

TRƯỜNG ĐẠI HỌC ĐÀ LẠT  
KHOA CÔNG NGHỆ THÔNG TIN



GIÁO TRÌNH  
CÔNG NGHỆ PHẦN MỀM

(Lưu hành nội bộ)

Tháng 8 năm 2016

## LỜI MỞ ĐẦU

## MỤC LỤC

<b>CHƯƠNG 1. TỔNG QUAN VỀ CÔNG NGHỆ PHẦN MỀM .....</b>	<b>9</b>
1.1    Giới thiệu .....	9
1.2    Phần mềm và công nghệ phần mềm .....	9
1.2.1    Phần mềm .....	9
1.2.2    Cấu trúc phần mềm .....	10
1.2.3    Chất lượng phần mềm .....	10
1.2.4    Công nghệ phần mềm .....	11
1.3    Các quy trình công nghệ phần mềm .....	12
1.3.1    Giai đoạn xác định yêu cầu .....	12
1.3.2    Giai đoạn phân tích hệ thống .....	13
1.3.3    Giai đoạn thiết kế .....	13
1.3.4    Giai đoạn xây dựng (mã hóa) .....	13
1.3.5    Giai đoạn kiểm thử .....	13
1.3.6    Giai đoạn triển khai .....	13
1.3.7    Giai đoạn bảo trì .....	14
1.4    Các phương pháp phát triển .....	14
1.4.1    Mô hình thác nước .....	14
1.4.2    Mô hình bản mẫu .....	15
1.4.3    Mô hình xoắn ốc .....	15
1.5    Các công cụ xây dựng phần mềm .....	16
1.5.1    Phần mềm hỗ trợ phân tích .....	16
1.5.2    Phần mềm hỗ trợ thiết kế .....	17
1.5.3    Phần mềm hỗ trợ lập trình .....	17
1.5.4    Phần mềm hỗ trợ kiểm thử .....	17
1.5.5    Phần mềm hỗ trợ tổ chức, quản lý việc triển khai .....	17
1.6    Kết chương .....	17

<b>CHƯƠNG 2. CÁC SƠ ĐỒ PHÂN TÍCH HƯỚNG ĐỐI TƯỢNG .....</b>	<b>18</b>
2.1    Tổng quan về UML .....	18
2.1.1    Lịch sử ra đời .....	18
2.1.2    Ngôn ngữ mô hình hoá hướng đối tượng UML .....	18
2.2    Giới thiệu các sơ đồ trong UML .....	20
2.2.1    Sơ đồ Use Case.....	20
2.2.2    Sơ đồ lớp - Mức độ phân tích.....	24
2.2.3    Sơ đồ cộng tác .....	26
2.2.4    Sơ đồ triển khai .....	27
2.2.5    Sơ đồ lớp - Mức độ thiết kế .....	29
2.2.6    Sơ đồ tuần tự .....	30
2.3    Kết chương .....	31
<b>CHƯƠNG 3. XÁC ĐỊNH YÊU CẦU PHẦN MỀM .....</b>	<b>33</b>
3.1    Giới thiệu .....	33
3.2    Sự ra đời của một sản phẩm .....	35
3.3    Use case.....	36
3.4    Khía cạnh nghiệp vụ .....	37
3.4.1    Xác định business .....	38
3.4.2    Viết dự án chuyên môn .....	39
3.4.3    Xác định business use case .....	41
3.4.4    Minh họa use case trong lược đồ giao tiếp.....	43
3.4.5    Minh họa use cases trong biểu đồ hoạt động .....	45
3.5    Khía cạnh người phát triển.....	46
3.5.1    Nhận biết system actors .....	48
3.5.2    Nhận biết hệ thống Use Cases .....	48
3.5.3    Chuyên môn hóa Actors .....	50
3.5.4    Quan hệ giữa các use case .....	51
3.6    Hệ thống chi tiết use case.....	55
3.6.1    Preconditions, postconditions and Inheritance.....	57
3.6.2    Những yêu cầu bổ sung .....	59

3.6.3	Phác thảo interface cho người dùng .....	59
3.6.4	Ưu tiên use case hệ thống.....	60
3.7	Kết chương.....	62
<b>CHƯƠNG 4. PHÂN TÍCH YÊU CẦU PHẦN MỀM.....</b>		<b>64</b>
4.1	Giới thiệu .....	64
4.2	Quá trình phân tích yêu cầu phần mềm .....	65
4.3	Xây dựng sơ đồ lớp .....	66
4.3.1	Tìm các lớp ứng viên.....	66
4.3.2	Nhận biết mối quan hệ giữa các lớp.....	67
4.3.3	Vẽ sơ đồ lớp .....	68
4.3.4	Bổ sung thuộc tính .....	72
4.3.5	Bổ sung các lớp kết hợp .....	75
4.4	Xây dựng sơ đồ cộng tác.....	76
4.4.1	Các đối tượng trong sơ đồ giao tiếp .....	77
4.4.2	Các thành phần của sơ đồ giao tiếp.....	78
4.4.3	Các bước tạo sơ đồ giao tiếp .....	81
4.5	Xây dựng sơ đồ tuần tự .....	82
4.5.1	Các thành phần của một sơ đồ tuần tự .....	83
4.5.2	Các nguyên tắc khi vẽ sơ đồ tuần tự .....	84
4.5.3	Các bước để xây dựng sơ đồ tuần tự .....	85
4.6	Xây dựng sơ đồ hoạt động .....	86
4.6.1	Các thành phần cơ bản của một sơ đồ hoạt động .....	88
4.6.2	Một số lưu ý khi vẽ sơ đồ hoạt động.....	91
4.6.3	Các bước để xây dựng sơ đồ hoạt động .....	92
4.7	Kết chương .....	92
<b>CHƯƠNG 5. THIẾT KẾ PHẦN MỀM.....</b>		<b>94</b>
5.1	Giới thiệu .....	94
5.1.1	Khái niệm .....	94
5.1.2	Mục tiêu.....	94
5.1.3	Quá trình thiết kế .....	95

5.2	Thiết kế dữ liệu .....	95
5.2.1	Mục tiêu.....	95
5.2.2	Kết quả của thiết kế dữ liệu.....	96
5.2.3	Quá trình thiết kế.....	98
5.3	Thiết kế xử lý .....	105
5.3.1	Các danh sách thiết kế xử lý.....	105
5.3.2	Mô tả hàm xử lý .....	107
5.4	Thiết kế giao diện.....	110
5.4.1	Người dùng.....	111
5.4.2	Các nguyên tắc thiết kế giao diện .....	112
5.4.3	Biểu diễn bố cục giao diện .....	116
5.4.4	Biểu diễn thông tin .....	125
5.4.5	Quy trình thiết kế giao diện.....	126
5.4.6	Khảo sát người dùng và phân tích, đánh giá giao diện .....	127
5.5	Xây dựng sơ đồ lớp .....	128
5.6	Xây dựng sơ đồ trạng thái .....	129
5.7	Kết chương .....	129
<b>CHƯƠNG 6. XÂY DỰNG PHẦN MỀM .....</b>	<b>130</b>	
6.1	Kiến trúc phần mềm .....	130
6.1.1	Các kiểu kiến trúc phần mềm.....	130
6.1.2	Sự kết hợp các kiểu kiến trúc .....	132
6.1.3	Phân tầng ứng dụng.....	132
6.1.4	Sự chọn lựa công nghệ sử dụng .....	135
6.2	Một số công cụ hỗ trợ xây dựng phần mềm .....	135
6.2.1	Quản lý mã nguồn với công cụ Git .....	136
6.2.2	Những mô hình làm việc với Git .....	139
6.2.3	Quản lý tiến độ công việc với Kanban .....	145
6.3	Một số nguyên tắc cơ bản trong xây dựng phần mềm .....	147
6.3.1	Chuẩn hóa mã hóa .....	147
6.3.2	Sử dụng mẫu thiết kế.....	152

6.4	Ví dụ minh họa.....	157
6.4.1	Mô hình kiến trúc chung .....	157
6.4.2	Kết nối và thao tác dữ liệu .....	160
6.4.3	Thực thi một số Use case .....	167
6.5	Kết chương.....	171
<b>CHƯƠNG 7. KIỂM THỬ PHẦN MỀM .....</b>	<b>173</b>	
7.1	Giới thiệu .....	173
7.1.1	Thuật ngữ “Kiểm thử” .....	173
7.1.2	Black-Box Testing (kiểm thử hộp đen).....	174
7.1.3	White-Box Testing (kiểm thử hộp trắng).....	175
7.2	Các mức kiểm thử phần mềm .....	176
7.2.1	Unit testing (kiểm thử hàm) .....	176
7.2.2	Integration Testing (Kiểm thử tích hợp) .....	176
7.2.3	Use Case Testing (Kiểm thử ca sử dụng).....	177
7.3	Một số nguyên tắc trong kiểm thử .....	177
7.3.1	Kiểm thử trong quá trình phát triển.....	178
7.3.2	Kiểm thử trong quá trình kiểm thử.....	179
7.3.3	Kiểm thử sau khi phát hành .....	179
7.4	TEST-DRIVEN DEVELOPMENT .....	180
7.4.1	Tổng quan về TDD.....	180
7.4.2	Ví dụ về TDD sử dụng JUnit .....	182
7.4.3	Tổ chức lại mã nguồn các bộ Kiểm thử .....	186
7.4.4	Tạo một bộ kiểm thử cho quá trình .....	189
7.4.5	Kiểm thử chéo các phương thức .....	191
7.4.6	Completing theStoreClass .....	192
7.5	Kết chương.....	194
<b>CHƯƠNG 8. MỘT SỐ KỸ THUẬT TRONG TRIỂN KHAI VÀ BẢO TRÌ HỆ THỐNG .....</b>	<b>195</b>	
8.1	Giới thiệu .....	195
8.2	Cấu hình phần mềm .....	195

8.2.1	Bối cảnh và các thành phần liên quan.....	195
8.2.2	Quá trình cấu hình phần mềm .....	196
8.2.3	Cấu hình cho các ứng dụng Web .....	199
8.3	Triển khai phần mềm trên các hệ thống.....	204
8.4	Một số nguyên lý trong bảo trì phần mềm.....	204
8.4.1	Dự đoán sự bảo trì .....	207
8.4.2	Tái cấu trúc phần mềm .....	208
8.4.3	Bảo dưỡng chương trình bằng tái cấu trúc .....	209
8.5	Kết chương.....	210

# CHƯƠNG 1. TỔNG QUAN VỀ CÔNG NGHỆ PHẦN MỀM

## 1.1 Giới thiệu

Ngày nay, khi công nghệ thông tin đã có nhiều bước phát triển thì việc ứng dụng các kỹ thuật của công nghệ thông tin vào trong các hoạt động nghiệp vụ đã trở nên phổ biến đối với nhiều người. Muốn thực hiện được điều đó, người ta phải tạo ra các hệ thống phần mềm nhằm hỗ trợ tối đa các nghiệp vụ cần thiết.

Ngành công nghệ phần mềm đã phát triển từ lâu và mang lại nhiều thành quả thiết thực. Vậy công nghệ phần mềm là gì, các bước tiến hành để xây dựng một phần mềm như thế nào, trong chương này ta sẽ tìm hiểu một số khái niệm liên quan đến những vấn đề đó.

## 1.2 Phần mềm và công nghệ phần mềm

### 1.2.1 Phần mềm

Chương trình máy tính là một tập các lệnh (chỉ thị) để giúp máy tính hoàn thành một công việc nào đó của con người. Chương trình máy tính thường được tạo ra bởi các nhà lập trình viên, một chương trình máy tính được phát triển giúp các nhà chuyên môn thực hiện các thao tác nghiệp vụ, giúp cho các công việc diễn ra nhanh hơn, thuận tiện hơn.

Theo Phạm Thị Quỳnh [1] phần mềm là các chương trình máy tính và những tài liệu liên quan đến như: các yêu cầu, mô hình thiết kế, tài liệu hướng dẫn sử dụng.

Các sản phẩm phần mềm thường được chia thành:

- Phần mềm đại chúng (Generic Product): được các lập trình viên phát triển để tung ra thị trường ở dạng miễn phí hoặc có bản quyền. Sản phẩm thuộc loại này thường dành cho các dòng máy tính cá nhân như: PC, Laptop...
- Phần mềm theo đơn hàng (Bespoke Product): được đặt hàng để phát triển cho một khách hàng riêng lẻ theo một yêu cầu chuyên dụng. Ví dụ: các phần mềm máy ATM, các phần mềm hệ điều hành.

Nhiệm vụ chủ yếu của phần mềm là giúp cho các nhà chuyên môn thực hiện công việc một cách hiệu quả hơn, nhanh hơn và chính xác hơn.

## 1.2.2 Cấu trúc phần mềm

Phần mềm ngày nay khi được lập trình viên tạo ra thường có các thành phần đi kèm. Các thành phần này thường được các bộ phận xây dựng riêng rẽ nhưng có các liên kết để giúp liên kết lại với nhau.

### 1.2.2.1 Thành phần giao tiếp người dùng

Thành phần này tiếp nhận các yêu cầu, công việc tương tự thực tế và các số liệu liên quan. Thành phần giao diện thường là một tập hợp các giao diện cho phép người dùng nhập số liệu và hiển thị kết quả.

### 1.2.2.2 Thành phần dữ liệu

Thành phần dữ liệu được tổ chức thành các tập dữ liệu có cấu trúc, cho phép lưu trữ các kết quả đã xử lý trên bộ nhớ phụ. Thành phần dữ liệu còn cho phép truy xuất lại hoặc tìm kiếm các dữ liệu đã có để phục vụ cho các công việc tương ứng.

### 1.2.2.3 Thành phần xử lý

Thành phần xử lý là thành phần ẩn đối với người dùng. Thành phần này là một tập các hàm, các phương thức nhằm kiểm tra tính hợp lệ của dữ liệu được cung cấp từ người dùng theo ràng buộc mà thành phần giao diện cho phép hoặc trong thế giới thực. Thành phần xử lý cũng giúp xử lý các kết quả theo mong đợi của người dùng.

## 1.2.3 Chất lượng phần mềm

Chất lượng phần mềm nhằm giúp xác định phần mềm được làm ra đáp ứng được nhu cầu của người dùng dựa trên các nghiệp vụ đã có. Sau đây là một số tính nhất hỗ trợ cho chất lượng của phần mềm.

### 1.2.3.1 Tính đúng đắn

Tính đúng đắn được thể hiện bằng cách sản phẩm đó thực hiện đầy đủ và chính xác các yêu cầu nghiệp vụ. Cần phải hiểu rộng hơn là chương trình cần phải thực hiện được trong cả những trường hợp mà dữ liệu đầu vào là không hợp lệ hoặc chỉ có một số trường hợp hợp lệ.

### 1.2.3.2 Tính tiến hóa

Dựa trên các quy định trong thực tế, những quy định này sẽ thay đổi theo thời gian hoặc khách quan, các phần mềm phải khai báo các giá trị để đáp ứng các quy định này. Các giá trị này thường được gọi là tham số của phần mềm. Ví dụ: Số sách được mượn tối đa trong thư viện, thuế của các mặt hàng...

#### **1.2.3.3 Tính hiệu quả**

Tính hiệu quả của một sản phẩm phần mềm được xác định thông qua các tiêu chuẩn về hiệu quả mà phần mềm đó mang lại như: hiệu quả kinh tế, hiệu quả công việc, tốc độ xử lý, tối ưu hóa tài nguyên...

#### **1.2.3.4 Tính tiện dụng**

Sản phẩm phần mềm phải thân thiện với người sử dụng, có giao diện trực quan, dễ thao tác, dễ thực hiện với những người không có chuyên ngành công nghệ thông tin.

#### **1.2.3.5 Tính tương thích**

Sản phẩm phần mềm có thể dễ dàng trao đổi dữ liệu với các phần mềm quản lý dữ liệu khác như: Word, Excel, Foxpro...

#### **1.2.3.6 Tính tái sử dụng**

Sản phẩm phần mềm có thể áp dụng trong nhiều lĩnh vực theo các chế độ làm việc khác nhau, dễ dàng tích hợp với các nhánh phần mềm khác khi được xây dựng thêm.

### **1.2.4 Công nghệ phần mềm**

#### **1.2.4.1 Lịch sử phát triển**

Lịch sử phát triển của công nghệ phần mềm gắn liền với lịch sử phát triển của máy tính. Lịch sử phát triển được chia thành các giai đoạn:

Giai đoạn 1: Thời kỳ thế hệ thứ nhất của máy tính điện tử vào thập niên 50. Giai đoạn này chủ yếu dựa trên quan điểm lập trình là một hoạt động nghệ thuật, ngôn ngữ lập trình là ngôn ngữ máy, bậc thấp, phương pháp lập trình tuyến tính nên năng suất thấp và mất nhiều thời gian.

Giai đoạn 2: Vào thập niên 60, thời kỳ này được coi là khủng hoảng phần mềm vì chi phí sản xuất phần mềm cao và các dự án không có tính khả thi do phương pháp xây dựng không bắt kịp với nhu cầu và phần cứng. Kết quả là sự ra đời của các hệ tiên đề chứng minh tính đúng đắn của chương trình bằng lý thuyết, và các phương pháp xây dựng cấu trúc phần mềm ra đời.

Giai đoạn 3: Từ giữa các thập niên 70 đến thập niên 90, hệ thống phần mềm phát triển mạnh, tập trung xử lý các phần mềm thông qua mạng (LAN, Internet...) và do sự phát triển nhanh của các máy tính cá nhân (PC) nên các phần mềm đáp ứng nhu cầu cá nhân cũng được xây dựng ngày càng nhiều và hiệu quả.

Giai đoạn thứ 4: Công nghệ phần mềm hướng đối tượng đã thay đổi tư duy và phương pháp tiếp cận phần mềm. Sự đa dạng hóa các sản phẩm, ứng dụng cũng cho phép nhiều loại chương trình phát triển nhằm đáp ứng nhu cầu của người dùng.

#### **1.2.4.2 Công nghệ phần mềm**

Công nghệ phần mềm là một lĩnh vực nghiên cứu của tin học nhằm đề xuất các nguyên lý, phương pháp, công cụ, cách tiếp cận phục vụ cho việc thiết kế, cài đặt các sản phẩm phần mềm đạt được đầy đủ các yêu cầu về chất lượng phần mềm.

Công nghệ phần mềm là một tập các giai đoạn, thường được chia thành 7 giai đoạn chính là: Khảo sát, phân tích, thiết kế, xây dựng, kiểm thử, triển khai và bảo trì. Mỗi một giai đoạn lại có các quy tắc riêng, có các tài liệu đi kèm để hoàn thiện các thao tác của giai đoạn đó.

Theo tác giả Nguyễn Thị Quỳnh [1] thì khoa học máy tính đề cập tới lý thuyết và những vấn đề cơ bản, còn công nghệ phần mềm đề cập tới các hoạt động xây dựng và đưa ra một phần mềm hữu ích.

Do vậy, các giai đoạn trong công nghệ phần mềm nếu phối hợp tốt sẽ cho ra một sản phẩm hoàn thiện, nếu chỉ một giai đoạn thực hiện chưa đúng yêu cầu sẽ có một phần mềm có chất lượng thấp và không đáp ứng được nhu cầu của người dùng.

### **1.3 Các quy trình công nghệ phần mềm**

Để xây dựng được hoàn thiện phần mềm có chất lượng, người xây dựng phần mềm phải trải qua nhiều giai đoạn, mỗi giai đoạn sẽ có mục tiêu và kết quả để chuyển giao cho các giai đoạn tiếp theo. Các giai đoạn có thể diễn ra theo hình thức nối tiếp hoặc song song hoặc gối đầu.

#### **1.3.1 Giai đoạn xác định yêu cầu**

Đây là giai đoạn đầu tiên hình thành bài toán hoặc đề tài, ở bước này bộ phận thiết kế phải khảo sát hệ thống, đặt ra những câu hỏi với người sử dụng hệ thống để biết các nghiệp vụ cơ bản trong thế giới thực. Trong bước này, thường người ta cũng xác định luôn các vai trò của phần mềm, ước lượng công việc và lập lịch biểu phân công công việc cho nhân viên.

Sau khi hoàn thành bước này, người phân tích sẽ có được tài liệu khảo sát yêu cầu, các yêu cầu cơ bản của phần mềm, các biểu mẫu và một số quy định liên quan.

### 1.3.2 Giai đoạn phân tích hệ thống

Dựa vào các yêu cầu đã xác định từ bước trước, nhóm phát triển phần mềm dùng các loại ngôn ngữ đặc tả hình thức hoặc phi hình thức để đặc tả các yêu cầu đó. Với phương pháp phân tích cấu trúc, người ta thường dùng mô hình DFD (Data Flow Diagram) để minh họa các yêu cầu; với phương pháp hướng đối tượng người ta sẽ dùng các mô hình UML để mô hình hóa các yêu cầu.

### 1.3.3 Giai đoạn thiết kế

Đây là giai đoạn các nhóm làm việc phải vẽ ra được mô hình hóa hệ thống dựa trên hai mặt: dữ liệu và chương trình.

Mô hình hóa dữ liệu cho phép thiết kế dữ liệu (bảng, tập tin...) để lưu trữ khi phần mềm hoàn thành. Mô hình hóa dữ liệu phải đảm bảo được các yêu cầu phần mềm và phải có các ràng buộc dữ liệu cần thiết.

