedly extortionate) charges to the Account. The second method - "PrintStatement" - will take the details of the Account and print them out.
- The inheritance "triangle" (labelled "account-type") shows us that our system knows about three types of Account: the basic account (in this case a virtual class called Account), and two specialised accounts - the CurrentAccount and SavingsAccount - which are derived from Account. The fact that the "CalcCharges" is shown in both sub-classes indicates that its implementation is re-defined for these classes (in C++ terms it is a virtual function). This is indicative of the fact that charges on a "SavingsAccount" are calculated in a completely different manner to charges on a "CurrentAccount".
- Implicit in the decision to use inheritance and redefine methods in sub-classes is the fact that the system, when implemented, will use the polymorphism features of the target language (C++, Java...) to enable all Accounts to be treated in a single coherent fashion, regardless of the particular charges mechanism involved. This is of course one of the reasons we use an object-oriented development language in the first place.
- Each Account "belongs-to" exactly one owner - the Customer class on the diagram. Customers, on the other hand, may have many Accounts.

It's worth noting here that because an Account may "belong-to" a Customer, both CurrentAccounts and SavingsAccounts may also belong to a Customer. In other words, the "belongs-to" relationship between Accounts and Customers is inherited by the CurrentAccount and SavingsAccount classes. This fact simplifies the diagram considerably, removing the need for these relationships to be noted explicitly. This simplification will also be apparent in our final implementation of the system.

- Finally, you can see that there are **two** relationships shown between the Account and the Transaction classes. This is because, in our banking system, each individual transaction (credit, debit, etc.) must have two associated accounts - the Account the money is "debit(ed)-from", and the

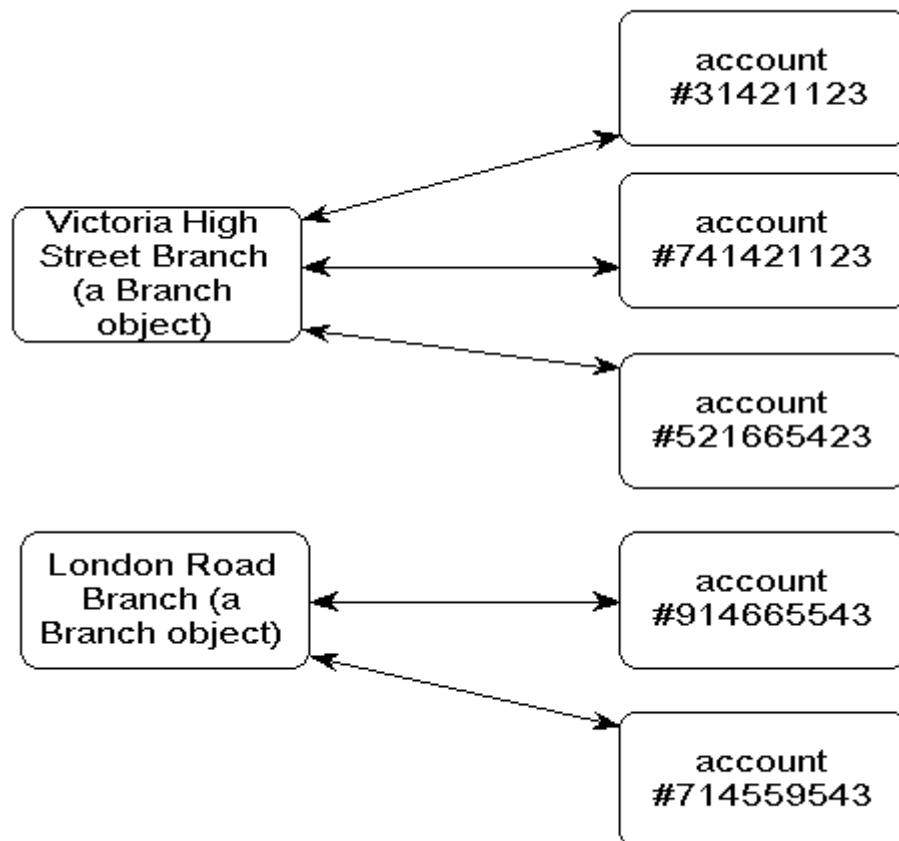
Account the money is "credit(ed)-to". This enables the bank to record exactly where each transaction has come from, and gone to, so to speak.

These last point brings out an interesting feature of what is being shown on an Object Model: clearly it wouldn't make sense for each Transaction to be "debit(ed)-from" and "credit(ed)-to" the same Account - no money would be transferred! Obviously, although the lines (relationships) are shown to the same Account class, they do not (necessarily) represent links to the same Account object at run-time.

A relationship shown on an Object Model indicates that some kind of run-time link will exist between two instances of the classes shown on the Object Model. Thus the Branch to Accounts relationship should be read as follows:

An instance of the class Branch will be linked to (zero to) many instances of the class Account, whilst an instance of the class Account will be linked to (one and only) one instance of the class Branch.