Mô hình hóa chương trình cho phép thiết kế trước các giao diện, các biểu mẫu nhập để người xây dựng dựa vào đó xây dựng các biểu mẫu. Trong bước mô hình hóa chương trình ngày nay người ta thường hay dùng phương pháp xây dựng sẵn các biểu mẫu trống, để người xây dựng từ đó xây dựng theo. Ví dụ: nếu làm web, người ta sẽ thiết kế dạng HTML sẵn, nếu làm lập trình Windows thì các giao diện được thiết kế ở bước này.

### 1.3.4 Giai đoạn xây dựng (mã hóa)

Giai đoạn này các lập trình viên sẽ tiến hành dùng các ngôn ngữ thích hợp để cài đặt các thao tác cần thiết để chương trình thực hiện đúng các yêu cầu đã được đặc tả.

Giai đoạn này người ta thường áp dụng các phương pháp lập trình hiện nay như: lập trình với mô hình đa tầng (n-tier), lập trình tự sinh mã một phần... để giúp tạo ra ứng dụng nhanh và hiệu quả hơn.

### 1.3.5 Giai đoạn kiểm thử

Công việc kiểm thử nhằm chứng minh tính đúng đắn của chương trình sau khi đã tiến hành cài đặt. Để kiểm thử phải có các bộ dữ liệu nhập, khi đó áp dụng các phương pháp để tìm ra các lỗi cần thiết và sửa các lỗi đó. Công việc kiểm thử cũng giúp kiểm tra tính ổn định của chương trình, tính hiệu quả cũng như khả năng hỗ trợ tối đa của chương trình.

### 1.3.6 Giai đoạn triển khai

Giai đoạn triển khai sẽ triển khai phần mềm ra thực tế. Giai đoạn triển khai thường bao gồm giai đoạn cài đặt chương trình và đào tạo hướng dẫn sử dụng. Giai đoạn này, cũng có

một quá trình được gọi là kiểm thử Beta, là dạng kiểm thử do người sử dụng dùng thử nhằm mục đích tìm lỗi cần thiết và sửa chúng.

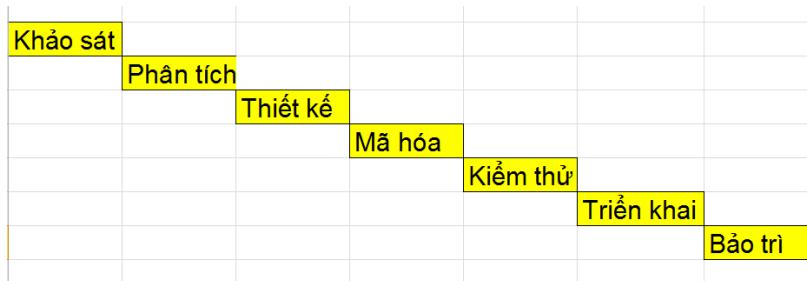
### 1.3.7 Giai đoạn bảo trì

Công việc bảo trì phần mềm được tiến hành bởi một giai đoạn sau quá trình triển khai nhằm khắc phục các lỗi cần thiết để đáp ứng các yêu cầu người dùng. Có những chức năng sau khi triển khai mới phát sinh lỗi và nhiệm vụ của giai đoạn bảo trì là khắc phục các lỗi này.

## 1.4 Các phương pháp phát triển

### 1.4.1 Mô hình thác nước

Mô hình thác nước là mô hình đầu tiên và phổ biến được áp dụng trong quá trình phát triển phần mềm. Mô hình này dựa trên sự hoạt động của một thác nước trong tự nhiên, theo đó thì các lớp nước sẽ nối tiếp nhau chảy xuống dưới mà không bao giờ chảy ngược lên lại. Quá trình phát triển phần mềm cũng có nhiều giai đoạn, nếu ta coi mỗi giai đoạn như là một lớp nước thì các giai đoạn này phải diễn ra một cách tuần tự nối tiếp nhau. Kết quả của giai đoạn trước sẽ là thông tin đầu vào cho giai đoạn tiếp theo sau.



Hình 1.1 Mô hình thác nước với 7 giai đoạn trong công nghệ phần mềm

Hình 1.1 biểu diễn một mô hình thác nước điển hình, theo mô hình này thì 7 giai đoạn sẽ được nối tiếp nhau thực hiện, giai đoạn trước thực hiện xong thì mới tới giai đoạn tiếp theo. Các giai đoạn diễn ra cho tới khi toàn bộ sản phẩm hoàn thành.

Mô hình thác nước giúp chúng ta có thể dễ dàng phân chia quá trình xây dựng phần mềm thành những giai đoạn độc lập. Tuy nhiên, các dự án lớn hiếm khi tuân theo dòng chảy tuần tự của mô hình vì thường phải lặp lại hoặc quay lại một số bước để nâng cao chất lượng. Hơn nữa, khách hàng cũng khó có thể lường trước được hết các yêu cầu khi phần mềm chưa hoàn thiện. Do đó, phương pháp này chỉ thích hợp cho những trường hợp là bài toán nhỏ hoặc người phân tích hiểu rất rõ các yêu cầu của khách hàng. Mô hình thác

nước có thể được cải tiến bằng cách cho phép quay lui khi phát hiện lỗi trong giai đoạn phía trước.

### 1.4.2 Mô hình bản mẫu

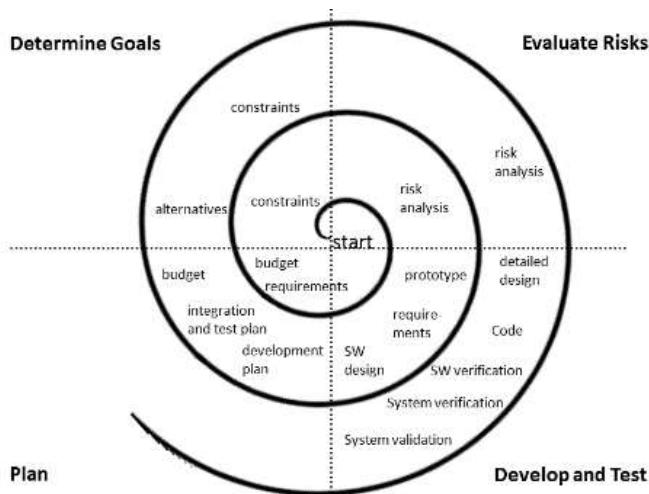
Mô hình bản mẫu tương tự như mô hình thác nước nhưng có bổ sung vào các giai đoạn thực hiện một bản mẫu ngay khi xác định yêu cầu nhằm phát hiện nhanh các sai sót về yêu cầu. Ngay sau khi giai đoạn xác định yêu cầu, nhà phát triển phần mềm đưa ra ngay một bản thiết kế sơ bộ và tiến hành cài đặt bản mẫu đầu tiên và chuyển cho người sử dụng. Bản mẫu này chỉ nhằm để miêu tả cách thức phần mềm hoạt động cũng như cách người sử dụng tương tác với hệ thống.

Người sử dụng sau khi xem xét sẽ phản hồi thông tin cần thiết lại cho nhà phát triển. Nếu người sử dụng đồng ý với bản mẫu đã đưa thì người phát triển sẽ tiến hành cài đặt thực sự theo bản mẫu, ngược lại phải quay lại giai đoạn xác định yêu cầu. Công việc này được lặp lại liên tục cho đến khi người sử dụng đồng ý với các bản mẫu do nhà phát triển đưa ra.

Như vậy đây là một hướng tiếp cận tốt khi các yêu cầu chưa rõ ràng và khó đánh giá được tính hiệu quả của các thuật toán. Tuy nhiên, mô hình này cũng có nhược điểm là tính cấu trúc không cao do đó khách hàng dễ mất tin tưởng, mô hình này cũng mất khá nhiều thời gian để hoàn thành bản mẫu và bản chính thức.

### 1.4.3 Mô hình xoắn ốc

Mô hình xoắn ốc dựa trên hoạt động của vỏ ốc, có trúc xoắn từ ngoài vào trong, mỗi một vòng xoắn chính là một cấu trúc hoàn chỉnh. Mô hình này chính là sự kết hợp của mô hình bản mẫu thiết kế và mô hình thác nước được lặp đi lặp lại nhiều lần.



Hình 1.2 Mô hình xoắn ốc

Ở mỗi lần lặp các yêu cầu của người sử dụng sẽ được hiểu ngày càng rõ ràng hơn và các bản mẫu phần mềm cũng ngày một hoàn thiện hơn. Ngoài ra ở cuối mỗi lần lặp sẽ có thêm công đoạn phân tích mức độ rủi ro để quyết định xem có nên đi tiếp theo hướng này nữa hay không.

Mô hình này phù hợp với các hệ thống phần mềm lớn do có khả năng kiểm soát rủi ro ở từng bước tiến hóa. Tuy nhiên vẫn chưa được sử dụng rộng rãi như mô hình thác nước hoặc bản mẫu do đòi hỏi năng lực quản lý, năng lực phân tích rủi ro cao.

## 1.5 Các công cụ xây dựng phần mềm

Các công cụ và môi trường phát triển phần mềm chính là các phần mềm hỗ trợ nhà phát triển trong quá trình xây dựng phần mềm. Các phần mềm này có tên gọi chung là CASE (Computer Aided Software Engineering), tạm dịch là Thiết kế phần mềm có sự hỗ trợ của máy tính.

Trong quá trình phát triển phần mềm theo các quy trình trên, các CASE có thể hỗ trợ cụ thể cho một giai đoạn nào đó hay cũng có thể hỗ trợ một số giai đoạn, trong trường hợp này tên gọi chung thường là môi trường phát triển phần mềm (SDE - Software Development Environment).

### 1.5.1 Phần mềm hỗ trợ phân tích

Phần mềm hỗ trợ phân tích hỗ trợ cho người phân tích phần mềm thực hiện công việc một cách tốt hơn. Các thao tác được hỗ trợ như: soạn thảo các mô hình thế giới thực, ánh xạ vào mô hình logic, phân tích các chức năng... Các phần mềm thường được sử dụng như: WinA&D, Analyst Pro, Microsoft Office...

### 1.5.2 Phần mềm hỗ trợ thiết kế

Nhằm giúp cho các công việc thiết kế phần mềm mang tính trực quan, hiệu quả. Các công việc chính được hỗ trợ như: mô hình hóa yêu cầu, soạn thảo các mô hình logic, ánh xạ vào mô hình vật lý. Các phần mềm thông dụng như: QuickUML, Power Designer, Oracle Designer, Microsoft Visio...

### 1.5.3 Phần mềm hỗ trợ lập trình

Phần mềm hỗ trợ lập trình hiện nay rất đa dạng, hỗ trợ nhiều ngôn ngữ trong việc xây dựng phần mềm. Các bộ phần mềm thông dụng như: Visual Studio, Java Sun, Web Application, Mobile Application, Game Application...

### 1.5.4 Phần mềm hỗ trợ kiểm thử

Phần mềm hỗ trợ kiểm thử giúp cho công tác tìm lỗi một cách hiệu quả. Các công việc được hỗ trợ chính như: phát sinh tự động các bộ dữ liệu thử nghiệm, phát hiện lỗi, cảnh báo lỗi tiềm ẩn... Các phần mềm như: WinRuner, XMind...

### 1.5.5 Phần mềm hỗ trợ tổ chức, quản lý việc triển khai

Ngoài những phần mềm hỗ trợ các giai đoạn trong việc xây dựng phần mềm, người ta còn phải dùng đến các phần mềm hỗ trợ trong việc tổ chức, quản lý triển khai dự án, lập lịch biểu, quản lý rủi ro. Có thể kể đến các phần mềm như: MS Project, Visio...

Ngoài ra, để tổ chức và quản lý dự án hiệu quả, người ta cũng sử dụng tới các hệ thống quản lý như: Redmind, Trellor... hoặc các phần mềm quản lý tổ chức code như: TortoiseSVN và Visual SVN.

## 1.6 Kết chương

Chương này nêu một số kiến thức cơ bản về công nghệ phần mềm và quá trình phát triển, cũng như các công cụ để xây dựng phần mềm. Muốn tìm hiểu sâu và chi tiết hơn các quy trình, cũng như các bước thực hiện, độc giả có thể tham khảo bằng một phương pháp thiết kế cụ thể.

### Tài liệu tham khảo

1. Phạm Thị Quỳnh, “Nhập môn Công nghệ phần mềm”, Đại học Bách khoa Hà Nội

## CHƯƠNG 2. CÁC SƠ ĐỒ PHÂN TÍCH HƯỚNG ĐỐI TƯỢNG

### 2.1 Tổng quan về UML

#### 2.1.1 Lịch sử ra đời

Việc áp dụng rộng rãi phương pháp hướng đối tượng đã đặt ra yêu cầu cần phải xây dựng một phương pháp mô hình hóa để có thể sử dụng như một chuẩn chung cho những người phát triển phần mềm hướng đối tượng trên khắp thế giới. Trong khi các ngôn ngữ hướng đối tượng ra đời khá sớm, ví dụ như Simula-67 (năm 1967), Smalltalk (đầu những năm 1980), C++, CLOS (giữa những năm 1980)... thì những phương pháp luận cho phát triển hướng đối tượng lại ra đời khá muộn. Cuối những năm 80, đầu những năm 1990, một loạt các phương pháp luận và ngôn ngữ mô hình hóa hướng đối tượng mới ra đời, như Booch của Grady Booch, OMT của James Rumbaugh, OOSE của Ivar Jacobson, hay OOA and OOD của Coad và Yordon.

Mỗi phương pháp luận và ngôn ngữ trên đều có hệ thống ký hiệu riêng, phương pháp xử lý và công cụ hỗ trợ riêng. Chính điều này đã thúc đẩy những người tiên phong trong lĩnh vực mô hình hóa hướng đối tượng ngồi lại cùng nhau để tích hợp những điểm mạnh của mỗi phương pháp và đưa ra một mô hình thống nhất chung. Nỗ lực thống nhất đầu tiên bắt đầu khi Rumbaugh gia nhập nhóm nghiên cứu của Booch tại tập đoàn Rational năm 1994 và sau đó Jacobson cũng gia nhập nhóm này vào năm 1995. James Rumbaugh, Grady Booch và Ivar Jacobson đã cùng cố gắng xây dựng được Ngôn ngữ mô hình hóa thống nhất và đặt tên là UML (Unifield Modeling Language). UML đầu tiên được đưa ra năm 1997 và sau đó được chuẩn hóa để trở thành phiên bản 1.0. Hiện nay, phiên bản mới nhất của ngôn ngữ UML là 2.5.

#### 2.1.2 Ngôn ngữ mô hình hóa hướng đối tượng UML

UML là ngôn ngữ mô hình hóa thống nhất được xây dựng để đặc tả, phát triển và viết tài liệu cho các khía cạnh trong phát triển phần mềm hướng đối tượng. UML giúp người phát triển hiểu rõ và ra quyết định liên quan đến phần mềm cần xây dựng. UML bao gồm một tập các khái niệm, các ký hiệu, các sơ đồ và hướng dẫn.

UML hỗ trợ xây dựng hệ thống hướng đối tượng dựa trên việc nắm bắt khía cạnh cấu trúc tĩnh và các hành vi động của hệ thống.

- Các cấu trúc tĩnh định nghĩa các kiểu đối tượng quan trọng của hệ thống, nhằm cài đặt và chỉ ra mối quan hệ giữa các đối tượng đó.

- Các hành vi động định nghĩa các hoạt động của các đối tượng theo thời gian và tương tác giữa các đối tượng hướng tới đích.

Các mục đích chính của việc sử dụng UML:

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

UML quy định một loạt các ký hiệu và quy tắc để mô hình hóa các pha trong quá trình phát triển phần mềm hướng đối tượng dưới dạng các sơ đồ. Bảng 2.1 mô tả mối quan hệ giữa các đối tượng và các giai đoạn hệ thống có sử dụng UML.

Giai đoạn	Chi tiết		UML
Hình thành	Nhiệm vụ hoặc các yêu cầu không chính thức		Không
	Vai trò		Không
	Trách nhiệm		Không
	Kế hoạch dự án		Không
	Danh mục công việc (Workbook)		Không
	Bảng chú giải (cập nhật liên tục)		Không
	Kế hoạch kiểm tra		Không
Các yêu cầu	Nghiệp vụ	Danh sách tác nhân (với mô tả)	Không
		Danh sách use case (với mô tả)	Không
		Chi tiết use case	Không
		Sơ đồ hoạt động (tùy chọn)	Có
		Sơ đồ cộng tác (tùy chọn)	Có
	Hệ thống	Danh sách tác nhân (với mô tả)	Không
		Danh sách use case (với mô tả)	Không
		Chi tiết use case	Không
		Sơ đồ use case	Có
		Tổng quát use case	Không

		Phác thảo giao diện người dùng	Không
Phân tích		Sơ đồ lớp	Có
		Sơ đồ cộng tác	Có
Thiết kế	Hệ thống	Sơ đồ triển khai	Có
		Sơ đồ tầng	Không
	Hệ thống con	Sơ đồ lớp Sơ đồ tuần tự Lược đồ cơ sở dữ liệu	Có Có Không
Lớp riêng biệt	Ghi chú	Không	
Thực thi	Source code	Không	
Kiểm tra	Báo cáo kiểm tra	Không	
Triển khai	Giải pháp bao phủ tối thiểu	Không	
	Tài liệu hướng dẫn	Không	
	Tài nguyên huấn luyện	Không	
Bảo trì	Báo cáo lỗi	Không	
	Kế hoạch phát triển	Không	

Bảng 2.1 Các giai đoạn hệ thống và UML

## 2.2 Giới thiệu các sơ đồ trong UML

### 2.2.1 Sơ đồ Use Case

Use Case là một mô tả tinh của một số cách thức mà một hệ thống hoặc một doanh nghiệp sử dụng, bởi khách hàng của mình, người sử dụng hoặc các hệ thống khác. Một sơ đồ Use Case thể hiện sự liên quan của các Use Case trong hệ thống và cách người dùng có thể thao tác được trong từng Use Case. Mỗi vòng elip trên một sơ đồ Use Case đại diện cho một Use Case và mỗi người thanh đại diện cho một người sử dụng.

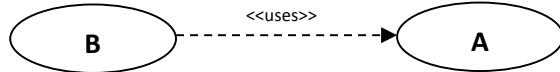
Ký hiệu	Ý nghĩa
	Tác nhân là người

	Tác nhân không phải là người
	Trường hợp sử dụng
	Quan hệ giữa tác nhân và trường hợp sử dụng (tác nhân kích hoạt trường hợp sử dụng)
	Quan hệ giữa các trường hợp sử dụng
	Đường biên hệ thống (tùy chọn)

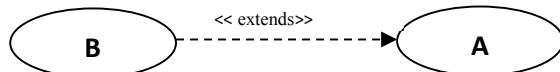
Bảng 2.2 Các thành phần của sơ đồ Use Case

Quan hệ giữa các trường hợp sử dụng trong một sơ đồ hoạt vụ:

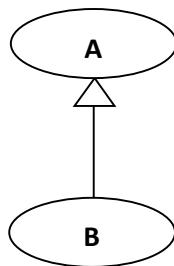
**Quan hệ bao hàm (uses):** Trường hợp sử dụng B được coi là “bao hàm” trường hợp sử dụng A nếu hành vi mô tả B bao hàm hành vi mô tả A. Ta nói B phụ thuộc vào A. Ký hiệu:



**Quan hệ mở rộng (extension):** Nếu hành vi của trường hợp sử dụng B có thể được mở rộng bởi hành vi của trường hợp sử dụng A, ta nói A mở rộng B. Một sự mở rộng thường phải chịu một điều kiện, điều kiện này được biểu diễn dưới dạng một ghi chú (note). Phải cho biết chính xác ở điểm nào của trường hợp đang xét thì mở rộng nó. Ký hiệu:



**Quan hệ tổng quát hóa (generalization):** Một trường hợp sử dụng A là một sự tổng quát hóa trường hợp sử dụng B nếu B là một trường hợp đặc biệt của A. Mọi quan hệ «tổng quát hóa» sẽ được diễn dịch thành khái niệm «thừa kế» trong các ngôn ngữ hướng đối tượng. Ký hiệu:

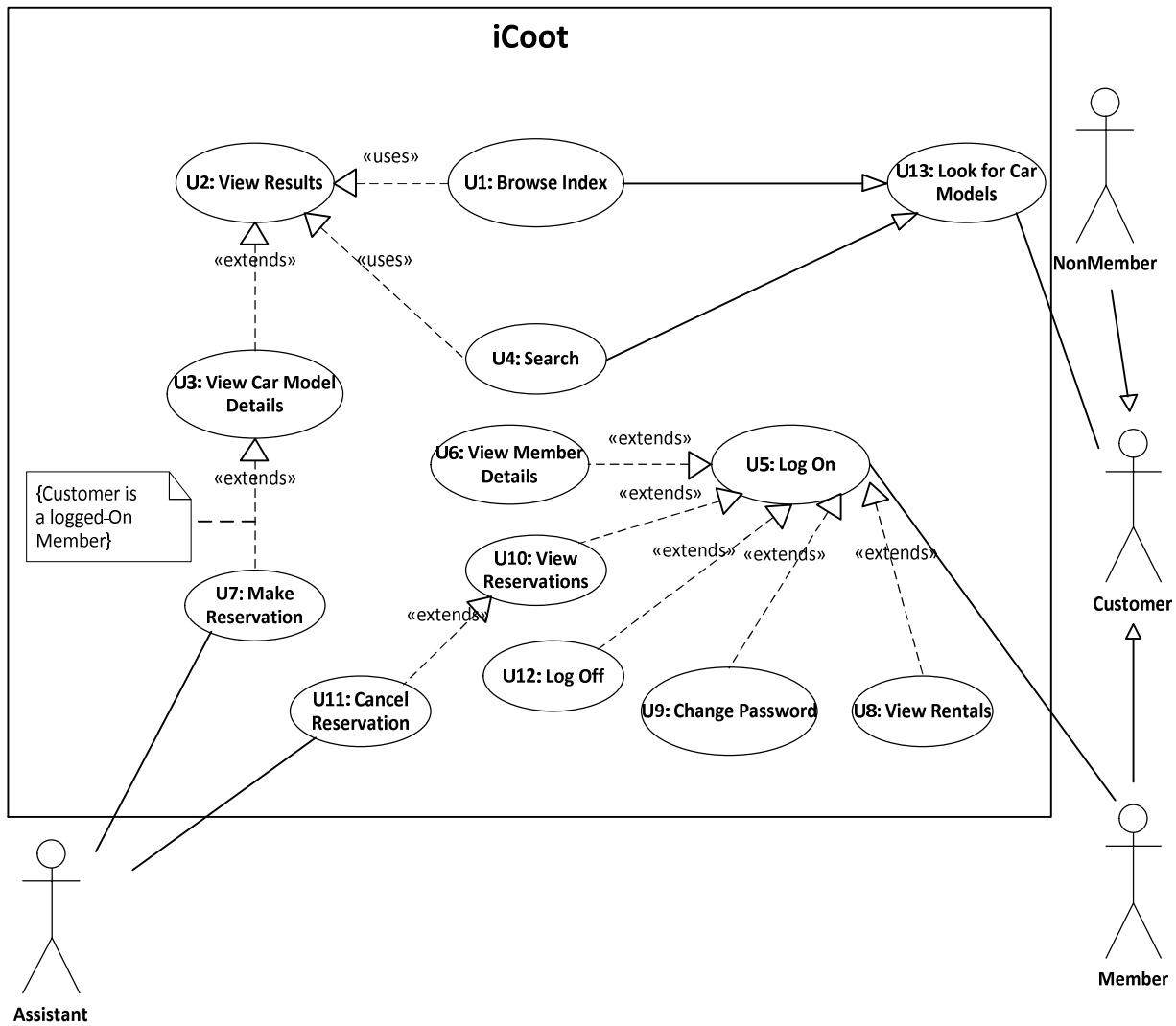


Ví dụ

Hình 2.1 mô tả một cửa hàng cho thuê ô tô qua Internet. Từ hình ảnh này, chúng ta có thể trích xuất rất nhiều thông tin khá dễ dàng. Ví dụ, một trợ lý có thể Đăng ký (mua xe); Khách hàng có thể tìm kiếm các mẫu xe; Thành viên có thể đăng nhập; người dùng phải đăng nhập trước khi họ có thể đặt mua.

Mỗi Use Case bao gồm các bước thực hiện liên quan đến việc sử dụng các hệ thống hoặc nghiệp vụ, chẳng hạn **U7: Make Reservation** và **U13: Look for Car Models**. UML chỉ sử dụng ký hiệu mô tả cho các sơ đồ Use Case nhưng nó không mô tả các bước của mỗi Use Case.

RUP (Rational Unified Process) sẽ mô tả các bước của mỗi Use Case và chi tiết Use Case. Các chi tiết Use Case cho **U3:View Car Model Details** được thể hiện trong Hình 2.2. Xem chi tiết mẫu xe bao gồm một khách hàng lựa chọn một mô hình xe, yêu cầu chi tiết của nó, và sau đó nhận được thông tin cụ thể về mô hình xe.



Hình 2.1 Sơ đồ Use Case

U3: View Car Model Details. (Extends U2, extended by U7.)

Preconditions: None.

1. Customer selects one of the matching Car Models.
2. Customer requests details of the selected Car Model.
3. iCoot displays details of the selected Car Model  
(make, engine size, price, description, advert and poster).
4. If Customer is a logged-on Member, extend with U7.

Postconditions: iCoot has displayed details of selected Car Models.

NonFunctional Requirements:

- r1. Adverts should be displayed using a streaming protocol rather than requiring a download.

Hình 2.2 Chi tiết Use Case

## 2.2.2 Sơ đồ lớp - Mức độ phân tích

Một sơ đồ lớp hiển thị các lớp tồn tại trong nghiệp vụ (trong suốt quá trình phân tích) hoặc trong hệ thống (trong khi thiết kế hệ thống con).

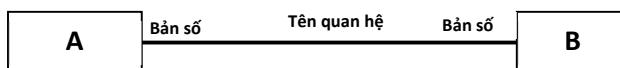
### 2.2.2.1 Các thành phần

Lớp:

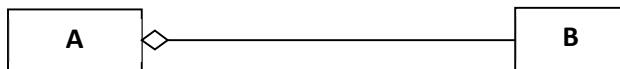
Tên lớp

Quan hệ giữa các lớp:

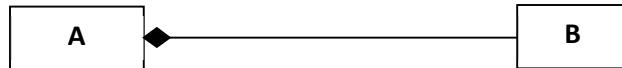
- **Liên kết (association):** Một liên kết biểu diễn mối liên hệ ngữ nghĩa bền vững giữa 2 lớp. Bản số có thể mang các giá trị sau: 1, \*, 1..\*, 0..\*, 0..1. Ký hiệu:



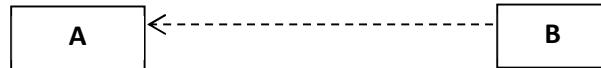
- **Quan hệ kết tập (aggregation relation):** Một kết tập là một trường hợp đặc biệt của liên kết không đối xứng biểu diễn một mối quan hệ «chứa đựng» về cấu trúc hoặc hành vi của một phần tử trong một tập hợp. Ký hiệu:



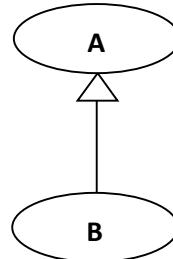
- **Quan hệ cấu thành (composition):** Quan hệ cấu thành còn được gọi là quan hệ kết tập phức hợp, là một quan hệ kết tập đặc biệt. Nó mô tả một sự chứa đựng về cấu trúc giữa các thể hiện. Lớp chứa sẽ chịu trách nhiệm tạo ra, sao chép và xóa các lớp thành phần của nó. Mặt khác, việc sao chép hoặc xóa đi lớp chứa sẽ kéo theo sao chép hoặc xóa các lớp thành phần của nó. Một thể hiện của lớp thành phần chỉ thuộc về duy nhất một thể hiện của lớp chứa nó. Ký hiệu:



- **Quan hệ phụ thuộc (dependancy)**: lớp này (B) phụ thuộc vào lớp khác (A). Ký hiệu:

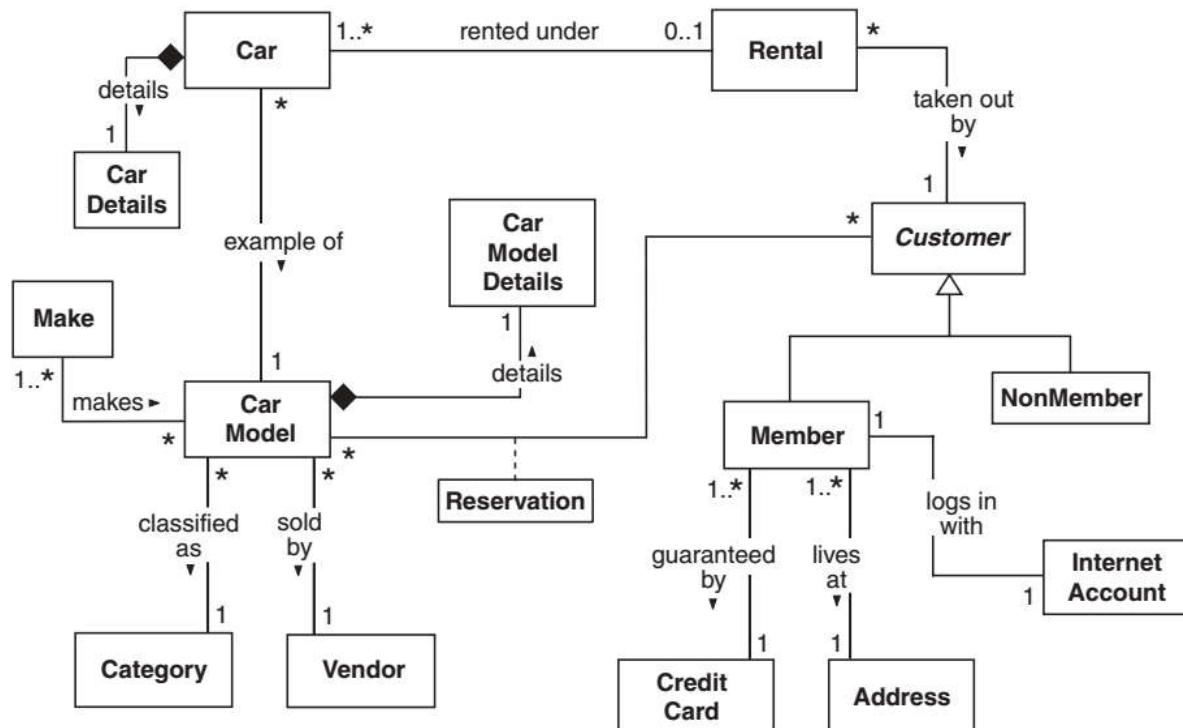


- **Quan hệ kế thừa (inheritance)**: lớp con (B) kế thừa từ lớp cha (A). Ký hiệu:



### 2.2.2.2 Ví dụ

Hình 2.3 là một ví dụ về sơ đồ lớp mức độ phân tích, với mỗi lớp được biểu diễn bởi một hình chữ nhật có nhãn như **Car**, **Rental**,...



Hình 2.3 Sơ đồ lớp mức độ phân tích

Một sơ đồ lớp thể hiện mối quan hệ của các đối tượng của các lớp có thể được kết nối với nhau như thế nào. Ví dụ, Hình 2.3 thể hiện một đối tượng thuộc lớp **CarModel** chứa nhiều đối tượng thuộc lớp **CarModelDetails**, được gọi là chi tiết của **CarModel**.

### 2.2.3 Sơ đồ cộng tác

Một sơ đồ cộng tác thể hiện sự hợp tác giữa các đối tượng như tên gọi của nó.

#### 2.2.3.1 Các thành phần

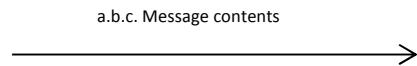
Dòng đời (Lifeline) trong sơ đồ cộng tác gồm:

Ký hiệu	Ý nghĩa
	Tác nhân (actor)
	Giao diện (Boundary)
	Điều khiển (Control)
	Thực thể (Entity)

**Liên kết (link):** mối liên kết giữa các thành phần trong sơ đồ cộng tác. Ký hiệu:

---

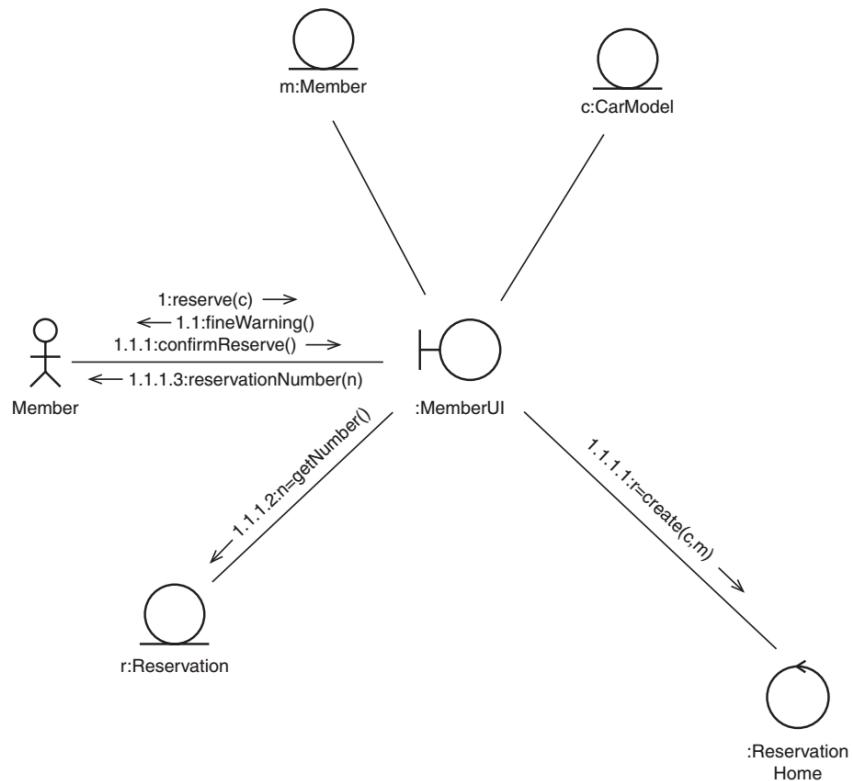
**Thông điệp (message):** biểu diễn biểu thức trình tự (sequence expression) và được ghi trên dấu mũi tên. Ký hiệu:



**Biểu thức trình tự:** được ghi với số thứ tự thể hiện thứ tự hiện thị thông điệp hoặc trình tự thực hiện các thủ tục.

### 2.2.3.2 Ví dụ

Hình 2.4 mô tả quá trình đặt một mô hình xe qua Internet: Một **Member** thông báo **MemberUI** đặt một mẫu xe; **MemberUI** thông báo **ReservationHome** tạo một **Reservation** cho **CarModel** và **Member** hiện tại; **MemberUI** sau đó yêu cầu một **Reservation** mới và trả về cho **Member**.



Hình 2.4 Sơ đồ công tác

### 2.2.4 Sơ đồ triển khai

Một sơ đồ triển khai (xem hình 2.5) cho thấy cách hệ thống hoàn thành sẽ được triển khai trên một hoặc nhiều máy như thế nào. Sơ đồ triển khai có thể bao gồm tất cả các loại tính năng như máy móc, quy trình, tập tin và phụ thuộc.

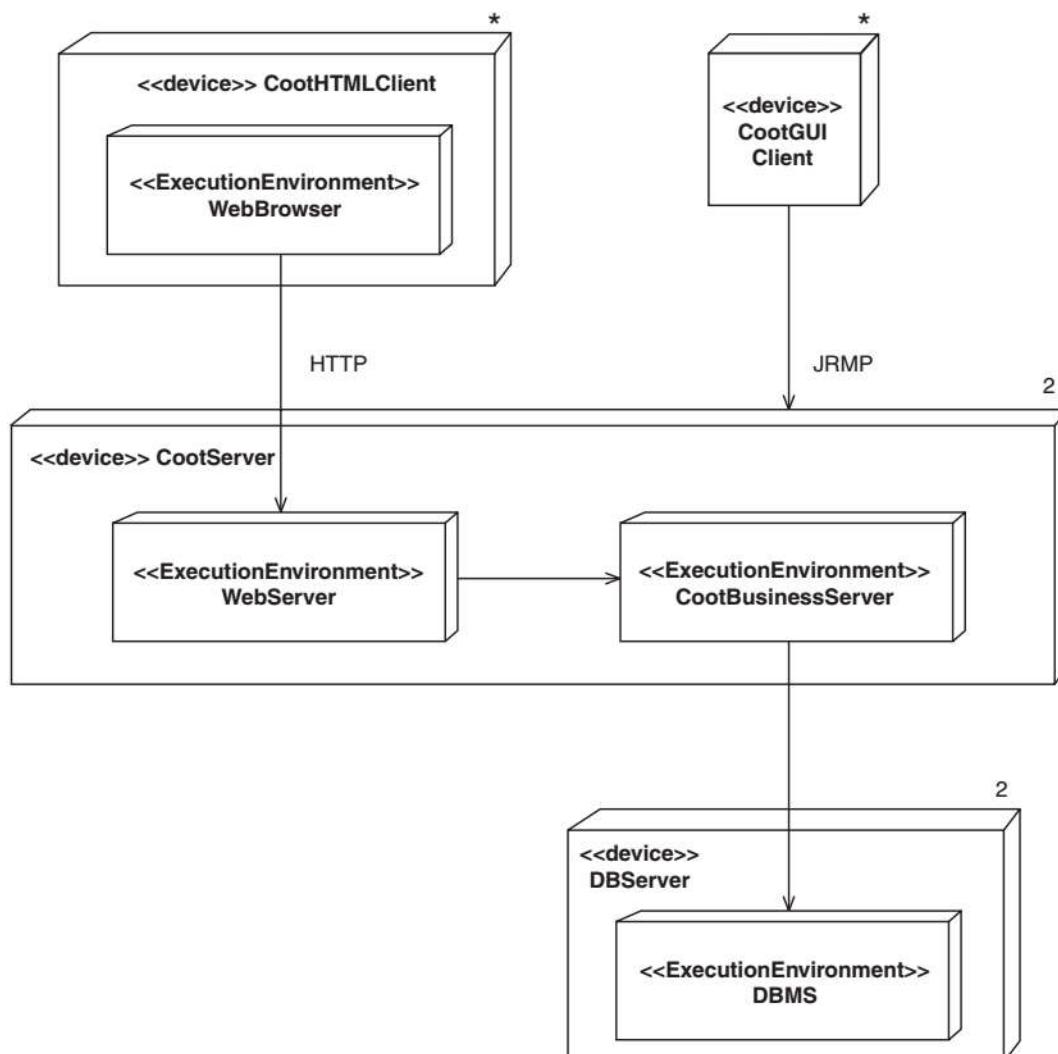
#### 2.2.4.1 Các thành phần

Ký hiệu	Ý nghĩa
	Nút là một đối tượng vật lý (các thiết bị) có tài nguyên tính toán.

→	Một liên kết biểu diễn mối liên hệ ngữ nghĩa bền vững giữa 2 nút.
Bản số: n, *	n: số thiết bị mỗi nút; *: chỉ nhiều thiết bị.

#### 2.2.4.2 Ví dụ

Hình 2.5 cho thấy một số lượng nút **HTMLClient** (mỗi nút lưu trữ một **WebBrowser**) và các nút **GUIClient** giao tiếp với hai máy chủ, mỗi máy lưu trữ một **WebServer** và một **CootBusinessServer**; mỗi **WebServer** giao tiếp với một **CootBusinessServer**; và mỗi **CootBusinessServer** giao tiếp với một **DBMS** chạy trên một trong hai nút **DBServer**.



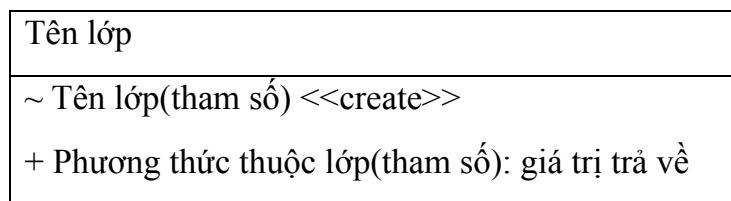
Hình 2.5 Sơ đồ triển khai

## 2.2.5 Sơ đồ lớp - Mức độ thiết kế

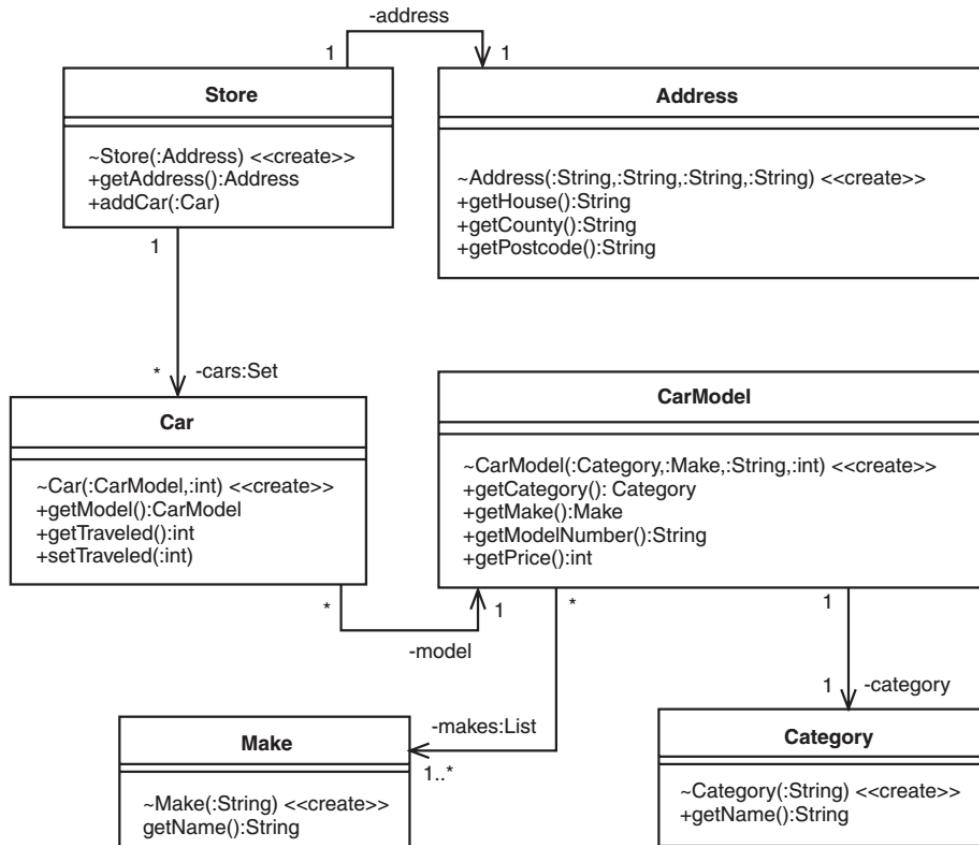
Sơ đồ lớp mức thiết kế thể hiện trong Hình 2.6 sử dụng các ký hiệu giống như sơ đồ lớp mức phân tích trong Hình 2.3. Sự khác biệt duy nhất là sơ đồ lớp mức độ thiết kế sử dụng nhiều hơn các ký hiệu mức phân tích và chi tiết hơn. Sơ đồ lớp mức thiết kế mở rộng một phần của sơ đồ lớp mức phân tích để hiển thị các phương pháp, khởi tạo và điều hướng.