This can be shown more clearly by the following instance diagram (instance diagrams are used to assist in understanding and clarifying Object Models - they also give quite a hint as to how relationships can be implemented in



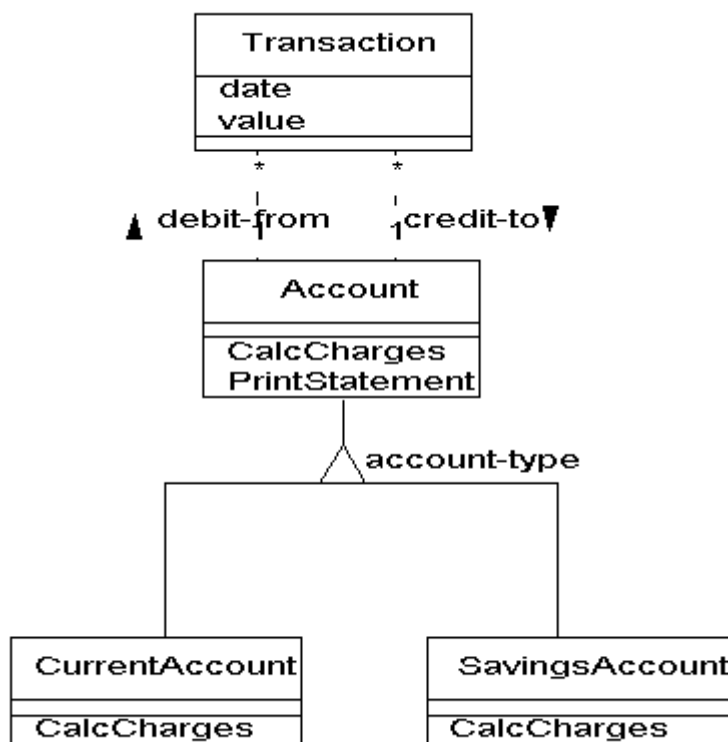
C++!):

Figure 3 - Instance Diagram Showing Branch and Account objects

By now, you may be beginning to see how Object Models can assist the analysis/design process. They assist in the clarification of the relationships that should be (somehow) represented in a software system. The important point to note here is that we are first working out what relationships we need to represent in our system ("belongs-to", etc.), without worrying too much about exactly how they should be stored. Put another way, Object Modelling allows us to focus on exactly what problem we are trying to solve, before we look at the best way of implementing our model in a particular programming language.

Implementing Object Models

OK, that's fine, you may say, but how do Object Models relate to C++ or Java, exactly? Let's take a look at a sub-set of our previous example



:

Figure 4 - Subset of Banking Model

Our Object Model shows us that we need four classes: Transaction; Account; Current Account and Savings Account, and that our implementation must enable us to represent the fact that any particular Account has two sets of Transactions associated with it - which will be needed to implement the PrintStatement

method. The Account, CurrentAccount and SavingsAccount classes are easily mapped to the C++ (or Java) inheritance mechanisms:

```

class Account {
/*... data... */
public:
virtual void CalcCharges();
virtual void PrintStatement();
};

class SavingsAccount: public Account {
/* any additional data */
public:
virtual void CalcCharges(); /* re-definition */
/* use the base class PrintStatement method */
};

class SavingsAccount: public Account {
/* any additional data */
public:
virtual void CalcCharges(); /* another re-definition */
/* use the base class PrintStatement method */
};

```

Figure 5 - Mapping Object Model Inheritance To C++ Inheritance

The Transaction class may be implemented as follows:

```

class Transaction {
long value; /* stored in pence */
date_t date; /* date of transaction */
public:
/* Access and Update functions */
Date(date_t); /* set */
date_t Date(); /* get */
Value(long); /* set */
long Value(); /* get */
};

```

Figure 6 - Transaction Class In C++

This leaves us with the "debit-from" and "credit-to" relationships to be implemented. Here we have a number of choices: linked-lists; collection-classes; (dynamically bounded) arrays of pointers; etc. could all be used to represent these relationships.

```

class TransactionList {
    Transaction * next; /* ptr to next element */
    Transaction * data; /* store the transaction here */
public:
    void Data(Transaction *); /* set */
    Transaction * Data(); /* get */
    void NextItem(TransactionList *); /* set next ptr */
    TransactionList * NextItem(); /* get next ptr */
};

```

Figure 7 - Simple Transaction List Handler Class

For brevity, a linked-list class with a somewhat limited interface is used in this example - although this may not be the best choice. Amending our Account class definition to use this class gives us the following new definition:

```

class Account {
    TransactionList * debitedFrom; /* debited from Tx list */
    TransactionList * creditedTo; /* credited to Tx list */
public:
    virtual void CalcCharges();
    void PrintStatement();

    /* some new methods to manipulate the Transaction list */
    DebitTx(Transaction *); /* Add a debit Tx */
    CreditTx(Transaction *); /* Add a credit Tx */
    Transaction* NextDebitTx(); /* Iterator: get debit Tx */
    Transaction* NextCreditTx(); /* Iterator: get cred Tx */
};

/* sample method implementation */
Account::DebitTx(Transaction * theTx) {
    /* add a new list contained at the beginning of the list */
    TransactionList * tmpTxLp = debitedFrom;
    debitedFrom = new TransactionList;
    debitedFrom->NextItem(tmpTxLp);

    /* new put the transaction data into the list */
    debitedFrom->Data(theTx);
};