### 2.2.5.1 Các thành phần

Ngoài các ký hiệu thành phần giống như sơ đồ lớp mức phân tích, mỗi lớp ở mức thiết kế được mở rộng thêm bao gồm phương thức khởi tạo (cùng tên với tên lớp) và các phương thức thuộc lớp:



### 2.2.5.2 Ví dụ



Hình 2.6 Sơ đồ lớp mức thiết kế

## 2.2.6 Sơ đồ tuần tự

Một sơ đồ tuần tự cho thấy sự tương tác giữa các đối tượng theo thứ tự. Các sơ đồ cộng tác cũng cho thấy sự tương tác giữa các đối tượng, nhưng trong một cách nhấn mạnh liên kết chứ không phải là thứ tự. Sơ đồ tuần tự được sử dụng trong thiết kế hệ thống con, nhưng chúng đều áp dụng cho mô hình động trong quá trình phân tích, thiết kế hệ thống và thậm chí các yêu cầu.

### 2.2.6.1 Các thành phần

Ký hiệu	Ý nghĩa
	Tác nhân (actor)
	Hình chữ nhật biểu diễn đối tượng (object)
	Đường gạch rời (dashed line) thẳng đứng biểu thị đường đời đối tượng
	Đường thẳng thông điệp (message) nằm ngang nối các đường đời đối tượng
	Ghi chú giải thích

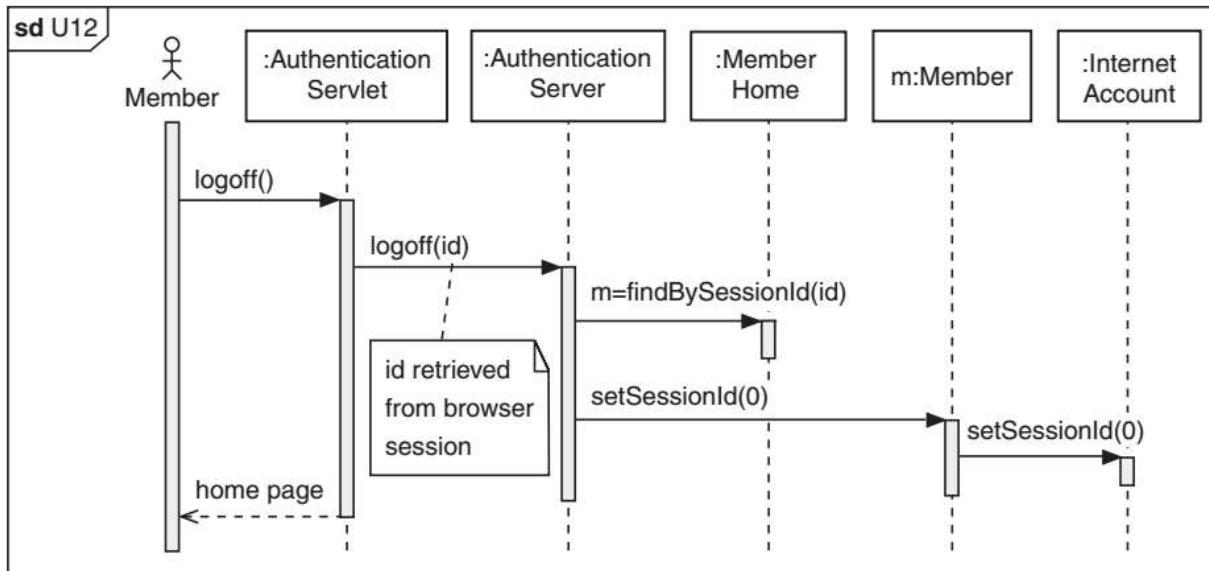
Để đọc biểu đồ tuần tự, hãy bắt đầu từ phía bên trên của biểu đồ rồi chạy dọc xuống và quan sát sự trao đổi thông điệp giữa các đối tượng xảy ra dọc theo tiến trình thời gian.

### 2.2.6.2 Ví dụ

Các sơ đồ trong Hình 2.7 chỉ rõ cách một **Member** có thể đăng xuất khỏi hệ thống. Thông điệp được gán làm nhãn cho các mũi tên giữa thanh dọc đại diện cho các đối tượng (tên các đối tượng được đặt trên cùng các thanh dọc).

Thời gian được thể hiện theo trực ngang của một sơ đồ tuần tự. Hình 2.7 quy định cụ thể: một **Member** yêu cầu **AuthenticationServlet** để logoff; **AuthenticationServlet**

chuyển yêu cầu sang **AuthenticationServer**, đọc id từ phiên trình duyệt; **AuthenticationServer** tìm kiếm đối tượng **Member** tương ứng và thông báo thiết lập phiên id là 0; **Member** thông qua yêu cầu này vào **InternetAccount** của nó. Cuối cùng, **Member** được hiển thị tại trang chủ.



Hình 2.7 Sơ đồ tuần tự giai đoạn thiết kế

### 2.3 Kết chương

Trong chương này, các nội dung chính được giới thiệu như sau:

- Tổng quan UML và áp dụng UML vào các giai đoạn sản xuất phần mềm: Các yêu cầu, phân tích, thiết kế hệ thống, thiết kế hệ thống phụ, đặc điểm kỹ thuật, thực hiện, kiểm tra, triển khai và bảo trì.
- Giới thiệu các sơ đồ UML sẽ được sử dụng trong các chương còn lại như: Sơ đồ Use Case, Sơ đồ lớp – Mức độ phân tích, Sơ đồ cộng tác, Sơ đồ triển khai, Sơ đồ lớp – Mức độ thiết kế, Sơ đồ tuần tự.

### Tài liệu tham khảo

- (Chính) Mike O'Docherty, “Object-Oriented Analysis and Design Understanding System Development with UML 2.0”, John Wiley & Sons, 2005.

2. Trần Đình Quê, Nguyễn Mạnh Sơn, “*Phân tích và thiết kế hệ thống thông tin*”, Học viên công nghệ bưu chính viễn thông, 2007.
3. Phạm Thị Xuân Lộc, “*Phân tích hệ thống hướng đối tượng*”, trường Đại học Cần Thơ, 2009.

## CHƯƠNG 3. XÁC ĐỊNH YÊU CẦU PHẦN MỀM

### 3.1 Giới thiệu

Mục đích của các yêu cầu gồm 2 phần:

- Khảo sát tình hình việc kinh doanh , đầu tiên chúng ta cần làm rõ lí do ( nhu cầu) để phát triển phần mềm. Nếu chúng ta không thể đến với lý do tốt thì chúng ta không nên viết một phần mềm nào cả. Khi chúng ta quyết định làm, sản xuất hệ thống phần mềm chúng ta cần chắc chắn rằng mình hiểu về lĩnh vực kinh doanh và rằng sự hiểu biết của chúng ta đúng với những người tài trợ (sponsor). Đây là cơ hội tốt để chúng ta làm rõ người tài trợ (sponsor) là ai.
- Mô tả những yêu cầu: Điều này không chỉ liên quan đến các chức năng của hệ thống mà còn phát hiện ra những điều bắt buộc (tính chất, thuộc tính) : khả năng thực thi (tốc độ làm việc và xử lý của chương trình), chi phí phát triển , tài nguyên ....

Chúng ta dựa các điều trên để xác định các yêu cầu của phần mềm. Nhưng tại sao chúng ta lại chọn Model the business (Mô hình của lĩnh vực kinh doanh). Hình 3.1 chỉ cho chúng ta thấy một định nghĩa xác định yêu cầu tốt. Hai lập trình viên bắt đầu với vài ý của hệ thống mà theo họ cho là nên được phát triển. Trong khi quan tâm đến sponsor một cách miễn cưỡng. Cách làm thiển cận rất phổ biến với lập trình viên nghiệp dư, những người mà thực sự không biết rõ mình đang làm gì. Nhưng chúng ta hi vọng rằng thái độ chỉ tồn tại đến khi chúng ta trở thành lập trình viên chuyên nghiệp.



Hình 3.1 Những nhà phát triển phần mềm tự học

Xu hướng đào sâu vào việc code không chỉ xuất phát từ arrogance. Mà nó có thể xuất phát từ nỗi sợ : Chúng ta không chắc chúng ta có thể tạo ra được đúng điều mà sponsor cần , nhưng chúng ta biết cái điều mà chúng ta có thể làm : Hi vọng rằng một khi chúng ta đã hoàn thành chúng ta có thể thuyết phục mọi người rằng những gì mà chúng ta làm là những thứ người ta cần. Khó khăn là điều đó nằm ngoài khả năng code của chúng ta ( stay a way from our keyboard) , chúng ta phải chắc chắn rằng chúng ta hiểu về tình hình kinh doanh, sau đó làm việc với sponsor để thống nhất các tính năng của hệ thống sẽ được triển khai là . Cụm từ ‘sponsor’ thường có ý chỉ một người nào đó ,người thích thấy kết quả chương trình được phân phát ví dụ người dùng cuối người quản lý và những người có cổ phần.

Trước khi chúng ta viết một phần của chương trình chúng ta phải tìm hiểu rất rõ về lĩnh vực kinh doanh nơi phần mềm hoạt động. Không có sự hiểu biết thấu đáo về lĩnh vực kinh doanh, chúng ta sẽ thấy khó khăn để phát triển lĩnh vực này. Cụm từ business được sử dụng theo nghĩa mơ hồ. Mặc dù vậy , phải thừa nhận là cuốn sách hướng tập trung vào ngân hàng, người quản trị ,thương mại điện tử, hầu như những cái gì liên quan đến phần mềm yêu cầu. Nếu bạn muốn nghĩ business như là problem domain cũng chẳng có vấn đề gì cả.

Một khi chúng ta đã hiểu về kinh doanh và phác thảo sự hiểu biết của chúng ta thành yêu cầu kinh doanh (business requirements) , chúng ta cần hiểu về phần mềm chúng ta sẽ làm cho người sử dụng .Quyết định những gì nên làm và quan trọng như là những gì chương

trình không nên làm ,sẽ giúp chúng ta tập trung code vào phần cần thiết. Nếu không có sự hiểu biết thấu đáo về những yêu cầu của chương trình chúng ta sẽ bị lãng phí thời gian trong việc viết những phần chúng ta không được trả tiền.

Những yêu cầu của chương trình thường phân thành 2 loại : functional (có chức năng) và nonfunctional (phi chức năng). Những yêu cầu của loại có chức năng ( functional requirememt) là tất cả những điều mà chương trình phải làm, tức là đáp ứng được đầy đủ các yêu cầu cần thiết từ bên ngoài như là: duyệt catalog và dự trữ mẫu xe. Những yêu cầu của loại không có chức năng (nonfunctional requirememet) là tất cả những gì đưa vào chương trình theo lý thuyết, nonfunctional requirememet bao gồm những ứng dụng web cần được hỗ trợ. Việc sử dụng các đoạn video online khác và những file được phép tải dùng cho quảng cáo, giao diện phải thân thiện...

### 3.2 Sự ra đời của một sản phẩm

Chúng ta có thể có đủ may mắn để nhận được một tài liệu chi tiết từ khách hàng thường thì với những cách sắp đặt nội dung riêng. Đơn giản hóa chúng ta có thể đưa ra những ý định dự kiến (mission statement)

Là lập trình viên chúng ta phải chuyển đổi những yêu cầu của khách hàng hoặc miêu tả đầy đủ rõ ràng những yêu cầu của chương trình để phát triển phần mềm . Theo một chuẩn mà khách hàng có thể hiểu được và phê duyệt. Phải thừa nhận rằng sự đầy đủ và rõ ràng khó mà đạt được. Chúng ta không nên trông chờ có thể tiến đến gần thành quả đó trong lần đầu tiên. Tuy nhiên nó sẽ hữu ích để biết rằng chúng ta có một tài liệu miêu tả mọi thứ mà chương trình sẽ làm với ít sẽ làm.

#### Ví dụ:

Vì chúng ta đã tự động hóa quản lý xe trong cửa hàng chúng ta: dùng các thiết bị (using bar codes,counter top terminals and laser reader (đầu đọc laser)). Chúng ta thấy được nhiều lợi ích như sau: Năng suất tăng 20%, xe hiếm khi đi làm và số lượng khách hàng tăng lên nhiều (theo khảo sát thị trường sự nhận thức được tính chuyên nghiệp và hiệu quả).

Internet cung cấp cho chúng ta những cơ hội hấp dẫn ,tăng năng suất giảm giá thành .Ví dụ: Thay vì in những catalog của xe chúng ta có thể đưa những catalog ấy lên web. Nhưng khách hàng vip chúng ta có thể cung cấp dịch vụ đơn giản như việc click vào một nút. Mục tiêu của chúng ta là tiết kiệm trong khu vực giảm giá 15% với những cửa hàng bán chạy.

Trong vòng 2 năm sử dụng sức mạnh của thương mại điện tử, chúng ta hướng đến cung cấp tất cả các dịch vụ của chúng ta trên web. Với việc phân phối và đón khách hàng ngay

tại nhà. Từ đó đạt được mục tiêu cuối cùng là một công ty cho thuê xe ảo. Với giá thành duy trì so với các cửa hàng khách trên thị trường.

Dự định kéo dài 3 giai đoạn chứa đựng nhiều thông tin: Lịch sử của công ty về sự hài lòng của khách hàng được tự động cập nhật hàng ngày (online catalog và đặt chỗ) ,khách hàng vip và nonvip ,lưu lịch sử và lưu lại những mục đích của công ty thuê xe ao.

Phải thừa nhận rằng , giấc mơ về việc quản lý này là một con đường dài (có thể hơn 2 năm khách hàng mới có thể thoái mái với cửa hàng thuê xe ảo của chúng ta). Nhưng ít nhất chúng ta có hai điểm khởi đầu tốt trong việc tìm hiểu và khám phá: Cửa hàng sẽ cung cấp những dịch vụ vào? Và những dịch vụ nào phù hợp với việc phân phối trên internet?

Misson statement (ý định dự kiến) ở trên là nền tảng để case study được sử dụng trong phần còn lại của quyển sách này.

Điểm nhân nổi bật trong Nowhere Car là họ đặc biệt cho thuê xe với những khách hàng vip trong thời gian nhiều hơn. Vì số lượng xe có hạn khách hàng phải đến cửa hàng(có mặt trên web) khi thực sự muốn thuê. Xe sẽ được thuê theo phương thức đến trước thì thuê trước và khách hàng có thể chọn các xe khác còn tồn trong kho. Một cách khác, khách hàng muốn thuê các mẫu xe hiện không còn trong kho thì họ vẫn có thể đặt chỗ. Một assistant trong công ty sẽ liên lạc với khách hàng khi loại xe đó có trở lại. Khách hàng phải đến nhận xe trong vòng 2 ngày(khách hàng phải trả thêm tiền chờ). Không có dịch vụ chuyển và nhận tại nhà (vì lý do bảo hiểm). Mỗi thành viên (người phải đăng ký) đặt hàng qua điện thoại

### 3.3 Use case

Ivar Jacobson đã phát minh ra use case để định nghĩa một khía cạnh của chương trình được dùng trong lĩnh vực kinh doanh. Mặc dù , đầu tiên use case mang tính chất hướng đến tiến trình hơn là đối tượng. Nó được công nhận rộng rãi là một công cụ hữu hiệu cho việc mô tả những chức năng yêu cầu của chương trình. Hầu hết nonfunction requirement được ghi nhận bên cạnh các use case khác có mối quan hệ.

Use case trong quyển sách này chứa đựng những nhân tố quan trọng trong một định dạng phù hợp để học. Trong quyển sách này use case liệt kê sự hiểu biết của chúng ta trong hoạt động kinh doanh – business requirements modeling – và xác định được những gì mà chương trình chúng ta có thể làm – system requirements. Use case trong quyển sách này được mô tả rất gần gũi về những thứ thật sự tồn tại do đó dễ hiểu cho đối tượng không chuyên. Nói một cách khác hệ thống use case sẽ đưa ra nhiều quy tắc chỉ rõ những chức năng chính cần được thực thi vì những cái lợi chính cho người lập trình.

Một use case với người tham gia gọi là actor; sau đó đi sâu vào lĩnh vực kinh doanh rồi có lúc nó sẽ trở lại actor. Hiệu quả của mỗi use case sẽ là giá trị của mỗi actor đó( nếu không tại sao chúng ta lại khởi tạo actor đó ban đầu ?) dĩ nhiên , giá trị đó có ý nghĩa khác nhau đối với mỗi người. Nó có thể là một vài thông tin mà actor muốn lấy lại, một vài hiệu quả mà actor muốn có trong hệ thống, một số tiền , một sự mua bán hay là bất cứ thứ gì tốt thúc đẩy chúng. Khi được điều khiển bởi use case chúng sẽ giúp chúng ta tìm thấy đúng đối tượng thuộc tính và hoạt động (object ,attributes, operation) mà không phải đi theo các quy luật truyền thống.

#### **Ví dụ: Nowhere Cars use case**

Member reserves Car Model là một use case trong lĩnh vực kinh doanh miêu tả thế nào là một thành viên đặt chỗ trước. Theo ví dụ hiện giờ làm thủ tục, nó có thể diễn tả theo một cách nào đó mà có thể ứng dụng cho mọi loại hình kinh doanh, hoặc có những chi tiết phù hợp cho cửa hàng Nowhere Cars .Trong quá trình mô hình hóa lên use case business đây là bước đầu tiên. Một business use case có thể chỉ dẫn đến sự tồn tại của một software system hoặc nó có thể không liên quan gì đến máy tính cả. Ví dụ như lần cuối bạn gọi điện đến cửa hàng thuê xe để hỏi đặt chỗ trước một mẫu xe , bạn không cần quan tâm rằng người đang đầu kia (assistant) thực hiện quá trình giao dịch bằng máy tính hay giấy tờ.

Make reservation là một hệ thống use case được miêu tả như thế nào để chương trình của chúng ta đủ hiểu để sản xuất sẽ được chấp nhận. Chất lượng đặt chỗ trước của Nowhere Cars sẽ được đánh giá cao trên Internet.

Một hệ thống use case miêu tả một dịch vụ mới hay một sự thay thế, hệ thống sẽ cung cấp: Trong ví dụ này một thành viên dùng vài software một trình duyệt web và một sever. Một phân công việc của chúng tôi là chỉ rõ chính xác Input của use case( dữ liệu cần nhập vào) và họ nên chờ đợi để lấy lại dữ liệu.

Vì một mục đích đơn giản use case đặc biệt là hệ thống use case không nên lồng lên nhau. Một use case được thể hiện trong ngôn ngữ tự nhiên, chia thành chuỗi các bước. Lược đồ gắn liền với use case nếu cần thêm sự giải thích

### **3.4 Khía cạnh nghiệp vụ**

Trong phần này chúng ta sẽ thấy như thế nào là kết hợp mô hình kinh doanh giúp hình dung ra mô hình sắp tới chúng ta làm. Một mô hình kinh doanh có thể đơn giản là một biểu đồ lớp (class diagram) biểu diễn các mối quan hệ giữa các thực thể trong kinh doanh. Điều này , thỉnh thoảng quy vào domain model . Một domain model có thể đủ cho một đề án nhỏ, tuy nhiên đối với hầu hết các dự án chúng ta muốn tạo một mô hình business model

tổng thể để thể hiện được hình thức kinh doanh hoặc ít nhất thể hiện một phần những thứ cần phát triển.

Use case không chỉ là một phần của mô hình kinh doanh . Nhưng cách khác phức tạp hơn bao gồm business process (mô hình sử lý việc kinh doanh) và work flow analysis(phân tích luồng công việc) . Use case đơn giản bởi vì để tạo ra nó bạn không cần phải là một chuyên gia mà chỉ cần cảm quan chung và một số điều kiện logic bắt buộc là được.

The use case model cái mà chúng ta sản xuất ở đây sẽ chứa đựng bản thân nó và thêm các phần sau đây:

- Danh sách các actor (với các ghi chú)
- Thuật ngữ chuyên môn
- Lược đồ giao tiếp(không bắt buộc)
- Lược đồ hoạt động(không bắt buộc)

Định nghĩa UML là khái niệm và cú pháp dùng cho activity diagram (lược đồ hoạt động), communication diagram(lược đồ giao tiếp).

Chúng sẽ nhìn vào các thành phần của business model từng cái một. Theo thứ tự mà bạn tạo ra nó đúng với cuộc sống thực. Tuy nhiên luôn luôn nhớ trong đầu rằng điều này là workflow cứng nhắc: Như mọi mặt của lập trình hướng đối chúng ta có thể nhắc lại cho đến khi chúng ta hình dung được một cách cụ thể.

### 3.4.1 Xác định business

Việc đầu tiên chúng ta nên làm là xác định business actor. Một actor hoặc là một người đóng vai trò nào đó trong kinh doanh hoặc là một phòng ban(cửa hàng) hoặc là một hệ thống chương trình riêng biệt.

Lý do để xác định cửa hàng và hệ thống như là một actor là vì: theo logic chúng sẽ tương tác lẫn nhau như con người. Chúng ta quan tâm người nào tạo ra hoạt động tương tác và các bước thực hiện chúng ta không quan tâm actor là người hay là cửa hàng hay là software. Việc xác định các actor giúp chúng ta xác định việc sử dụng các mặt của lĩnh vực kinh doanh. Điều này giúp chúng ta xác định được use case.

Ví dụ Fred Bloggs có thể được một assistant trong cửa hàng Nowhere Car tư vấn mãi cho đến khi assistant hết giờ làm việc nếu Fred Bloggs quyết định thuê xe trước khi assistant hết giờ làm thì anh ta trở thành khách hàng. Đến bước này bạn sẽ làm việc với sponsor (chủ yếu là khách hàng ) để tìm ra thế nào là hoạt động kinh doanh.

**Ví dụ: Nowhere Cars business actor list**

- Assistant(người tư vấn nhân viên công ty) : một người làm việc ở cửa hàng giúp khách hàng thuê xe và đặt chỗ mua xe.
- Customer(khách hàng): một người mà trả tiền cho cửa hàng để nhận được một dịch vụ đúng tiêu chuẩn.
- Member(thanh viên) : Một khách hàng mà identity và danh thiếp có giá trị và người đó được sử dụng các dịch vụ đặc biệt như là đặt chỗ qua điện thoại hay internet
- Nonmember (phi thành viên): Một người mà identity và danh thiếp chưa được kiểm tra và những người đó phải đặt tiền trước để đặt chỗ hoặc có một bản copy giấy tờ bản quyền để được thuê xe.
- Auk: Tồn tại trước hệ thống chi tiết khách hàng , đặt chỗ, dịch vụ cho thuê, catalog hiện có của CarModel.
- Debt department :Văn phòng Nowhere Cars giải quyết phí chưa được trả.
- Legal Department: Văn phòng Nowhere Cars giải quyết những tai nạn nơi mà những người thuê xe của công ty dính líu tới.

### 3.4.2 Viết dự án chuyên môn

Nó là ý tốt để bắt đầu một glossary giữ được lâu dài. Một sự thay thế hiện đại là data dictionary . Cum từ data dictionary bao gồm từ data một trong các từ của hướng đối tượng không được thoái mái và dễ chịu cho lắm, đơn giản là dữ liệu được tạo nên trong cô lập .Sự tồn tại riêng rẽ từ các tiến trình là một trong những phương thức lâu dài: Tốt hơn là nó nên giữ data và xử lý với nhau.

Glossary là một biệt ngữ khó hiểu cho bất cứ cuộc kiểm tra về sự phát triển phần mềm nó cũng cho phép chúng ta gặt giữ các cụm từ , cho phép chúng ta thoái mái hơn trong các tài liệu

**Ví dụ: Nowhere Cars glossary**

Cụm từ	Định nghĩa
Car (business object)	Ví dụ như Car model được giữ ở cửa hàng với mục đích cho thuê
Car model (business)	Một mô hình trong catalog của chúng tôi có giá trị trong việc đặt chỗ

Customer (business object,business actor)	Người trả tiền cho cửa hàng để mong nhận được sự phục vụ của các dịch vụ đạt tiêu chuẩn
Member (business object)	Người sở hữu indentity và danh thiếp có giá trị, và là người được phép sử dụng các dịch vụ đặc biệt như đặt chỗ qua điện thoại, internet

Có một định nghĩa, định nghĩa đó có thể ngắn hoặc dài miễn là phù hợp. Những chú thích actor mà chúng ta đã nhìn thấy cho tới thời điểm này là sự khởi đầu tốt. Nhưng định nghĩa thuật ngữ này sẽ thường có kết cục mang tính tổng quát hơn. Bởi vì hầu hết các cụm từ sẽ được ứng dụng .

Như bạn có thể thấy từ cụm từ case study, bạn có thể ghi nhận mối quan hệ mỗi cụm từ có mỗi giai đoạn phát triển ( business actor, system actor). Phía dưới là danh sách các mối quan hệ bạn có thể sử dụng:

- Business Actor: Một actor xuất hiện trong yêu cầu kinh doanh
- Business object : Một đối tượng xuất hiện trong yêu cầu kinh doanh
- System actor: Một actor xuất hiện trong hệ thống yêu cầu
- System object: Một đối tượng xuất hiện trong hệ thống yêu cầu
- Analysis object: Một đối tượng xuất hiện trong hệ thống số
- Deployment artifact: Vài thứ triển khai trong hệ thống như là file
- Design object: Một đối tượng xuất hiện trong mô hình thiết kế
- Design node :Một máy tính hoặc tiến trình tạo nên một phần trong cấu trúc hệ thống
- Design layer: A vertical portion of or subsystem
- Design package : Một nhóm logical của các lớp được sử dụng để nhận biết sự phát triển

Mỗi loại của đối tượng business , system , analysis hoặc design là sự tinh tế khác nhau. Với một số đối tượng mang phẩm chất cao hơn là một loại . Ví dụ một khách hàng thuê một xe chúng tôi giải quyết một business object là bên ngoài chương trình, phương thức được lan truyền tự nhiên được hiển thị ở từng phần và một đối tượng chương trình nằm ngay trong chương trình là chính nó (chính là class khi chúng ta thực thi).

Trong mỗi thuật ngữ chúng ta dùng để đặt tên cho class chúng ta phải sử dụng nó trong toàn dự án, nó phải rõ ràng mạch lạc.

### 3.4.3 Xác định business use case

Một khi chúng ta có actor, tiếp đến nhiệm vụ của chúng ta là xác định business use case. Mỗi use case là một mảng của lĩnh vực business. Ở giai đoạn này use case có thể bao gồm hai phương thức giao tiếp giữa số của actor, đặc biệt nếu actor mang tính là người . Sau đó chúng ta sẽ thấy rằng hệ thống use case đã có cấu trúc hơn bởi vì một người bình thường thích nói đến trọng tâm công việc hơn là vòng vo.

Không có luật nào quyết định làm thế nào để phá vỡ business thành use case - tri giác chung ,logic ,kinh nghiệm, làm việc với sponsor, trong khi nói chuyện với assistant sẽ giúp chúng ta.

Ví dụ có gắng xác định những nhiệm vụ assistant làm mỗi ngày vì assistant tiếp xúc trực tiếp với customer nên họ biết được cách mà customer sử dụng loại hình kinh doanh mỗi ngày. Tài liệu huấn luyện quản lý về nhân viên, ý định dự kiến, các tài liệu liệt kê yêu cầu cần thiết, bán những sách thông báo nhỏ và các loại khác. Mỗi khi cố gắng tìm use case kéo theo những câu hỏi trong đầu : Cái gì là chìa khóa trong công việc kinh doanh?

#### *Ví dụ: iCoot business use case list*

- B1: khách hàng thuê xe ( customer rents car) ,khách hàng chọn trong những chiếc xe có giá trị ở trong kho để thuê
- B2 : Thành viên đặt mẫu xe(member resveres car model) một thành viên hỏi về mẫu xe và sẽ được thông báo khi mẫu xe đó có giá trị
- B3:Không là thành viên (Nonmember) thì phải trả một khoản tiền để được thông báo khi một mẫu xe có giá trị
- B4: Khách hàng hủy đặt chỗ (customer cancel resveration) khách hàng hủy một mẫu đặt trước bằng điện thoại hoặc trực tiếp đến thông báo
- B5 : Khách hàng trả xe(Customer return car) khách hàng trả lại xe sau khi hết thời gian thuê
- B6 : Customer told carmodel is available . Một khách hàng đến coi xe khi xe đó có giá trị trả lại trong kho
- B7: Xe được thông báo mất (car reported missing) .Một khách hàng hay assistant phát hiện xe bị mất

- B8 : Khách hàng làm lại đặt chỗ mới (customer renew resveration) . Khách hàng làm lại đặt chỗ mới của mình khi họ nhận thấy có sản phẩm mới nổi bật hơn tuần trước.
- B9 : Khách hàng truy cập catalog (customer accesses catalog) . Khách hàng có thể mượn catalog ở cửa hàng hoặc mang về nhà
- B10 :Customer fined for uncollected resveration .Khách hàng thất bại trong khi thu thập một chiếc xe với giá thấp nhất mà họ trả được
- B11 : Customer collect reserved car.Khách hàng nhận được xe đúng với giá thấp nhất họ chấp nhận được
- B12 : Khách hàng trở thành thành viên. Một khách hàng cung cấp chi tiết tài khoản và địa chỉ để làm chứng để trở thành thành viên
- B13 : Khách hàng được thông báo xe họ thuê đã quá hạn . một assistant thông báo cho customer để cảnh báo họ biết rằng họ đã thuê xe quá hạn cho phép một tuần
- B14 : Khách hàng mất chìa khóa. Cấp khóa lại cho khách hàng khi họ mất
- B15 : Làm lại thẻ thành viên: Một assistant tiếp xúc với member để làm mới lại thẻ thành viên của họ khi thẻ của họ hết hiệu lực
- B16 : Xe gặp trục trặc không thể trả lại cho cửa hàng ,xe bị hỏng nặng

B3: NonMember Reserves CarModel.

1. NonMember tells Assistant which CarModel to reserve.
2. Assistant finds CarModel on Auk.
3. Assistant asks for a deposit for the Reservation.
4. Assistant asks for NonMember's License and phone number.
5. Assistant checks License visually.
6. If License looks okay, assistant creates new Reservation and records License number, phone number and a scan of the License in Auk.
7. Assistant gives NonMember a ReservationSlip containing the unique reservation number.

Hình 3.2 Một use case nghiệp vụ trong hệ thống Nowhere Cars

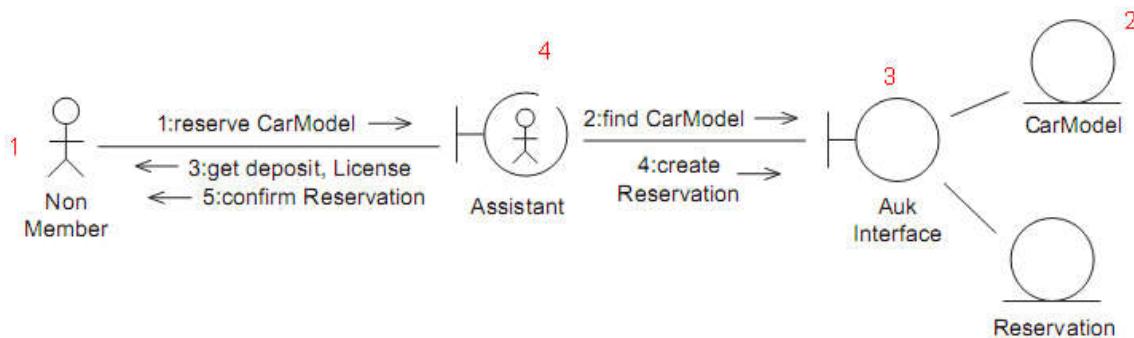
Nhớ rằng trong suốt quá trình kinh doanh, chúng ta không quan tâm đến phương thức mà hệ thống mới chúng ta hoạt động. Giai đoạn này, chúng ta không chỉ đơn giản miêu tả cách mà quá trình kinh doanh hoạt động. Điều này có hoặc không có liên quan đến sự tồn tại của phần mềm.

Lược đồ số là tùy ý.UML không chỉ rõ sự tùy chọn và danh sách , không hàm ý là một loại. Một khi chúng ta có danh sách use case chúng ta có thể liệt kê đến từng bước thực hiện liên quan đến từng use case . UML không chứa nội dung của use case . Do vậy chúng ta tự sử dụng ngôn ngữ tự nhiên từng bước giải thích cấu trúc ngôn ngữ(cấu trúc loop, if then else...) hoặc bất cứ thứ gì.

Sử dụng từng bước thay vì ngôn ngữ tự nhiên sẽ giúp chúng ta giới hạn những điều cần bản của use case. Nếu chúng ta sử dụng ngôn ngữ có cấu trúc . Ta sẽ làm cho chủ thích của mình mang tính thuật toán (computer oriented). Tạo một use case rõ ràng và không phụ thuộc vào tính thực thi .Các bước không cấu trúc sẽ được dùng chi tiết cho use case trong quyển sách này.

### 3.4.4 Minh họa use case trong lược đồ giao tiếp

Ngay khi chúng ta viết chi tiết use case chúng ta có thể cung cấp minh họa use case bằng lược đồ giao tiếp. Một lược đồ giao tiếp chỉ ra những chuỗi ảnh hưởng tác động lẫn nhau giữa các actors và object. Một chuỗi các lược đồ tập trung vào sự tác động lẫn nhau và thứ tự của chúng. Để giảm thiểu sự sử dụng quá nhiều các chi tiết kỹ thuật trong giai đoạn ban đầu, lược đồ giao tiếp phù hợp với mô hình hóa kinh doanh hơn bởi vì UML được thiết kế để ứng dụng cho mỗi tình huống có thể xảy ra, có nhiều lợi ích cho người lập trình viên sử dụng cho mỗi loại lược đồ khác nhau. Trong cuốn sách này để tránh đề cập đến tất cả các chi tiết có thể xảy ra , chỉ sử dụng những phần quan trọng , nhưng chúng ta vẫn minh họa.



Hình 3.3 Sơ đồ giao tiếp cho use case B3. NonMember Reserves CarModel

Hình 3.3 chỉ ra 5 nhân tố giao tiếp. Tính chất của mỗi nhân tố được thể hiện bởi những icon tượng trưng cho nó:

- 1 - biểu diễn actor
- 2 - tượng trưng cho business object hay một thực thể

- 3 - thể hiện boundary (ranh giới) , boundary là thứ dùng để quản lý sự tương tác giữa các nhân tố thường là software hoặc person
- 4 - tượng trưng cho actor theo kiểu người đóng vai trò nào đó cho việc tương tác(chỉ ra rằng actor thường là người hay software)

Không cần quá nhiều kiến thức về các kí hiệu cũng dễ nhận ra rằng hình 3.3 cho biết về việc thuê xe liên quan đến một phi thành viên, một assistant đang hoạt động bằng business boundary , một phần mềm gọi là Auk và hai đối tượng kinh doanh. Trong giai đoạn này vì chúng tôi đang mô hình hóa một cách thức thực hiện nên chúng ta biết rằng giao diện và hệ thống phải được thực thi và triển khai không đề cập đến bất kì phần mềm nào mà chúng ta tự sản xuất ra. Bởi vì chúng ta chưa đi đến giai đoạn đó.

Một đường nối hai nhân tố trong lược đồ giao tiếp cho biết các nhân tố có thể ảnh hưởng lẫn nhau. Vì vậy, chúng ta có thể nhìn từ hình 3.3 rằng: Nonmember yêu cầu dịch vụ từ assistant ,assistant yêu cầu dịch vụ từ Aukinterface ,Aukinterface yêu cầu dịch vụ từ CarModel và reservation ( vì chúng được truy cập từ software).

Đó là những đối tượng bên trong hệ thống chứ không phải bên ngoài ( sự khác biệt đó có thể quan trọng hoặc không quan trọng đối với chúng ta).

Ngay khi biểu tượng và sự kết nối hình 3.3 miêu tả từng tương tác như sự gán nhãn bằng các con số cùng với mũi tên bạn có thể nghĩ đó là thông báo được gửi từ nhân tố khác, các con số thể hiện vị trí thông báo vì chúng có thể giải thích toàn bộ sự kết hợp đó.

- Nonmember hỏi assistant để đặt một mẫu xe
- Assistant hỏi nonmember về tiền và giấy tờ tùy thân của họ
- Assistant yêu cầu Aukinterface để tạo một đặt chỗ nếu mọi điều kiện trên ok
- Assistant xác nhận việc đặt chỗ với nonmember

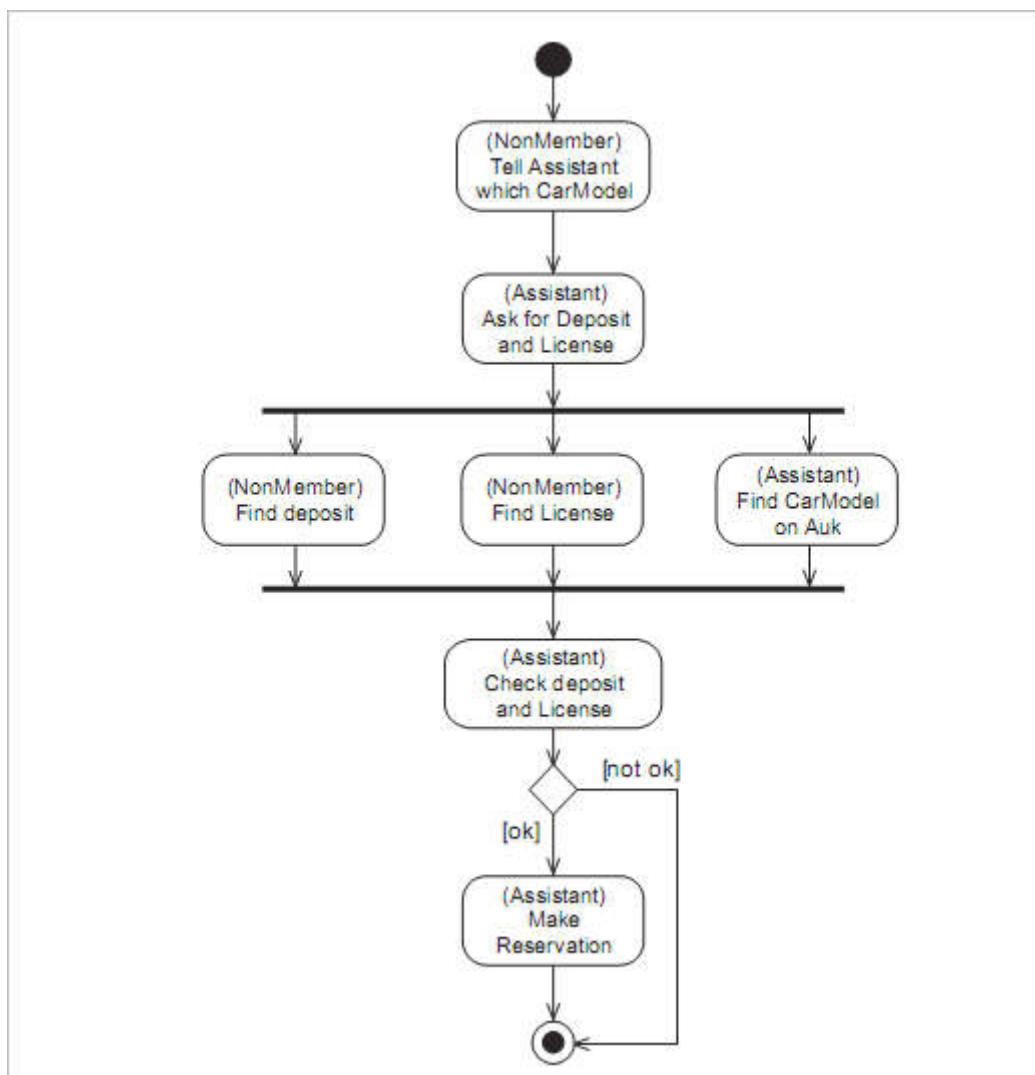
Bạn có thể nghĩ rằng chuỗi các ảnh hưởng nên phù hợp với từng bước trong chi tiết của use case .Tuy nhiên , vì ngôn ngữ tự nhiên không phải là chuỗi từng bước nên không thể có sự phù hợp 1-1 . Có vẻ như mỗi sự tương tác tượng trưng cho sự tổng hợp của một hay nhiều bước.

Mặc dù không đủ sự kết hợp của các use case nhưng lược đồ giao tiếp cũng hữu ích bởi vì nó cũng cung cấp thêm các chi tiết của use case và giúp chúng ta tạo ra các chi tiết trong giai đoạn đầu tiên. Vì sự tương tác lẫn nhau mà chúng ta giải quyết ở bước đầu còn đơn giản nên chúng ta có lý do tạo ra một lược đồ giao tiếp. Để cho ngắn gọn bất kì sự kết hợp nào chúng ta miêu tả nếu là trường hợp bình thường (nomal path) trong use case. Khi chúng ta xử lý các hệ thống use case chúng ta có thể thấy rõ sự bất thường (abnormal path). Nhưng

cho đến bây giờ, chúng ta có thể liên tưởng trường hợp bắt thường bên trong use case. Ví dụ bước 6 trong mô hình kinh doanh ở mô hình 3.2 sẽ bắt đầu nến mọi thủ tục giấy tờ của người đó ok. Ý ở đây là đôi khi các thủ tục của người ấy sẽ không hợp lệ nhưng không được thể hiện.