```

Figure 8 - Account Class amended to use Transaction List

Although this is a somewhat simplistic implementation - it demonstrates the point that the model shown in figure 4 is easily translated into C++. Of course, better implementations of the "debit-from" relationship are possible, but the fact

that the Account class interface completely hides the underlying implementation of this relationship means that we can improve on our first cut implementation at a later date with little impact on our overall system code. In other words, use of the Account class interface has limited the impact of the relationship implementation method: something we strive to achieve when writing OO based applications.

A couple of other points are worthy of note at this stage:

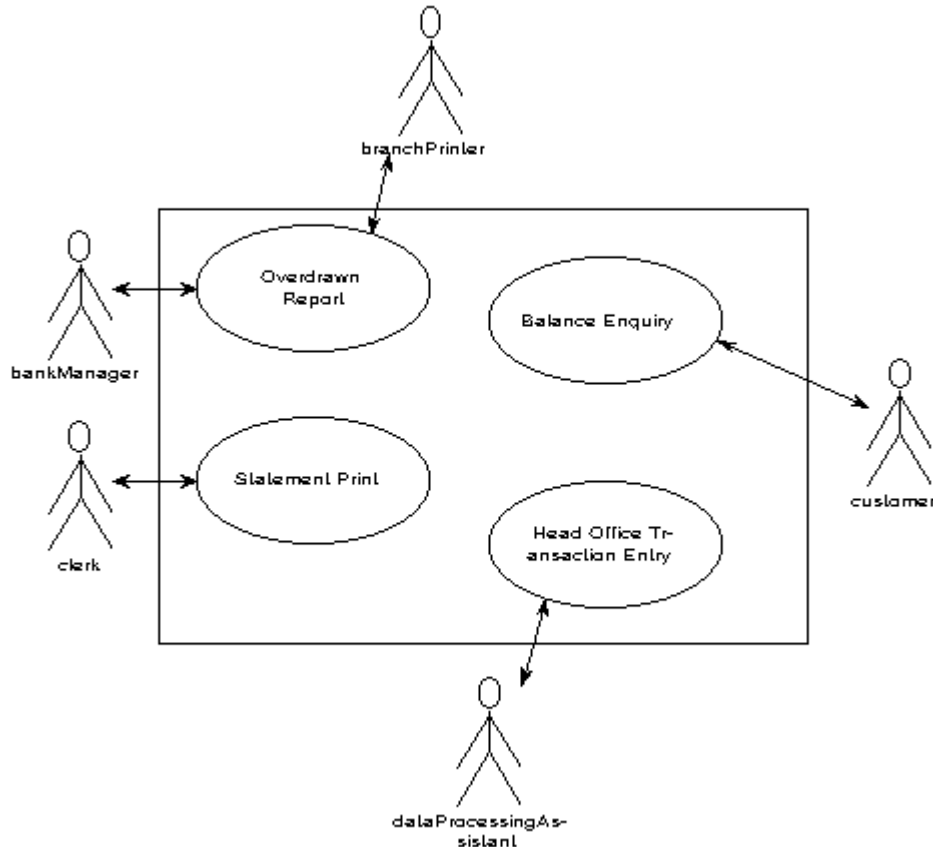
- The linked list class contains pointers (references in Java) to Transaction objects. This is implicit in our Object Model, and is what the system's users would expect to see. To see why, consider the case when a new Transaction value is entered in error. The Transaction is linked to two accounts ("debit-from" and "credit-to"). If the Transaction object is shared, only one object need be modified to rectify the situation. Using two objects would either mean that either the system has to update two objects (equals more coding work), or that the user has to update two Transactions (equals greater potential for mistakes).
- Although our Object Model "debit-from" relationship uses a linked list, there are many alternatives to this choice - including the use of a relational database to underpin the system. The point is, however, no matter what mechanism is used, we are actually trying to implement a "many-to-one" relationship between an Account and a Transaction. It is this relationship that exists in the banking problem domain - not a relationship involving linked lists or collection classes. Object Modelling enables us to spot the relationship required by the problem domain, and then choose the best way of implementing it.
- So far, we have only implemented the "debit-from" relationship in one direction- from the Account class to the Transaction class. Our model does not (yet) specify in which direction the relationship will be traversed. If we need to traverse the relationship in both directions - getting from the Transaction to the related Account - our implementation will prove insufficient, and some form of double pointer schema may be needed. Much work would have been saved if we had known this fact before we had started writing the code.
- Although our Object Model provided a starting point for our implementation, it was not complete, for example new methods have been added to the Account class.

- Other factors may also influence our choice of implementation: do we need a direct form of access - for example using a Transaction number to go directly from the Account to the relevant Transaction? If we do, then a linked-list will prove an inefficient choice of implementation. Again, it would be very useful to know this type of information before we start trying to implement the relationship.

From these points we can see that we need to consider the wider requirements of our system before we can come up with the right implementation of our "debit-from" relationship (not to mention the many other classes and relationships that might be required). We can't produce a good design for a system unless we consider all the required functionality - in detail. Use Cases provide the mechanism for doing this.

Use Cases In UML

Use Cases are used to document system requirements. They provide a useful technique which, in conjunction with Object Modelling, help us to clarify exactly what the system is supposed to do. Let's take a look at the requirements for our banking system



:

Figure 9 - Use Cases for Banking System

This Use Case diagram shows us the following:

- The required business functions - that is, the type of operation you'd expect to find on the menu of the application once it had been developed. In this case we have identified the following functions:
- Bank managers need to periodically print out a report detailing all the customers who are overdrawn; these appear on the branch printer
- Customers may use the system for balance enquiries
- Data processing staff use the system to do basic data entry (transactions on accounts)
- Clerks may periodically request statements on behalf of Customers;
- There are four distinct types of user of the system: Bank Managers; Clerks; Data Processing Assistants; and Customers. Each type of user typically has their own particular set of requirements for a system: hence identify user types assists in identifying all the required system functions.

The Use Case diagramming technique allows us to make a first cut at defining the system requirements, and will help us in presenting these requirements back to the users of the system. It also partitions the system into single atomic business functions, which may be used as a basis for costing the system, or for planning a phased system delivery. In this case each successive phase would deliver further batches of Use Cases. Further information is still required, however, to tie down the detail of what each business function does. Use Case Detail provides this information (explanatory notes are shown in bold):

Use Case Detail: Overdrawn Report

Used

By:

Bank Manager

Inputs:

Details what information flows from the user to the system for this particular Use Case.

theBranchSortCode - The Sort Code of the branch for which the report is required.

theOverdraftPeriod - how long an Account has been overdrawn before it is forms

part of the report.

Outputs:

Details what information flows from the system to the external environment, in this case the printer!
overdraftReport (to branchPrinter) - structured as follows: customer name; current overdraft; period overdrawn (days);
Printed for all accounts that have been overdrawn for a period greater than theOverdraftPeriod, and which have not already been reported (on another report) in the last 30 days.

Pre-Conditions:

What validity checks or constraints apply on the inputs (or the internal system as a whole, in some cases).
theBranchSortCode - must be a branch sort code held within the system.
theOverdraftPeriod - must be a number between 0 and 100 days.

Post-Condition:

What changes does the Use Case make to the internal system state.
Updates the reportedOnDate field of overdrawn accounts.

As work progresses on the Use Cases, the requirements of the system become clearer enabling the Object Model to be updated in parallel, helping us make sure our model (and the subsequent implementation in the chosen language) contains all the necessary classes and class inter-links. Whilst we're nearer to resolving some of the issues identified at the end of the discussion of implementing Object Models, a number are still outstanding: we still can't be sure in what direction the relationships must be implemented, whether we have identified all the methods; or what implementation of the links will best suit the use to which they'll be put. To sort out the remaining issues we'll need to look in more detail exactly how each Use Case will be implemented, using Sequence Diagrams.

Sequence

Diagrams

Sequence diagrams, performed on a per Use Case basis, examine the flow of method call calls within a system.