### 3.4.5 Minh họa use cases trong biểu đồ hoạt động

UML bao gồm các kiểu khác của biểu đồ có thể được sử dụng trong suốt các hoạt động giao dịch. Một biểu đồ hoạt động chỉ ra sự phụ thuộc giữa (song song) các hoạt động như khi chúng ta di chuyển từ một điểm bắt đầu đến đích mong muốn. Chúng tương tự biểu đồ mạng lưới Hình minh họa 3.4 chỉ ra biểu đồ hoạt động sử dụng minh họa một giao dịch của use case.



Hình 3.4 Sơ đồ hoạt động cho use case B3: NonMember Reserves CarModel

Mỗi một khung được khoang tròn trong biểu đồ hoạt động miêu tả một hành động; bắt đầu và kết thúc mũi tên (an edge) cho biết hành động nguồn (gốc) phải hoàn tất trước khi tiếp tục các hành động tiếp theo, chấm đen cho biết điểm bắt đầu cho sự hành động, chấm đen được bao quanh bởi vòng tròn trắng cho biết điểm kết thúc sự hành động , hình thoi biểu trưng cho quyết định – hành động tiếp theo và biểu thị lý do cho hành động đó.

Chỉ ra sự đang bắt đầu và kết thúc của việc thiết lập hành động xảy ra. Mỗi hành động chúng ta có thể được thấy ‘ai hoặc cái gì’ là có trách nhiệm cho hành động đó (bởi việc đặt tên trong phần ngoặc đơn), trước tên của hành động chính nó). Phần tên nhận dạng được phân phân cách với các hoạt động , và có thể được sử dụng để nhận dạng actors, department, system ( hệ thống) hay đối tượng (object). Các phần phân cách có thể chỉ ra nhóm các hoạt động trong các hàng ,các cột hay các ô. Chúng ta có thể thấy dễ dàng từ hình minh họa 3.4:

1. NonMember nói với người bán hàng (the assistant) về CarModel
2. The assistant yêu cầu về tiền đặt cọc(deposit) và giấy phép
3. Trong khi NomMember tìm kiếm deposit và license , người bán hàng tìm kiếm CarModel trong hệ thống Auk
4. Khi việc tìm kiếm hoàn tất, người bán hàng kiểm tra tiền và giấy phép
5. Nếu tiền và giấy phép hợp lệ (valid), the assistant tạo reservation và hành động kết thúc
6. Nói cách khác, hành động đã kết thúc

Giống như với biểu đồ giao tiếp, chúng ta không nên cho rằng sự giải thích làm cho phù hợp chính xác từng bước một của use case.

Trong sự phổ biến với rất nhiều biểu đồ UML, biểu đồ hoạt động có thể được sử dụng nhiều hơn một mục đích. Ví dụ một biểu đồ hoạt động có thể sử dụng xây dựng mô hình giao dịch hoặc tài liệu thuật toán người sử dụng lao động bởi một số đối tượng phần mềm.

### 3.5 Khía cạnh người phát triển

Thứ hai, của yêu cầu chụp mẫu là làm mô hình phần mềm cái mà chúng ta đang phát triển theo yêu cầu cải thiện giao dịch. Bất chấp bạn chọn tài liệu giao dịch sử dụng một mô hình domain đơn giản, mô hình use case, hay việc gì nhiều phức tạp như một mô hình truy cập giao dịch hay phân tích luồng công việc, các thủ tục của hệ thống phần mềm nên sử dụng mô hình use case vì nó đã được chấp nhận rộng rãi và vì các use cases được tạo ra tương đối dễ dàng tạo và cũng dễ hiểu.

Các mô hình use cases của một hệ thống rất phức tạp và nhiều quy tắc hơn so với một giao dịch. Hệ thống use case bao gồm:

- Danh sách actor (với sự miêu tả)
- Danh sách use case(với sự miêu tả)
- Biểu đồ use case
- Chi tiết use case (bao gồm bất cứ các thủ tục có liên quan)
- Tổng quan use case
- Các thủ tục bổ sung ( các thủ tục hệ thống không đúng với bất cứ use case riêng biệt)
- Phác thảo giao diện người dùng
- bảng chú giải thuật ngữ
- Quyền ưu tiên use case

Danh sách ở trên bao gồm một số artifacts (nhân tạo) cái mà chúng ta đã bỏ qua trước đó. Ngay cả cái chúng ta luôn luôn nhìn thấy trong suốt mô hình giao dịch bao gồm nhiều chi tiết trong mô hình hệ thống. Biểu đồ giao tiếp không có sự bao hàm; mặc dù bạn vẫn sử dụng chúng minh họa cho hệ thống use case ở trạng thái đó, Ripple cho họ được hoãn đến một giai đoạn sau trong sự phát triển (dynamic analysis), nơi mà họ đang xem xét quan trọng hơn.

Sử dụng một thời gian nhất định đáng giá sự tồn tại của hệ thống. Như một hệ thống được quy vào là một hệ thống kế thừa bởi vì chúng ta kế thừa nó một phần của giao dịch hiện hành.

### **Ví dụ: Hệ thống kế thừa Cars**

Chúng ta cần quyết định Auk có thể mở rộng hợp lý được hay ko, hoặc chúng ta có thể thay thế chúng hoàn hoàn hay không. Sự quyết định đó không dễ, chúng ta có một hệ thống hoàn chỉnh đi vào hoạt động một thời gian, có những nhân viên quen thuộc với công việc, có thể rất khó để mở ra Auk và thêm vào những tiện ích mới, hoặc viết phần mềm giao tiếp mới.

Phải thừa nhận rằng việc chúng ta quyết định thay thế Auk với đầy đủ hệ thống mới sẽ tương thích với truy cập Internet và sẽ hỗ trợ ‘virtual rental store’ những cái khách hàng yêu cầu. Hệ thống mới được gọi là ‘Coot’.

Với mục đích của cuốn sách này chúng ta không cần nghiên cứu tất cả chức năng của Coot. Thay vì thế chúng ta có thể tập trung vào một số phần của hệ thống cái mà cung cấp những tiện ích Internet đến khách hàng.

### 3.5.1 Nhận biết system actors

Điều đầu tiên chúng ta cần làm là nhận biết và miêu tả các actor hệ thống. Các actor chúng ta nhận biết ở trạng thái bao gồm con người ( và các hệ thống bên ngoài) tương tác với đề xuất của hệ thống, hơn các actor từ hoàn cảnh giao dịch rộng hơn.

#### Ví dụ: iCoot system actor list

- Khách hàng (Customer ): người sử dụng web browser cập nhật đến iCoot
- Thành viên (member) : Một khách hàng sẽ được lưu giữ thông tin cá nhân (tên ,địa chỉ ,chứng minh thư) tại cửa hàng , một thành viên có password internet là mã số thành viên
- Người bán hàng: là một nhân viên tại cửa hàng, họ có thể liên lạc với các thành viên thông báo về tiến trình dành riêng cho mỗi khách hàng

### 3.5.2 Nhận biết hệ thống Use Cases

Từ các actor , chúng ta có thể tìm kiếm Use Cases, và tìm kiếm sự giúp đỡ từ các sponsor. Mỗi Use Case phải có một miêu tả ngắn.

#### Ví dụ: iCoot system use case list

- U1: Vị trí duyệt qua (browse Index) ; một khách hàng duyệt qua một vị trí của CarModels
- U2: Xem kết quả (view results) ; Khách hàng xem tập hợp tìm được của CarModel
- U3: Xem chi tiết ( view CarModel detail) ; khách hàng xem chi tiết kết quả tìm được ở CarModel như là sự miêu tả sự quảng cáo
- U4: Tìm kiếm (search) ; khách hàng tìm kiếm chi tiết phân loại (Categories),hình dáng ,cấu tạo ,cỡ ...
- U5: Đăng nhập (log on) một thành viên đăng nhập vào iCoot sử dụng mã số thành viên và password hiện thời
- U6: Xem chi tiết thành viên ; thành viên xem chi tiết các thông tin được lưu trữ trong iCoot như tên, địa chỉ, chứng minh
- U7: Make reservation ; thành viên duy trì CarModel khi xem chi tiết (a member reserves a CarModel when viewing its details)
- U8: view rentals (xem số tiền thuê) ; thành viên xem tóm tắt Cars hiện thời họ đang thuê.

- U9: Thay đổi password ; thành viên thay đổi được password khi họ đăng nhập
- U10: Xem phần dành riêng (view reservation) ; thành viên xem tóm tắt phần dành riêng
- U11: Hủy Reservation : thành viên hủy một Resevertion
- U12: Đăng xuất : thành viên đăng xuất ra khỏi iCoot

Hệ thống use cases có thể miêu tả trong một biểu đồ use cases chỉ ra các actor và sự kết hợp của actor với sự kiện use cases – điều này giúp chúng ta nhìn ra cách sử dụng hệ thống.Biểu đồ use case trong iCoot hình minh họa 3.5.



Figure 6.5: A simple use case diagram for iCoot

Hình 3.5 Sơ đồ use case đơn giản của iCoot

Trong một biểu đồ use case, mỗi use case là có số và tiêu đề . Bao quan các use case trình bày ranh giới của hệ thống – chúng ta có thể đặt tên hệ thống chỉ ở bên trong box.

Phía ngoài ranh giới hệ thống , là các actor , sự kết hợp giữa use cases và actor sử dụng chúng.

Tổng quan một use case là sự miêu tả thông tin của use cases khít(fit) với nhau : người phát triển có thể sản xuất khi sponsor đã thông qua lược đồ use case. Một use case nhìn chung cho phép đảm bảo có thể hiểu use case mà không cần bất cứ sự thiết kế hiện thời.

#### **Ví dụ: Use case tổng quát cho iCoot**

Bất cứ một khách hàng đều có thể tìm kiếm CarModels trên catalog , hoặc duyệt qua CarModel Index (U1) hoặc searching (U4). Trong trường hợp sau cùng , khách hàng có thể tìm kiếm chi tiết phân loại, hình dáng kích cỡ... Sau mỗi cách tìm kiếm khách hàng đều nhìn thấy kết quả như bộ sưu tập CarModel (U2) , với những thông tin cơ bản như tên CarModel. Khách hàng có thể lựa chọn các xem thông tin về sự đặc biệt CarModel như sự miêu tả và sự quảng bá (U3).

Khách hàng người trở thành thành viên có thể đăng nhập (U5) và được lợi khi truy xuất thêm các dịch vụ. Các dịch vụ thêm là :Making a Reservation (U7), hủy a Reservation (U11), kiểm tra chi tiết thành viên(U6),xem outstanding Reservation (U10), thay đổi password khi đăng nhập (U9), xem Rentals(U8) và đăng xuất (U12).

#### **3.5.3 Chuyên môn hóa Actors**

Một actor có thể chuyên môn hóa (làm thành) (kết thừa các hoạt động từ ) thành actor khác. Được thêm vào nhiều expressive power mô hình hệ thống use case. Ví dụ, chúng ta có thể lựa chọn khách hàng là một khái niệm trừu tượng được chuyên môn hóa bởi Member ; once we have introduced this specialization, nó tạo nên sự giới thiệu khái niệm của NonMember.

Tùy thuộc vào bạn mà sự kế thừa giữa các actor là sớm , hay muộn hay không có. Bạn cần khéo léo lựa chọn mỗi sự kế thừa là lợi ích hơn là hồn đệm. Nhớ rằng tất cả sponsor khả năng hiểu được những gì bạn đang làm , ít nhất mô hình giao dịch phải thông qua con đường tĩnh để phân tích . Nó đảm bảo rằng bạn đã hiểu đúng vấn đề và bạn sẽ chuyển những điều đó tới sponsor. Sẽ không một người lập trình nào vui vẻ được khi khách hàng đều là phi thành viên ? Tùy thuộc vào sự quyết định của bạn.

#### **Ví dụ: iCoot system actor list với sự thừa kế**

- Khách hàng : người sử dụng web browser truy cập vào iCoot.
- Thành viên: Là khách hàng có thông tin được lưu trữ , có password
- Không là thành viên(NonMember) : Khách hàng không là thành viên

- Người bán hàng: một nhân viên tại cửa hàng , có thể liên lạc với thành viên thông báo về tiến trình dành riêng cho mỗi khách hàng.

Chúng ta có thể sửa đổi biểu đồ use case , nhìn thấy sự kế thừa các mối quan hệ giữa actors bằng cách giống như giữa các classes ( xem minh họa 6.6). Lớp khách hàng là một lớp trừu tượng – tên được in nghiêng bởi vì không ai là đúng là một khách hàng, mỗi một người có thể hoặc là thành viên hoặc không là thành viên.

Ngay cả khi khách hàng là trừu tượng, chúng ta vẫn có thể kết hợp với một use case, để chỉ ra rằng tất cả các khách hàng tham gia trong use case đó. Với tư cách là một bên có hiệu lực,một số actor như Nonmember sẽ chỉ có gián tiếp

Bây giờ biểu đồ của chúng ta chỉ ra sự chuyên môn hóa rõ ràng, nhưng nó có ích để chú giải danh sách actor(tác nhân) rất tốt, trong tình huống này thì danh sách được xem tách biệt.

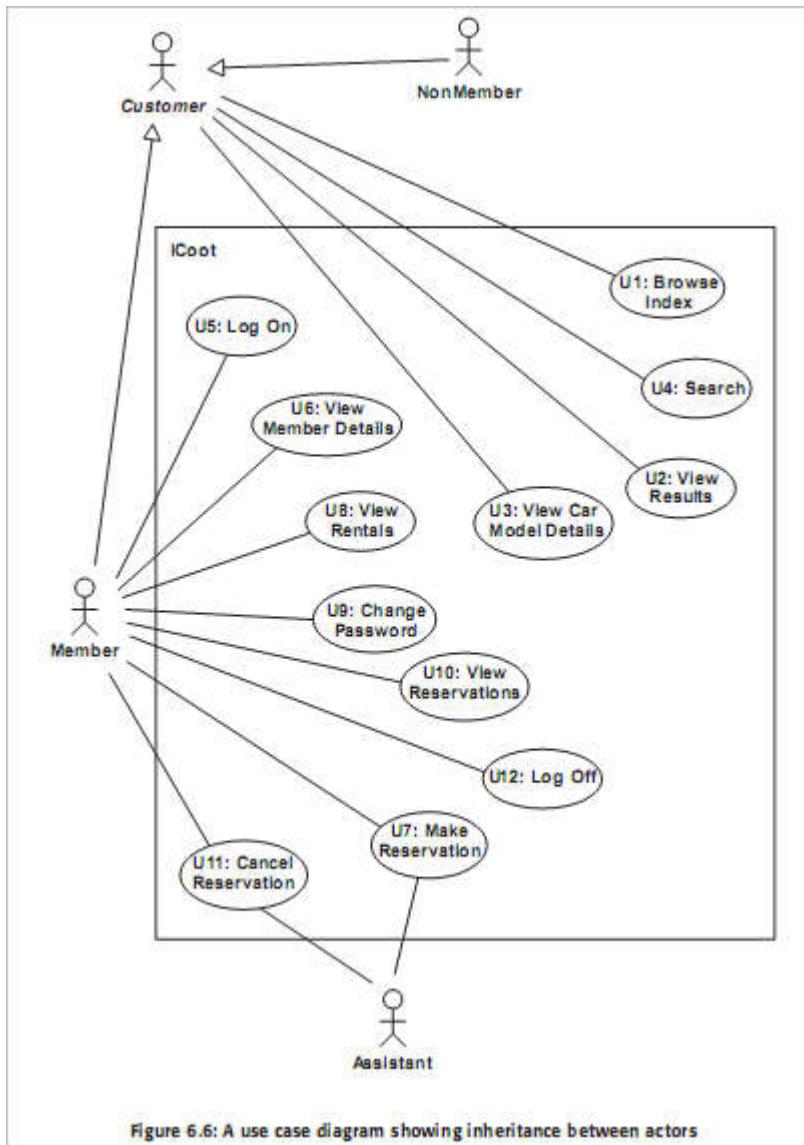
### 3.5.4 Quan hệ giữa các use case

Cũng như sự chuyên môn hóa giữa các actor và sự liên đới giữa actor và use cases. Có ba kiểu quan hệ giữa chính các use case: specializes, includes và extends. Cho phép liên kết nhóm các use case , phân tích rộng rãi các use case, tái sử dụng các hoạt động , và chỉ rõ các hoạt động tự do.

- Chuyên môn hóa (Specializes) : giống như các actor, các use case có thể kế thừa từ một use case khác. Có thể định nghĩa lại các mối quan hệ phức tạp và thêm vào các mối quan hệ mới , có thể giới hạn sự chuyên môn hóa các use case trừu tượng. Một use case trừu tượng thuần túy không có đủ các bước : mục đích duy nhất là nhóm các use case khác. Ví dụ , chúng ta có thể quyết định U1 : Browse Index và U4 : tìm kiếm các trạng thái khác nhau của use case trừu tượng , U13 : tìm kiếm CarModel
- Includes : Một use case quy định các bước bởi một use case khác được gọi là bao hàm use case.
- Extends( mở rộng): một use case đã tồn tại có thể thêm vào các giá trị để tạo thành use case khác được gọi là use case mở rộng . Ví dụ, khi đang xem kết quả(U2) , khách hàng có thể chọn xem chi tiết (U3). Sự mở rộng cho phép chúng ta thêm vào tùy ý – thường xuyên, những gì thêm vào sẽ xuất hiện ở cuối của use case , nhưng chúng có thể tìm thấy ngay đầu , hoặc ở giữa

Sự khác nhau cơ bản giữa Inclusion và Extension ; với Inclusion use case không làm việc không có mục đích ,với extension làm việc hoàn toàn không có có mục đích. Các use case bao hàm trong các use case khác có thể tồn tại độc lập – chúng có thể thực thi ngay

khi qua hướng khác. Một use case được mở rộng bởi một use case khác ,sẽ luôn luông chỉ tồn tại như một sự mở rộng.



Hình 3.6 Sơ đồ use case cho thấy mối quan hệ kế thừa giữa các actor

Bạn không chắc có thể nhận biết được các mối liên hệ use case trong mô hình yêu cầu hệ thống . Ngoài ra ,phụ thuộc vào quyết định của bạn là chúng có thực sự cần thiết và có thể đánh giá đúng khả năng của chúng. Nếu như bạn sử dụng các mối liên hệ như đối với phạm vi khác của hướng đối tượng , sẽ có nhiều cách phân tích các use case trong Inclusions, extension và inheritance. Không một cách nào là chính xác – chỉ là phát triển mô hình tạo nhiều hướng cho bạn và khách hàng của bạn.

#### Ví dụ: iCoot use case relationships

Các use case miêu tả giống như phía dưới. Các Use case trừu tượng và cả kết thúc của các mối liên hệ đã nhận biết – mối liên hệ đó là gì, xem sự chú thích của chúng, người đọc biết cái mà họ sẽ nhìn được từ use case nó tạo thành một bức tranh hoàn chỉnh.