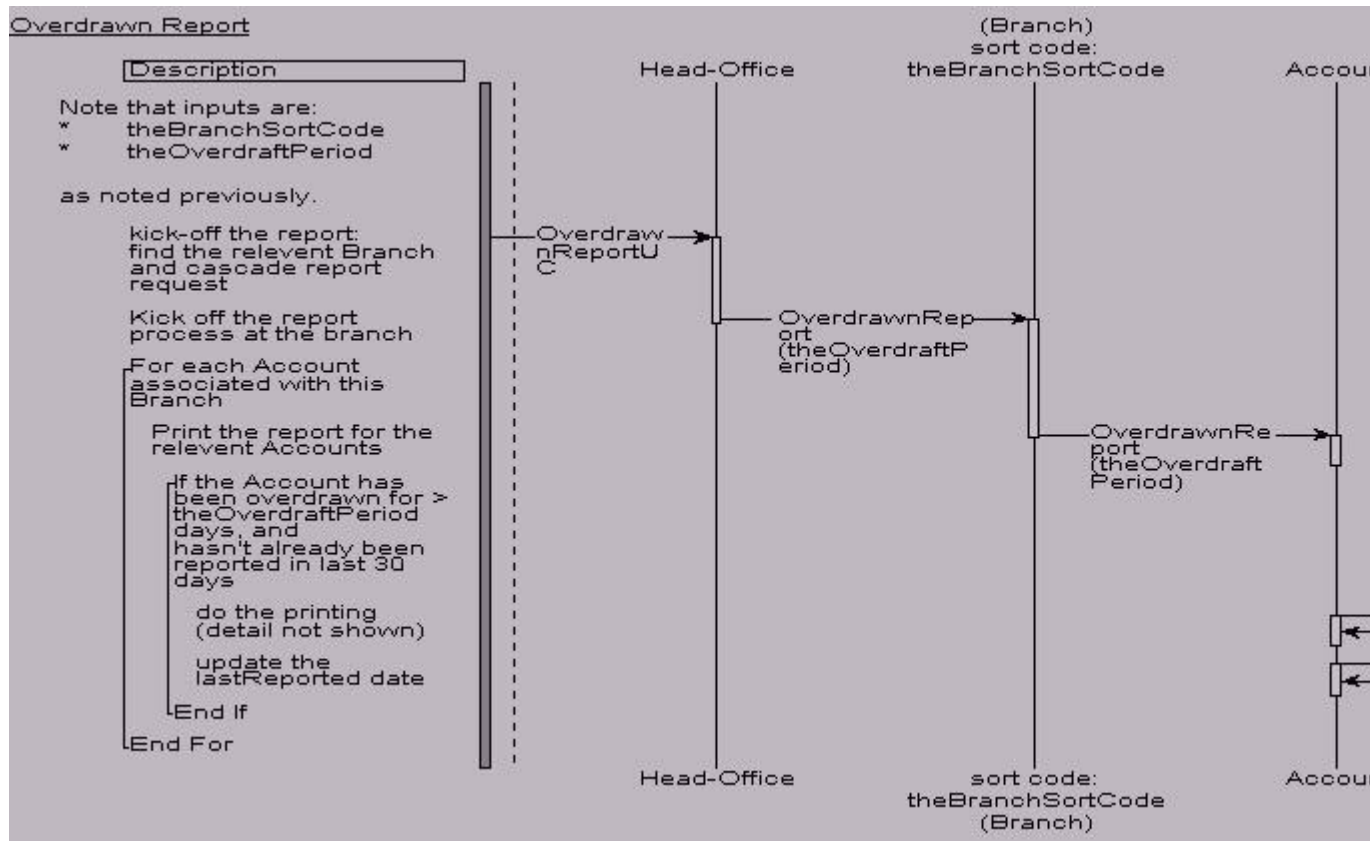


Figure 10 - Sequence Diagram for Overdrawn Report

Performing a complete analysis requires that each individual Use Case must be examined, although in practise only selected Use Cases may be examined. The Sequence Diagram in figure 10 shows the Overdrawn Report Use Case defined earlier.

The Overdrawn Report Use Case is thus implemented as follows:

- The Head-Office object (there is only one of them) has methods which correspond to each Use Case - in this case an OverdrawnReport method (this is a convenience for brevity, normally there would be a single "InitialObject" which the system would know about, and which would provide the entry point into the run-time model for all code).
- The Head-Office OverdrawnReport method locates the relevant Branch (as determined by the Use Case input: theBranchUseCase) and cascades the request to the Branch by calling its OverdrawnReport method.
- The Branch object in turn passes the request on down to each Account it holds (using the Account's OverdrawnReport method)!
- Each Account then:

- checks if it has been overdrawn for greater than the period specified by theOverdraftPeriod, which is passed as an argument to the Account.OverdrawnReport method (the detail of this is not shown - but involves summing up all the Transactions it holds, and checking the date on which it last became overdrawn).
- if appropriate it then calls one of its own methods to print the report (detail not shown), and resets its lastReportDate attribute, again using its own method.
- The method calls unwind until the Use Case is complete.