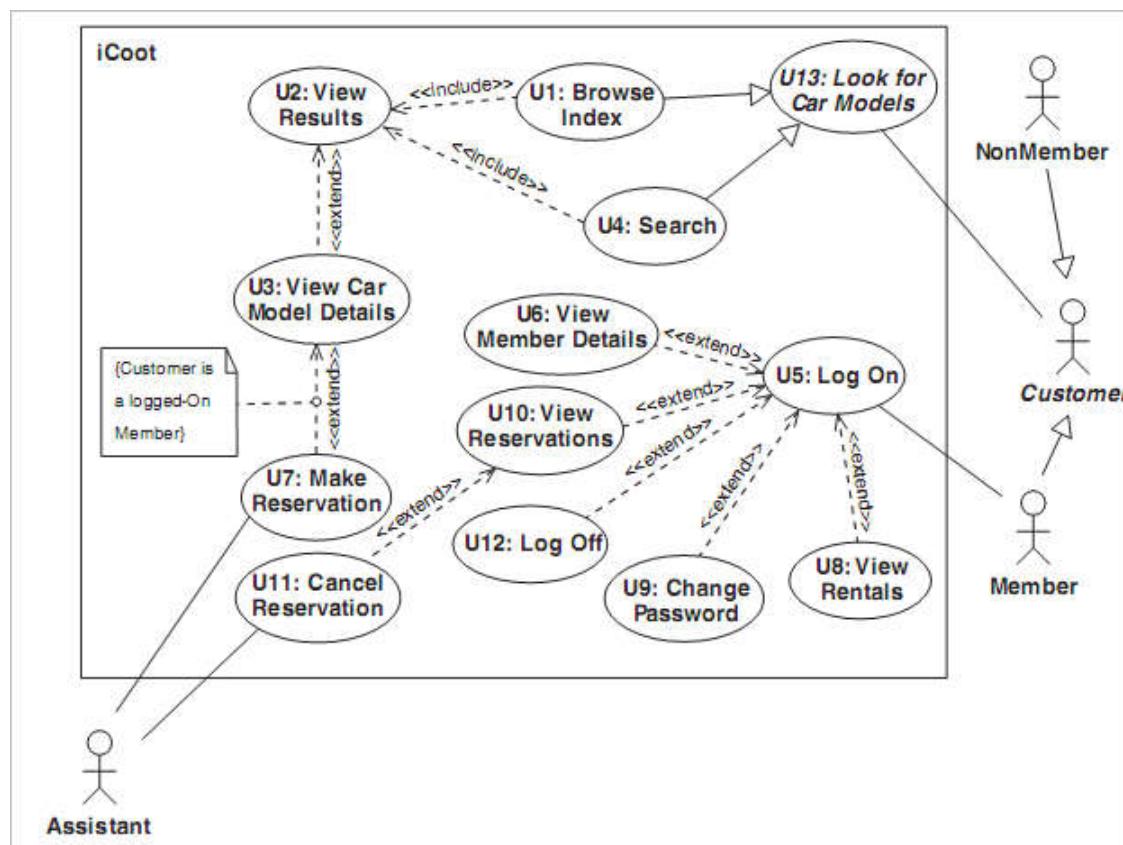
- U1: Browse Index : khách hàng duyệt qua các index của CarModel ( chuyên môn hóa U13 , bao hàm U2)
- U2: view results : Khách hàng xem tập hợp tìm được của CarModel ( bao hàm U1 và U4, mở rộng bởi U3)
- U3 : view CarModel Details : khách hàng xem chi tiết những tập hợp tìm được như sự miêu tả và sự quảng bá ( mở rộng U2, mở rộng bởi U7)
- U4 : Search : khách hàng tìm kiếm chi tiết phân loại (Categories),hình dáng ,cấu tạo ,cỡ ... (Specializes U13,includes U2)
- U5: Đăng nhập (log on) một thành viên đăng nhập vào iCoot sử dụng mã số thành viên và password hiện thời (Extended by U6, U8, U9, U10 and U12.)
- U6: Xem chi tiết thành viên ; thành viên xem chi tiết các thông tin được lưu trữ trong iCoot như tên, địa chỉ, chứng minh (Extends U5)
- U7: Make reservation ; thành viên duy trì CarModel khi xem chi tiết (a member reserves a CarModel when viewing its details) (Extends U3)
- U8: view rentals (xem số tiền thuê) ; thành viên xem tóm tắt Cars hiện thời họ đang thuê. (Extends U5)
- U9: Thay đổi password ; thành viên thay đổi được password khi họ đăng nhập (Extends U5)
- U10: Xem phần dành riêng (view reservation) ; thành viên xem tóm tắt phần dành riêng (Extends U5, extended by U11)
- U11: Hủy Reservation : thành viên hủy một Reseversion (Extends U10)
- U12: Đăng xuất : thành viên đăng xuất ra khỏi iCoot
- U13 :Look for CarModel : Khách hàng tìm kiếm tập hợp CarModel từ Catalog (Abstract ,generalized (suy rộng) by U1 and U4)

Các mối liên hệ của use case được chỉ ra trong biểu đồ use case (hình 3.7). Sự kế thừa giữa các use case được thể hiện bằng đường đứt quãng. Một sự bao hàm (Inclusion) là một nét vẽ ,mở đầu và kết thúc mũi tên từ use case này đến một use case khác , đã được gán nhãn với từ khóa <<include>> . Bởi vậy, U4 : Tìm kiếm Includes, ở một số điểm trong hành động của nó, trong tất cả các bước của U2 : Xem kết quả. Sự mở rộng cũng tương tự

như sự bao hàm ,quan hệ mở rộng được biểu thị bằng đoạn thẳng thẳng đứt quãng với mũi tên trỏ về phía use được dùng để mở rộng , được gán nhãn <<extend>> U3: View Carmodel details là thêm tùy ý U2: View Resutls.

Mặc dù ta có thể làm giảm hai bao hàm xuống thành bao hàm đơn , từ U13 đến U2, nhưng sẽ bị cho rằng không được rõ ràng (dễ hiểu).Use case trừu tượng có thể có nhiều bước, một số chúng ta cần cẩn trọng tránh.

Trong trường hợp này, sự mở rộng chỉ chấp nhận dưới một số điều kiện nào đó. Chúng ta có thể thấy nó bởi sự thêm vào một comment UML ( trông giống như một mảnh của tờ giấy) có điều kiện chi tiết. Một Comment (lời bình, chú thích) có thể chứa nội dung văn bản, có thể kết nối đến một điểm trên biểu đồ bằng đường kẻ, giới hạn với một vòng tròn nhỏ thể hiện sự gia nhập. Các điều kiện trong UML là rõ ràng có sự ràng buộc và có thể được bắt đầu bằng ngôn ngữ tự nhiên, hay giải mã ,hay trong định dạng ngôn ngữ ràng buộc đối tượng của UML ( UML's format Object Constraint Language – OCL) .Hình 3.7 , ngôn ngữ tự nhiên được sử dụng . Như bạn có thể thấy U7: Tạo Reservation chỉ khi được chấp nhận cho Member , những đối tượng đã đăng nhập vào hệ thống.



Hình 3.7 Sơ đồ use case hoàn chỉnh của iCoot

Thông thường, bắt đầu và kết thúc đường vẽ có mũi trong UML biểu thị một sự phụ thuộc the source relies on the target in some way. Nếu mục đích thay đổi thì nguồn sẽ bị ảnh hưởng. Về phần các use case , sử dụng của kí hiệu sự phụ thuộc là có liên quan và không hoàn toàn đúng.Ví dụ , sự mở rộng use case không cần thiết phụ thuộc vào với một biểu đồ use case, chúng ta nói ‘mối liên hệ use case’ hơn là nói sự phụ thuộc . Điểm cuối cùng về sự chú thích : Về sự bao hàm (inclusion) ,use case phụ thuộc vào mục đích ngược lại với sự mở rộng, sự phụ thuộc là vào nguồn – đây có thể là nguyên nhân của sự nhầm lẫn.

### **Ví dụ: Use case hoàn chỉnh cho hệ thống iCoot**

Bất cứ khách hàng có thể tìm kiếm CarModel trong catalog, bằng cách duyệt qua CarModel index(U1) hoặc bằng tìm kiếm (U4). Trong tình huống cuối cùng , khách hàng khách hàng tìm kiếm chi tiết phân loại (Categories),hình dáng ,cấu tạo ,cỡ ...Sau mỗi cách , thì khách hàng đều nhìn thấy kết quả. Khách hàng có thể chọn thêm cách xem thông tin ngoại lệ CarModels như miêu tả chi tiết và sự quảng bá (U3)

Các khách hàng thuộc trong hai loại: là thành viên và không là thành viên.

Khách hàng người là thành viên có thể đăng nhập(U5) và truy cập thêm các dịch vụ. Các dịch vụ thêm như : tạo một Reservation(U7) , hủy Reservation(U11), kiểm tra chi tiết thành viên (U6),xem Reservation nổi bật (U10), thay đổi password đăng nhập(U9),xem Rentals nổi bật(U8), và đăng xuất(U12).

Duyệt index và tìm kiếm(search) CarModels là hai cách khác nhau trong việc tìm kiếm (look for) CarModel. Yêu cầu xem chi tiết CarModel, khách hàng phải xem kết quả tìm kiếm của các mô hình (qua duyệt và tìm kiếm các hướng).

Trong yêu cầu đặt trước một CarModel , một thành viên phải xem chi tiết của nó (NonMember không thể tạo Reservation (vùng dành riêng), ngay cả khi họ đang xem chi tiết). Trong yêu cầu hủy Reservation, Member phải xem những Reservation nổi bật của họ.

### **3.6 Hệ thống chi tiết use case**

Khi đã xác định được use case và thấy được sự phù hợp giữa chúng như thế nào chúng ta cần phải chỉ ra các chi tiết . Do UML không chỉ ra các chi tiết nào của use case nên kể đến hay các chi tiết đó nên được sắp xếp như thế nào, một lựa chọn đã được đưa ra dựa trên thị yếu và kinh nghiệm. Theo Ripple, hệ thống chi tiết use case bao gồm:

- Use case là số và tiêu đề
- Use case có là trừu tượng hay không

- Preconditions (điều kiện mà phải được thỏa mãn trước khi use case được thực hiện)
- Các bước của chúng (tại nơi mà chúng ta cho rằng các Preconditions đã được đáp ứng)
- Postconditions (điều kiện mà được đảm bảo sau khi hoàn tất một use case)
- Các tiền trình bất thường và cách xử lý cho mỗi tình huống (mặc dù các tiền trình là bất thường, song chúng ta cũng cần tính đến nếu chúng là quan trọng để chúng ta có thể chỉ ra phản hồi của hệ thống trong các trường hợp)
- Các yêu cầu phi chức năng có liên kết với use case hiện tại.

Minh họa 3.8 thể hiện một định dạng chi tiết của use case được sử dụng trong tài liệu này. Trong tất cả các thành phần, chỉ có các trường như: số, tiêu đề, preconditions, các bước, postconditions là bắt buộc; các phần còn lại có thể lược bỏ nếu chúng không thể hiện giá trị gì (null).

```
Number, Title (relationships)
Preconditions
Steps
Postconditions
Abnormal paths
Nonfunctional requirements
```

Hình 3.8 Định dạng chi tiết use case hệ thống

Minh họa 6.9 thể hiện bốn loại iCoot use case theo dạng đã được định sẵn. Những hệ thống use case này chỉ ra chi tiết quan trọng hơn loại use case thương mại mà chúng ta đã biết trước đây. Điều này phản ánh một sự thật rằng chúng ta đang cố gắng để mô tả cụ thể hơn một mô tả đơn thuần: Chúng tôi muốn những dịch vụ mà hệ thống chung cấp sẽ chính xác hơn để bỏ đi những công đoạn mò mẫm cho các nhà phân tích và thiết kế.

Cho những mục đích của cuốn sách này, chúng tôi sử dụng một chuỗi các bước được viết bằng ngôn ngữ đời sống thông dụng . Tùy vào yêu cầu cá nhân , bạn có thể thêm vào từng bước những cấu trúc dữ liệu với các điều kiện và vòng lặp đi kèm (ví dụ như: if – then- else hoặc repeat – until)

Khi viết về chi tiết use case điều quan trọng là chúng ta phải xác định được chức năng hệ thống, nhưng không là các chức năng đã được định sẵn : Ví dụ, nếu chúng ta tính đến các bước như là 2.Khách hàng click vào biểu tượng < Chi tiết > chúng ta sẽ hạn chế “the user interface designer”

Trừ khi đó đó là một yêu cầu bắt buộc, bạn nên sử dụng các từ thông dụng như là: Lựa chọn, chỉ định, làm theo hay hiển thị.

Bất kì một người thiết kế bình thường nào cũng sẽ có thể đưa ra những quyết định đã được thông báo trước về việc chính xác thì những người thu nhập thông tin đang cố gắng cụ thể như thế nào: ví dụ người thiết kế có thể thực hiện các bước theo một trình tự khác nhau hoặc đồng thời; miễn là kết quả là như nhau.

Trường hợp tương tự như thế này có thể xảy ra – chúng ta có thể thấy ở U5: Đăng nhập; ba bước đầu tiên có thể thực hiện theo bất cứ trình tự nào vì nó là giống nhau với tất cả mọi người khi đăng nhập.

### 3.6.1 Preconditions, postconditions and Inheritance

Do đã phải xem xét đến sự kế thừa giữa các use case, chúng ta cần phải tự hỏi việc chuyên môn hóa ảnh hưởng tới các Preconditions, postconditions như thế nào? (Có một lời khuyên cho bạn rằng: Chỉ nên thừa kế từ những use case trùu tượng, không tuân theo các bước, những use case như vậy vẫn có thể có Preconditions, postconditions – như bạn có thể thấy ở minh họa 3.9). Dưới đây là các quy tắc:

- Khi một use case này lại được sử dụng (chuyên biệt) cho một use case khác, nó được thừa hưởng các Preconditions của thế hệ use case trước – như là một điểm khởi đầu. Bất kì Preconditions được thêm bởi use case thế hệ sau (con) chỉ làm (cho các Preconditions ban đầu) yếu đi. (chúng được kết nối với nhau bằng việc sử dụng ‘OR’).
- Đối với các postconditions, điểm khởi đầu của thế hệ sau (con – kế thừa) lại là điều kiện postconditions của thế hệ trước đó. Bất kì postconditions được thêm bởi thế hệ sau chỉ làm mạnh thêm những postconditions của use case mà nó thừa hưởng (chúng được kết nối với nhau bằng việc sử dụng ‘AND’).
- Các Preconditions, postconditions được cộng thêm bởi thế hệ sau thì không ảnh hưởng gì đến Preconditions, postconditions của thế hệ trước.

Trong danh sách kể trên, quy tắc thứ 3 hiển nhiên là đúng, theo lý thuyết Hướng đối tượng mà bạn đã biết. (đòi sau không tác động đến hoạt động của đòi trước) nhưng quy tắc thứ 1 và 2 có thể gây ngạc nhiên cho bạn. Một cách không chính thức, lý do mà chúng ta làm yếu đi (đối với Preconditions) hoặc mạnh thêm (đối với postcondition) là thế hệ use case sau có nhiệm vụ đối với người đọc của use case thế hệ trước chứ không phải là để kích thích bất kì sự ngạc nhiên nào. Ví dụ U13: tìm theo các đòi xe hơi – không có các postconditions, nếu U14: việc tìm kiếm lại có một postconditions là : không được làm điều đó (tìm kiếm) vào các ngày thứ 3. Một người nào đó thực hiện việc tìm kiếm xe hơi

theo các đời xe vào ngày thứ 3 sẽ có kết quả tìm kiếm: “Rất tiếc, theo “Tìm theo mẫu xe”, tôi có thể tìm vào một thời điểm khác”.

**U3: View CarModel Details. (Extends U2, extended by U7.)**

**Preconditions:** None.

1. Customer selects one of the matching CarModels.
  2. Customer requests details of the selected CarModel.
  3. iCoot displays details for the selected car model (makes, engine size, price, description, advert and poster).
  4. If Customer is a logged-on Member, extend with U7.
- Postconditions:** iCoot has displayed details of selected CarModels.
- Nonfunctional Requirements:**
- r1. Adverts should be displayed using a streaming protocol rather than requiring a download.

**U5: Log On. (Extended by U6, U8, U9, U10 and U12.)**

**Preconditions:** Member has obtained a password from their local Store.

1. Member enters their membership number.
  2. Member enters their password.
  3. Since iCoot must enforce one logon for a Member, Member can choose to steal (invalidate and thus take over from) an existing session.
  4. Member elects to log on.
- Postconditions:** Member is logged on.

**Abnormal Paths:**

- a1. If the membership number/password combination is incorrect, iCoot informs Member that one of the two is incorrect (for security, they're not told which one).
- a2. If the membership number/password combination is correct, but Member is already logged on and they have not elected to steal, iCoot informs Member.

**U13: Look for Car Models (Abstract, specialized by U1 and U4.)**

**Preconditions:** None.

**Postconditions:** Customer has been presented with summaries of retrieved CarModels.

### Hình 3.9 Chi tiết một vài use case của iCoot

Một precondition của thẻ hệ trước cung cấp một sự đảm bảo cho người dùng , và nó sẽ không hợp lý (hoặc có một lý do nào) để cho thẻ hệ sau có thể hoặc làm giảm đi sự bảo đảm đó.

Ví dụ :U4 Tìm kiếm có một postcondition : khách hàng đã được giới thiệu các đời xe hơi trùng khớp theo đúng ý của khách , vậy thì không có lý do gì để thẻ hệ sau thêm vào một đời xe hơi được lựa chọn ngẫu nhiên.

Quy tắc số 1 và 2 ngụ ý chỉ ra rằng nếu các thẻ hệ trước: không có các Precondition (không giới hạn khi các use case hoạt động) thì thẻ hệ sau của nó, phải có

Preconditions:không . Nếu thẻ hệ trước có postconditions :không (không bảo đảm về kết quả) thì thẻ hệ sau có thể chỉ ra bất cứ postconditions nào phù hợp với nó.

Tóm lại , khi một use case sử dụng chuyên cho một use case khác bạn phải xem xét cẩn thận postconditions và Precondition của thẻ hệ trước.

### 3.6.2 Những yêu cầu bổ sung

Hầu hết các trường hợp, nếu có thể bạn phải gắn kết các yêu cầu phi chức năng với một use case cụ thể.Những yêu cầu phi chức năng mà không thích hợp với bất cứ use case nào có thể được lưu lại ở một tài liệu yêu cầu bổ sung như đã trình bày ở minh họa 3.10.

#### Supplementary Requirements

---

- s1. The client applet must run in Java PlugIn 1.2 (and later versions).
- s2. iCoot must be able to cope with a catalog of 100,000 car models.
- s3. iCoot must be able to serve a million customers simultaneously with no significant degradation in performance.

Hình 3.10 Những yêu cầu bổ sung cho iCoot

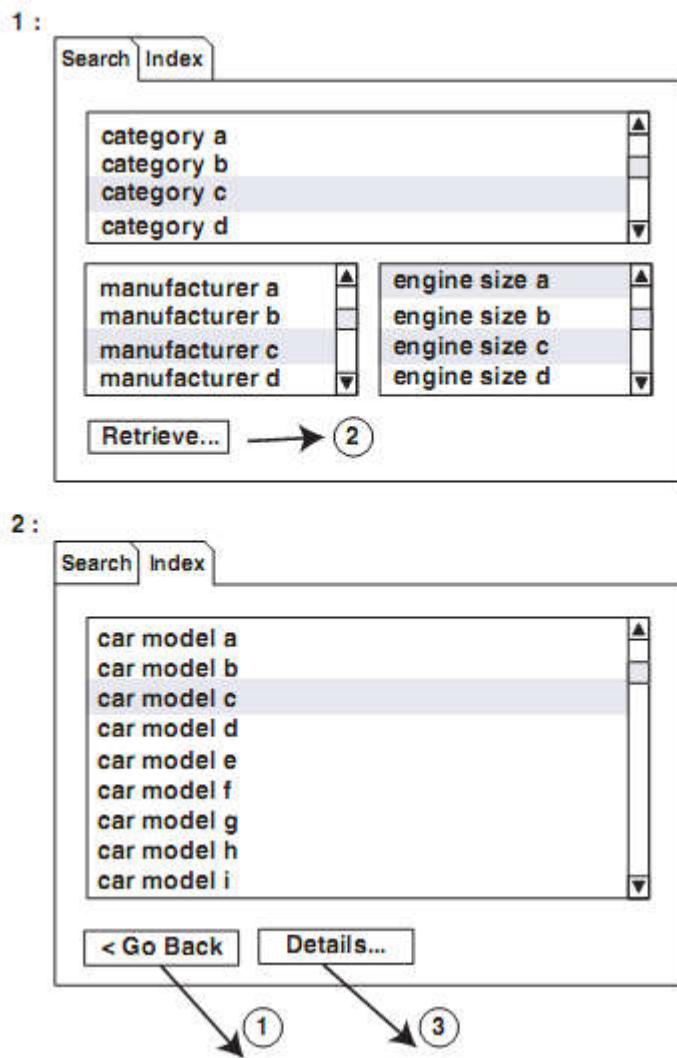
### 3.6.3 Phác thảo interface cho người dùng

Nghĩ về những thiết bị tương thích người dùng cho hệ thống có thể giúp chúng ta làm rõ use case. Những giao tiếp có thể được thảo luận với những nhà tài trợ của chúng ta ở giai đoạn khởi đầu, và kết quả được lưu trữ như là user interface sketches; những phác thảo này có thể được coi như là những chức năng hơn là những thiết kế GUI chuyên nghiệp nó giúp chúng ta xác định và phân chia partition theo cách mà có thể thực hiện được theo yêu cầu cá nhân. Ví dụ sketche 1 trong minh họa 6.11 mô tả một user interface mà cho phép người dùng chọn một hoặc nhiều hơn các chủng loại về nhà sản xuất, kích cỡ động cơ bằng cách click vào nút < Retrieve>, chúng ta đè cập đến sketche 2 thể hiện danh sách đời xe hơi trùng khớp; click vào nút < Go back>, đưa chúng ta trở lại sketche 1; click vào nút <Details > đưa chúng ta đến sketche 3 (không thể hiện) và cứ như vậy.

Vì use case và interface đều tượng trưng cho một sự phân chia của chức năng hệ thống chúng ta nên duy trì một kế hoạch rõ ràng giữa chúng, một kế hoạch chi tiết mà tồn tại suốt quá trình thực hiện.Ví dụ với iCoot, chúng ta có 3 loại truy cập : truy cập thành viên, truy cập không thành viên và truy cập hỗ trợ. Việc này yêu cầu 3 use riêng biệt.

Nội trong mỗi user interface, chúng ta nên cung cấp một cửa sổ hoặc một panel mà tương thích đối với mỗi use case (việc lựa chọn cửa sổ, panel và các thiết bị khác dĩ nhiên

là vấn đề thiết kế). Ví dụ phác thảo ở minh họa 6.11 thể hiện một widget theo phong cách notebook tương trung cho một interface không thành viên. Mỗi use case này thì được phân chia theo trang của nó theo notebook cả hai use case này bao gồm xem kết quả, use case khác thì có chính panel của nó. Chúng ta mong đợi: Search, index, results, panel được tái sử dụng ở interface thành viên. ( Để dễ xác định hơn, interface hỗ trợ sẽ giống như Auk interface hiện tại).