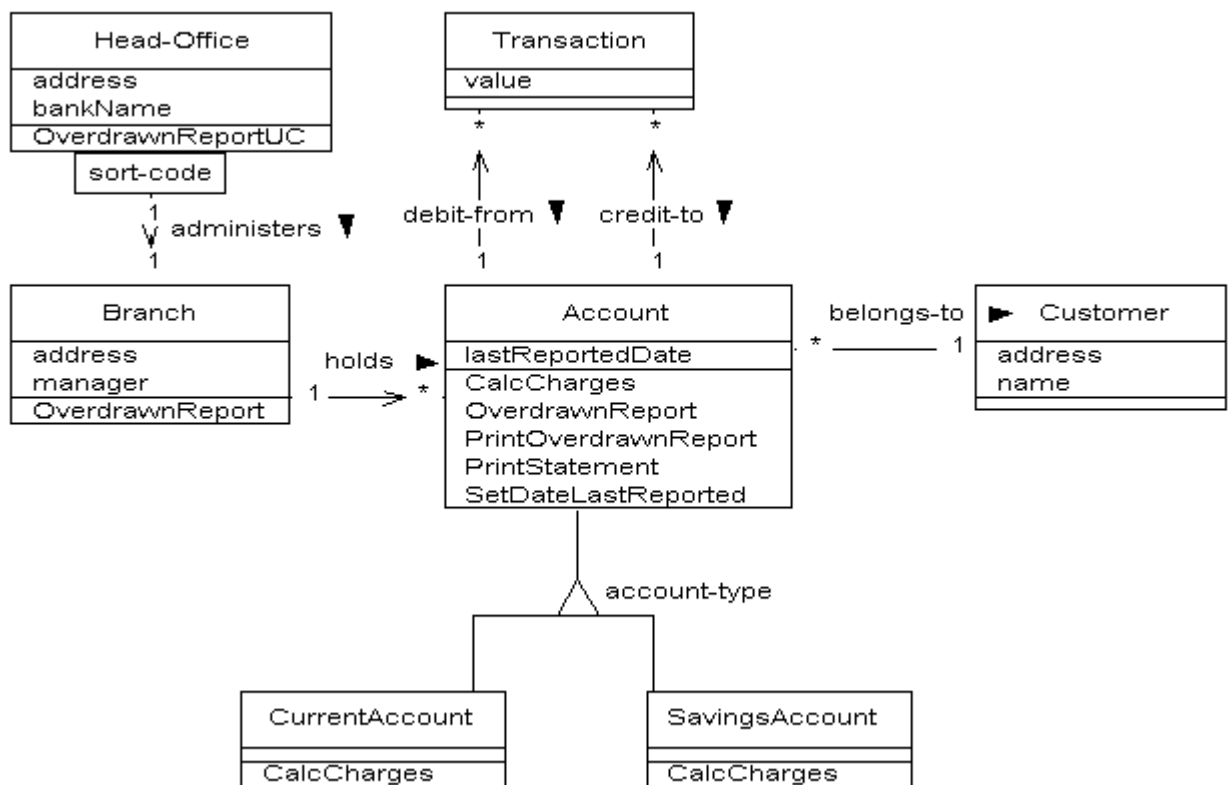


Figure 11 - Updated Banking System Object Model

Reviewing the Object Model (see figure 11), we can see a number of additions as a result of completing this Sequence Diagram:

- OverdrawnReport methods have been added to the Branch and Account classes.
- A lastReportedDate attribute and associated methods have been added to the Account class, along with a printOverdrawnReport method.

- The "administers" relationship between Head-Office and Branch has been qualified to indicate that "direct access" via the Branch's "sort-code" is required across the link (thus assisting in link design) - note the consequent change in the multiplicity of the relationship from many-to-one to one-to-one.
- We have added directionality to many of the links (e.g. see the arrow-head on the Branch to Account link).

Of course, we've only looked at one Use Case, so its likely the model will change further as more sequence diagrams are developed.

The Overall Process

From the above discussion we can see that Use Cases and Sequence Diagrams both add to integrity and completeness of our Object Model, and that a good Object Model provides a firm foundation for a good design, and hence a good implementation of the system.

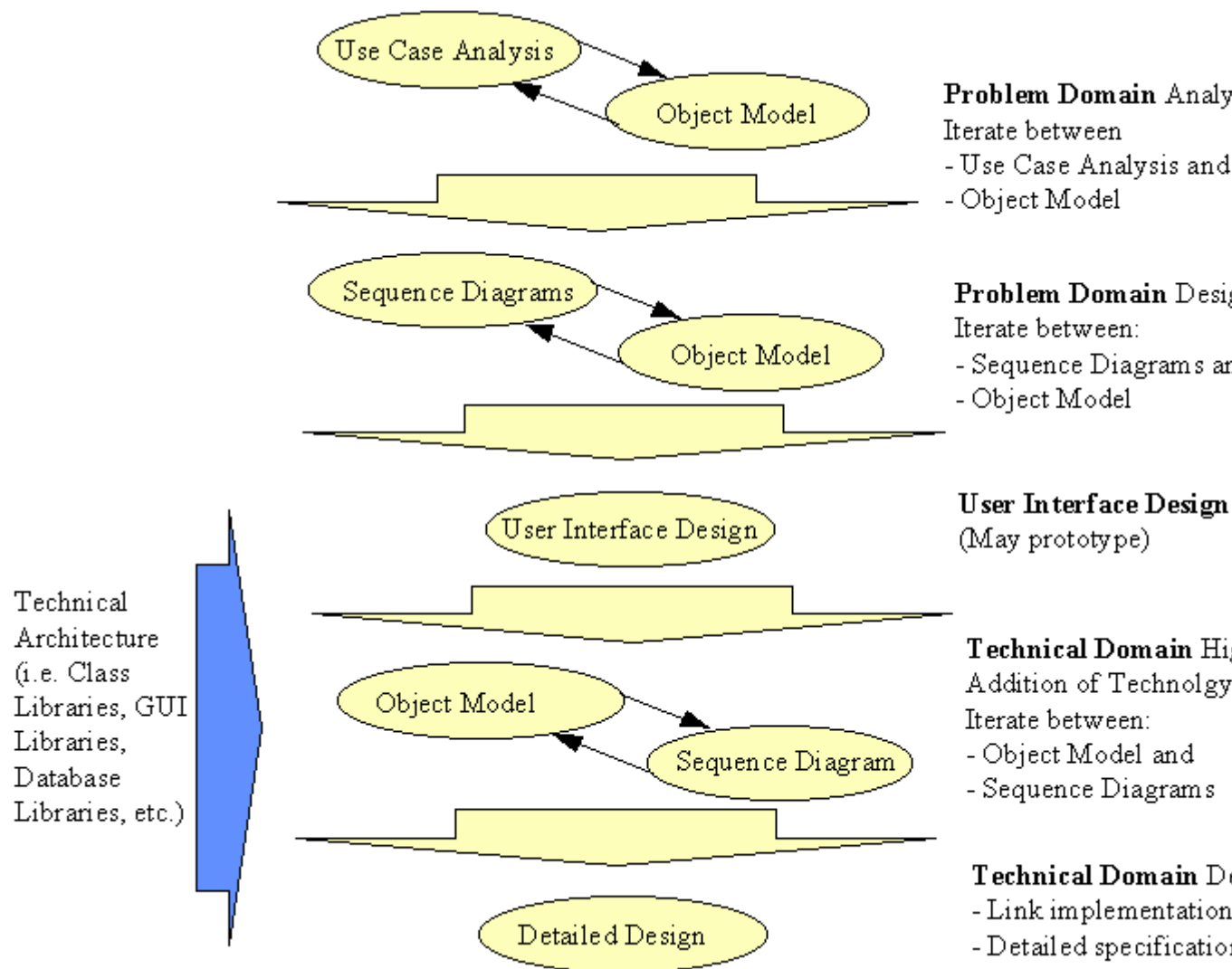


Figure 12 - The Overall Process

This approach separates the Problem and Technical Domain aspects of the project:

- Problem Domain Analysis is concerned with capturing requirements and producing a first cut Object Model. Typically the Object Model will be incomplete, having only a subset of the class attributes and methods defined.
- Problem Domain Design is concerned with finalising the detail of the problem domain parts of the Object Model, and results in an Object Model with a complete set of Problem Domain specific classes, attributes and methods.
- User Interface Design is the first step that focuses on the Technical Domain aspects of the problem, and involves taking the Use Cases as

defined earlier, and designing a Graphical User Interface appropriate to the Technical Architecture chosen for the project (MS Windows, X/Motif, etc.). Typically you would expect to find one controlling dialog box (which may use other subsidiary dialogs) for each Use Case in the system. Some prototyping may be appropriate at this point in the project. For small projects, prototyping and UI design may be undertaken in parallel with Use Case development.

- Technical Domain, High Level Design focuses on adding classes to meet the technical needs of the project, and is driven by the technical architecture of the project. Classes may be GUI related, DBMS (object or relational) related, distribution related (CORBA, DCOM, etc.), external systems related, or may provide an interface to internal system components such as printers. Previous Sequence Diagrams may be updated to confirm the validity of the technical design - in particular you would expect to see GUI classes appearing between the System Boundary and the Problem Domain classes.
- Finally, Detailed Technical Design, looks at link implementations, detailed data typing of attributes, full specification of all methods (including parameters), etc. The end result is a complete design of the full system.

The separation between Problem Domain and the Technical Domain aspects of the system is useful in large projects, allowing the focus of those working on the project to be clearly divided, as summarised in figure 13:

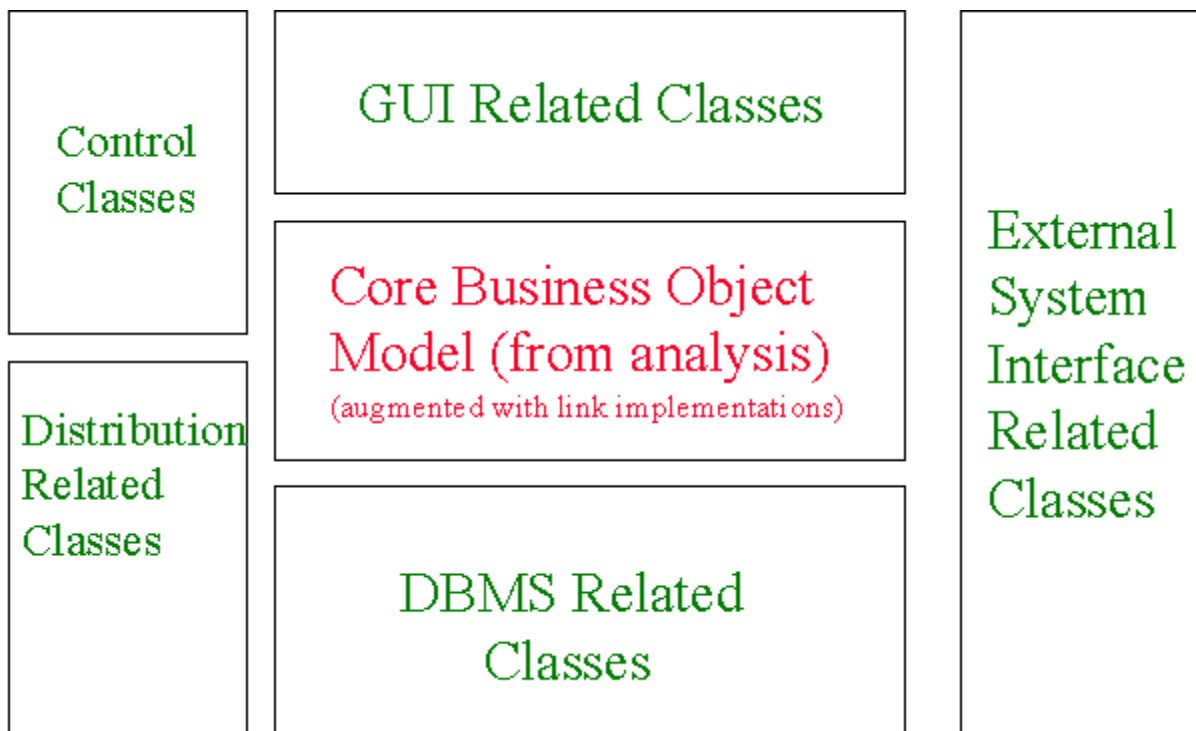


Figure 13 - Separation Of Problem and Technical Domain Components of a System

For smaller projects (one or two persons for a couple of months) the two domains may be merged, if desired. As mentioned previously, Use Cases may be used in phasing a project; the process shown earlier does not prohibit this. A project with 50 Use Cases could be structured in three phases as shown in figure 14:

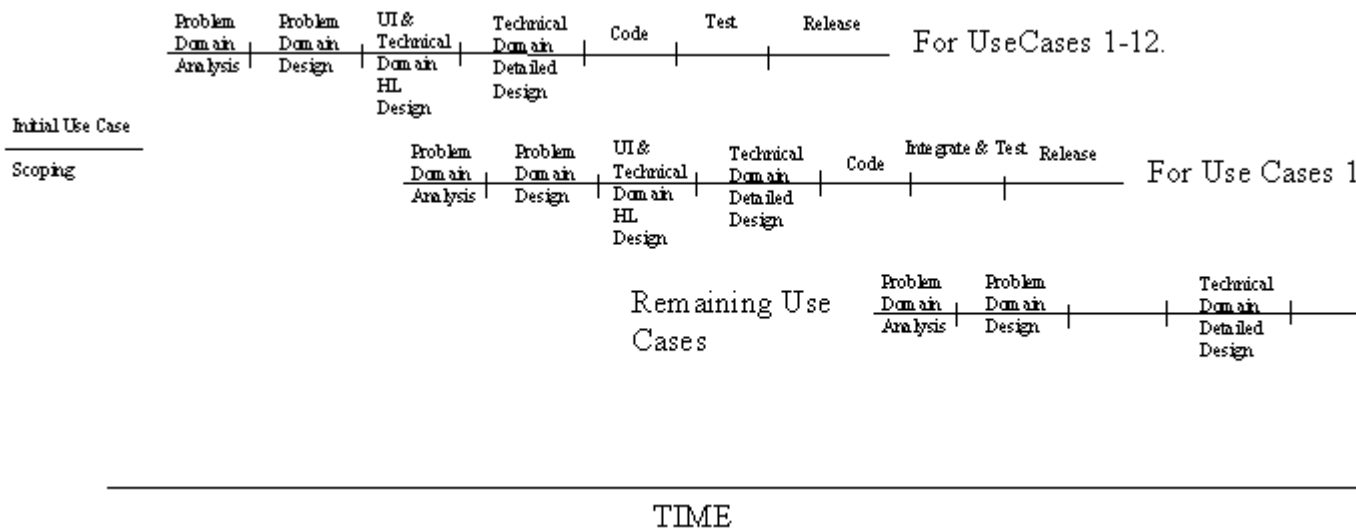


Figure 14 - Evolutionary Phasing Of OO Project

The object-based structure of the application lends itself well to this approach.

Summary

This paper has taken a look at the Use Case, Object Modelling, and Sequence Diagramming notations of UML, how Object Modelling maps to OO programming languages, and shown how these notations hang together to complement each other. A number of other UML notations are not covered in this article, however further information can be found on www.ratio.co.uk.

I hope you can see that OOA/D offers a number of potential benefits in an OO based development environment. These include:

- better modelling of the problem domain (equals happier users)
- better overall software design with a strong focus on class structure
- more flexible and maintainable systems through better class partitioning
- good documentation (the notations) - and a single central overall design notation (the Object Model)
- a flexible approach to project phasing

- assistance in tie-ing down requirements, and
- less work (in the long run)

Mark Collins-Cope is Technical Director at Ratio Group Ltd., a consultancy and training company specialising in OO related methods, languages and technologies. For further information on OOA/D using UML, Java, C++, Design Patterns or related topics please [contact us](#)

Copyright

This material remains the copyright of Ratio Group Ltd. Licence is granted for the use of this material for personal development purposes only.