Hình 3.11 Phát thảo giao diện người dùng của iCoot

### 3.6.4 Ưu tiên use case hệ thống

Đó là một ý kiến hay đặc biệt là trong ngữ cảnh của một tiến trình phát triển lớn dần, để xếp hạng yêu cầu hệ thống theo thứ tự ưu tiên của chúng. Với việc làm mẫu use case, điều hiển nhiên phải làm là sắp xếp use case mỗi cái sau đó có thể được cho điểm để chỉ ra

mức độ khẩn cấp của nó. Những sự ưu tiên và khẩn cấp có thể dùng để giúp lập kế hoạch cho phần còn lại của việc phát triển và bắt cứ phần tăng thêm nào khác .

Một kĩ thuật chấm điểm hiệu quả là đèn giao thông:

- Xanh: use case phải được thực hiện trong số current increment. Không thực hiện được điều này có nghĩa là dự án đã thất bại hoặc công việc tiến tới mục tiêu thấp nhất.
- Vàng : là lựa chọn thứ yếu trong current increment và chỉ nên được thực hiện chỉ khi mà use case xanh đã đã được hoàn thành ( chúng là những điểm thường cộng thêm mà chúng ta có thể dùng để gây ấn tượng với nhà tài trợ) bất kì use case đèn vàng nào mà chưa hoàn thành vào ngày bàn giao sẽ phải hoàn toàn bị bỏ đi (do việc thực hiện một phần có vẻ như là kém chuyên nghiệp)
- Đỏ: use case sẽ không được thực hiện trong số current increment trừ khi thời gian cho phép nó nằm ngoài lĩnh vực của current increment và sự cho phép nghiêm túc thì không có khả năng được làm cho chúng

Trong thực tế sự ưu tiên use case (mức độ khẩn cấp không chỉ được dựa trên sự mong muốn mà còn được dựa trên bao nhiêu nỗ lực cấu trúc hệ thống và mã hóa mà mỗi use case này sẽ bỏ vào mỗi số cộng thêm hiện tại: lựa chọn các ưu tiên đòi hỏi các kĩ năng , kinh nghiệm và khả năng phán đoán nhất định. Không có gì sai với việc đặt những use case dễ hơn lên trước nó sẽ giúp chúng ta hiểu hơn về hệ thống khi chúng ta lập lại, với ít rủi ro hơn

Nếu bạn may mắn có nhiều thời gian hơn vào cuối quá trình increment (sau khi hoàn thành use case xanh và tất cả các use case màu vàng ) bạn nên:

- Xem lại tiến trình của dự án
- Tổng kết những kết hoạch cho increment tiếp theo (ví dụ là tái lập ưu tiên với những use case chưa thực hiện )
- Làm một số công việc không liên quan
- Mở tiệc công ty

#### **Ví dụ: Sự ưu tiên use case trong iCoot**

- Xanh
  - U1: Browse Index
  - U4 : Search

- U2: View results
- U3 : View CarModel Details
- U5: Log on
- Vàng
  - U12 :Log off
  - U6 :View member details
  - U7: Make reservation
  - U10 :View reservation
- Đỏ
  - U11: Cancel reservation
  - U8: View rentals
  - U9: Change Password

U1 thiết yếu và đơn giản (vì nó không bao gồm thuê mượn hoặc đặt chỗ trước) vì vậy nó thường đứng đầu danh sách. U5:log on là thứ thiết yếu trước khi dịch vụ thành viên được thực hiện vì vậy nó phải xuất hiện trước các dịch vụ thành viên. U6 được thử trước U7 bởi vì nó đơn giản hơn (đặt chỗ trước có vòng đời phức tạp) và cứ như vậy...

Chia thứ tự sự ưu tiên và sự khẩn cấp cho use case hệ thống là một chỉ dẫn khác mà chúng ta nên phát triển để phục vụ cho việc mở rộng và tái sử dụng . Dưới đây là việc tóm tắt về việc những cái đèn giao thông kết hợp với các giai đoạn khác của phát triển như thế nào - sau mẫu thử thương mại:

- Xanh : yêu cầu hệ thống, phân tích, thiết kế hệ thống, thiết kế tiểu hệ thống, thông số kĩ thuật thực hiện và kiểm tra nên được hoàn thành trong use case nhóm này.
- Vàng: yêu cầu hệ thống nên hoàn thành và phân tích và thiết kế hệ thống nên hoàn thành hoặc gần hoàn thành cho use case nhóm này; thiết kế tiểu hệ thống, thông số kĩ thuật, thực hiện và kiểm tra là lựa chọn thứ yếu.
- Đỏ: yêu cầu hệ thống nên được hoàn thành cho use case nhóm này , phân tích là lựa chọn thứ yếu; thiết kế hệ thống nên hỗ trợ use case này, thiết kế tiểu hệ thống , thông số kĩ thuật, thực hiện và kiểm tra không nên tiến hành .

### 3.7 Kết chương

Trong chương này, chúng ta cần nắm được:

- Chỉ rõ tầm quan trọng các chức năng của yêu cầu
- Mô hình giao dịch và hệ thống sử dụng giao dịch , các use case và nhận biết actor
- Mô hình thủ tục hệ thống với mô hình use case hoàn chỉnh bao gồm các use case, biểu đồ use case, thủ tục bổ sung , giao diện người sử dụng, use case ưu tiên ... Mặc dù biểu đồ giao tiếp và biểu đồ hoạt động là tùy ý ở xem xét tùy chọn ở giai đoạn này, luôn luôn là một bảng chú giải thuật ngữ scessential.

### **Tài liệu tham khảo**

## CHƯƠNG 4. PHÂN TÍCH YÊU CẦU PHẦN MỀM

### 4.1 Giới thiệu

Phân tích nhằm chỉ ra những việc mà hệ thống sẽ xử lý chứ không phải mô tả cách thức hệ thống xử lý. Ta cần phân rã các yêu cầu phức tạp thành các thành phần cơ bản và mối quan hệ giữa chúng để làm cơ sở cho các giải pháp đề xuất sau này. Phân tích là cách hiệu quả nhất để nhóm phát triển hệ thống có thể hiểu rõ được hệ thống thông qua việc mô hình hóa thế giới thực thành các đối tượng.

Một mô hình phân tích có đủ hai khía cạnh tĩnh và động của hệ thống. Mô hình tĩnh được thể hiện thông qua sơ đồ lớp, nhằm chỉ ra các đối tượng mà hệ thống sẽ xử lý và mối quan hệ giữa chúng. Mô hình động được dùng để chứng minh rằng mô hình tĩnh đã đề xuất là khả thi và thường được thể hiện qua các sơ đồ tương tác.

Đầu vào của quá trình phân tích gồm có mô hình các yêu cầu nghiệp vụ và mô hình các yêu cầu hệ thống. Mô hình yêu cầu nghiệp vụ mô tả các dòng công việc nghiệp vụ tự động hóa hoặc được làm bằng tay, thường được mô tả qua các đối tượng, thuật ngữ, sơ đồ use case, sơ đồ hoạt động nghiệp vụ. Mô hình yêu cầu hệ thống cho thấy cách nhìn hệ thống từ bên ngoài, được mô tả thông qua các sơ đồ use case hệ thống, các bản phác thảo giao diện người dùng và các yêu cầu phi chức năng. Cả hai mô hình này sẽ được biến đổi thành một mô hình mới chứa các đối tượng – bao gồm cả các thuộc tính và mối quan hệ giữa chúng – sẽ được xử lý bởi hệ thống đề xuất.

Về lý thuyết, ta có thể bỏ qua giai đoạn phân tích để đi thẳng vào thiết kế và cài đặt giải pháp, điều chỉnh giải pháp thông qua phương pháp thử-sai. Tuy nhiên, dễ thấy là cách này không mang lại hiệu quả. Ta không thể hiểu hết mọi vấn đề từ mô hình yêu cầu nghiệp vụ vì chúng mô tả những hoạt động hiện có trong khi hệ thống phần mềm đang xây dựng cần xử lý được cả những hoạt động mới. Thậm chí khi ta đã có một mô hình use case hệ thống thì những hiểu biết đó vẫn chưa phải là đầy đủ. Bởi vì mô hình này chỉ làm việc với mối tương tác giữa người dùng (actor) với biên của hệ thống còn nội tại bên trong hệ thống vẫn là một hộp đen.

Do đó, phân tích yêu cầu là một công việc cần thiết và phải được thực hiện trước nhằm ngăn chặn việc thiết kế giải pháp trước khi hiểu rõ vấn đề. Kết quả của quá trình phân tích yêu cầu chính là các đặc tả đặc trưng hoạt động của phần mềm, chỉ ra giao diện của các thành phần của hệ thống và xây dựng các yêu cầu bắt buộc hệ thống phải tuân thủ. Phân tích yêu cầu phần mềm cho phép bạn - với vai trò kỹ sư phần mềm, nhà phân tích, hoặc người mô hình hóa - có thể phát triển những yêu cầu cơ bản thông qua tất cả các quá trình

từ tiếp nhận yêu cầu, phân tích xây dựng, đàm phán các chức năng trong phần thu thập yêu cầu.

### ***Một số nguyên tắc khi phân tích***

- Mô hình cần phải tập trung vào các yêu cầu có thể nhìn thấy được trong vấn đề cần phân tích hoặc phạm vi của doanh nghiệp. Mức độ trừu tượng hóa phải khá cao. Đừng quá sa đà vào tiểu tiết vì như vậy chính là cố gắng giải thích hệ thống làm việc như thế nào.
- Mỗi thành phần của mô hình yêu cầu phải đóng góp vào sự hiểu biết tổng thể về các yêu cầu phần mềm và cung cấp một cách nhìn sâu sắc về phạm vi của thông tin, chức năng và các hoạt động của hệ thống.
- Không xem xét cơ sở hạ tầng và các mô hình phi chức năng khác cho đến khi đi vào giai đoạn thiết kế. Ở đây, có thể cần đến cơ sở dữ liệu nhưng các lớp cần thiết để cài đặt nó, các chức năng yêu cầu để truy xuất nó và các thao tác cần có để sử dụng nó chỉ được xem xét đến sau khi hoàn tất việc phân tích phạm vi của vấn đề.
- Tối thiểu hóa sự phụ thuộc bên trong hệ thống. Việc biểu diễn mối quan hệ giữa các lớp và các chức năng là quan trọng. Tuy nhiên, nếu mức độ liên kết ở mức quá cao sẽ tạo nên sự phụ thuộc và cần phải nỗ lực để giảm bớt những sự phụ thuộc này trong hệ thống.
- Cần chắc chắn rằng mô hình yêu cầu có giá trị cho tất cả các bên liên quan. Mỗi bên sử dụng mô hình theo một cách khác nhau. Chẳng hạn, với các bên liên quan tới nghiệp vụ, mô hình được dùng để đánh giá yêu cầu. Nhóm thiết kế thì dùng mô hình làm cơ sở cho bản thiết kế. Nhân viên kiểm soát chất lượng sử dụng mô hình để lập kế hoạch kiểm thử.
- Giữ cho mô hình ở mức đơn giản nhất có thể. Đừng tạo thêm sơ đồ khi chúng không bổ sung thông tin gì mới. Không sử dụng những ký hiệu phức tạp trong khi những ký hiệu đơn giản đã đủ để giải quyết vấn đề.

## **4.2 Quá trình phân tích yêu cầu phần mềm**

Quá trình phân tích yêu cầu gồm có nhiều bước và có thể lặp lại cho đến khi cả nhóm phát triển hệ thống và khách hàng cảm thấy thỏa mãn.

1. Sử dụng mô hình yêu cầu hệ thống để tìm ra các lớp ứng viên, dùng để mô tả các đối tượng có liên quan tới hệ thống và đưa chúng vào một sơ đồ lớp.

2. Tìm ra mối quan hệ giữa các lớp, bao gồm các mối quan hệ sau: kết hợp (association), kết tập (aggregation), hợp thành (composition) và kế thừa (inheritance).
3. Tìm các thuộc tính (chỉ đơn giản là tìm tên thuộc tính của các đối tượng) cho các lớp.
4. Xem xét từng use case hệ thống để kiểm tra chúng có sử dụng tới các đối tượng đã tìm thấy, điều chỉnh các lớp, các thuộc tính và mối quan hệ giữa chúng.
5. Hiện thực hóa các use case bằng cách dùng các sơ đồ động. Việc hiện thực hóa các use case như vậy sẽ sản sinh thêm các thao tác để bổ sung cho các thuộc tính.
6. Cập nhật các thuật ngữ và các yêu cầu phi chức năng nếu cần. Tuy nhiên, không nên thay đổi các use case.

Các thao tác phát hiện được trong quá trình hiện thực hóa các use case sẽ được xem xét trong giai đoạn thiết kế. Ở giai đoạn phân tích, ta chỉ cố gắng dựng lên một giải pháp khả thi chứ không phải thiết kế giải pháp hoàn chỉnh.

Sơ đồ lớp tương đối dễ hiểu đối với những người không chuyên, không phải là lập trình viên. Do vậy, một khi đã có sơ đồ lớp với đầy đủ các thuộc tính, nhóm phát triển nên trình bày nó cho khách hàng để họ có thể giúp tìm ra lỗi trong quá trình phân tích. Bởi vì khách hàng mới là người hiểu rõ nghiệp vụ của họ nhất. Việc trình bày sơ đồ lớp cho khách hàng phải được thực hiện ít nhất hai lần: một lần để khách hàng tìm ra lỗi hoặc sai sót và một lần để họ xác nhận nhóm phát triển đã sửa các lỗi sai đó.

### 4.3 Xây dựng sơ đồ lớp

Có nhiều phương pháp khác nhau để xác định lớp. Tuy nhiên, khó có thể tìm được phương thức chung để xác định các lớp của một hệ thống. Đây là một công việc đòi hỏi phải trải qua nhiều vòng lặp, có sự thông nhất với chuyên gia trong lĩnh vực ứng dụng nghiệp vụ. Vì quá trình xác định lớp trong giai đoạn này là một quá trình lặp lại, kết quả bước sau có thể làm thay đổi kết quả đã có từ bước trước nên các lớp tìm thấy tại mỗi bước thường được gọi là các lớp ứng viên. Phần này sẽ đề cập đến phương pháp tìm các lớp ứng viên dựa trên tiếp cận theo cụm danh từ - được đề xuất bởi Rebecca Wirfs-Brock, Brian Wilkerson và Lauren Wiener.

#### 4.3.1 Tìm các lớp ứng viên

Các lớp ứng viên thường được xác định bởi các danh từ trong văn bản mô tả các use case hoặc các mô tả yêu cầu. Những danh từ này thường biểu diễn:

- Bản thân hệ thống: chẳng hạn như “e-Learning”, “System”, ... Thuật ngữ “hệ thống” chỉ là thuật ngữ cho biết phạm vi của những công việc mà ta đang thực hiện.
- Actor: chẳng hạn như “Assistant” (trợ lý), “HeadOffice” (trụ sở), ...
- Phạm vi: chẳng hạn như “Customer applet” (phần ứng dụng quản lý khách hàng), “Head office link” (mỗi liên kết giữa các trụ sở), ... Trong giai đoạn này, ta chỉ cố gắng tìm ra những đối tượng liên quan tới nghiệp vụ cùng với các thông tin và hoạt động thật sự cần thiết liên quan tới nó. Phạm vi là những thành phần phần mềm cụ thể cho phép các tác nhân lấy được các đối tượng.
- Các kiểu dữ liệu thông thường như string, number, ... với giả định rằng chúng được cung cấp bởi các ngôn ngữ lập trình và các thư viện hỗ trợ.

Sau khi xác định được các lớp ứng viên, bạn cần viết một đoạn mô tả ngắn cho mỗi lớp. Nếu không thể mô tả một cách ngắn gọn ý nghĩa của một lớp nào đó, có nghĩa là lớp đó thực hiện quá nhiều chức năng hoặc biểu diễn quá nhiều thông tin, do vậy, bạn cần tách nó ra thành nhiều lớp khác.

#### 4.3.2 Nhận biết mối quan hệ giữa các lớp

Khi đã có được các lớp ứng viên, bạn cần vẽ ra mối quan hệ giữa chúng. Có bốn kiểu quan hệ được dùng, bao gồm:

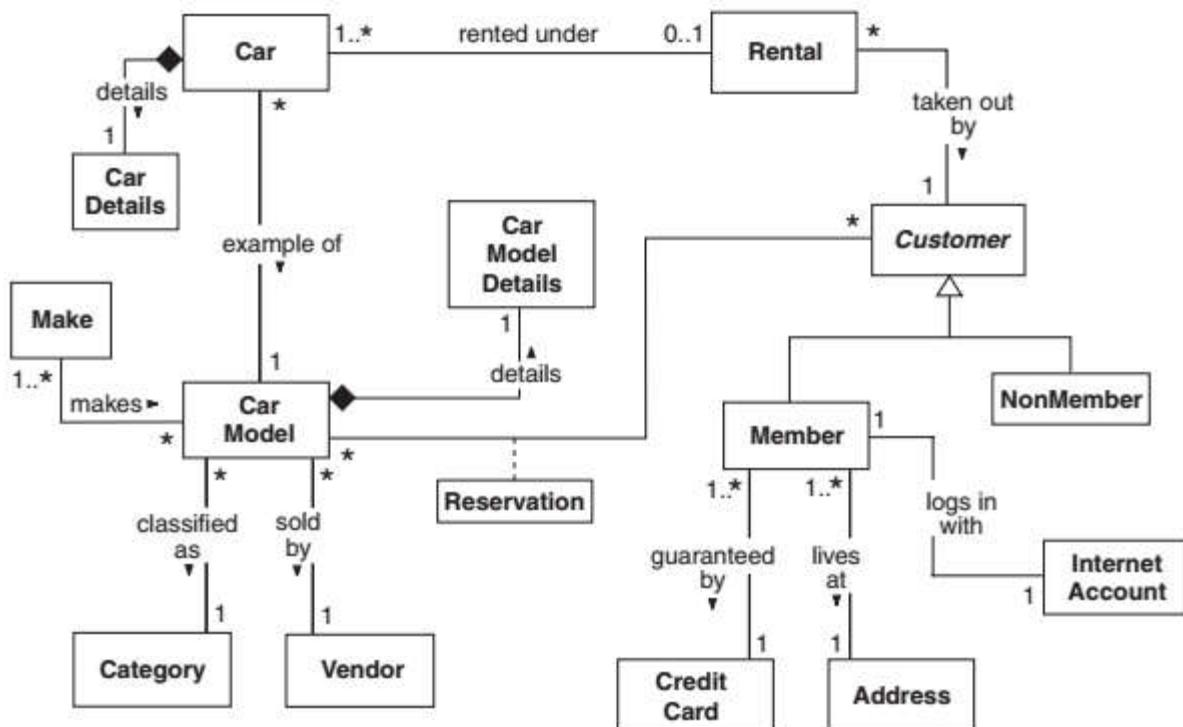
- Kế thừa (inheritance): một lớp con thừa hưởng lại những thuộc tính và hành vi đã có của lớp cha.
- Kết hợp (association): các đối tượng của một lớp được dùng kết hợp với các đối tượng của một lớp khác.
- Tụ hợp (aggregation): là một dạng của quan hệ kết hợp nhưng có ý nghĩa mạnh hơn. Nó được dùng cho trường hợp một thể hiện của một lớp được tạo thành từ nhiều thể hiện của một lớp khác. Ví dụ, một gia đình có nhiều thành viên, một lớp có nhiều sinh viên,...
- Hợp thành (composition): là một dạng đặc biệt của quan hệ tụ hợp với ý nghĩa mạnh hơn nhiều. Đối tượng hợp thành không thể tồn tại nếu thiếu các đối tượng thành phần. Hay nói khác hơn, mỗi đối tượng thành phần là một bộ phận cấu tạo nên đối tượng hợp thành và thể hiện vật lý của nó nằm trong đối tượng hợp thành. Chẳng hạn, đối tượng xe ô-tô được tạo nên bởi động cơ, cửa, đèn, bánh xe, ... Việc tạo thành một chiếc xe là việc lắp ráp các thành phần này với nhau. Một chiếc xe không thể không có khung xe.

Trong những mối quan hệ trên thì kế thừa là một dạng quan hệ khác hoàn toàn với ba dạng còn lại. Kế thừa mô tả một quan hệ giữa các lớp tại thời điểm biên dịch trong khi các dạng quan hệ còn lại mô tả mối quan hệ giữa các lớp tại thời điểm thực thi. Theo chuẩn UML, tất cả các mối quan hệ tại thời điểm thực thi được gắn cùng một thuật ngữ chung là mối kết hợp. Tuy nhiên, hầu hết mọi người đều sử dụng thuật ngữ mối kết hợp với ý nghĩa “mối kết hợp không phải là quan hệ tự hợp hay hợp thành”. Trong giai đoạn phân tích, tần suất xuất hiện các dạng quan hệ giảm dần như sau:

Association > Aggregation > Inheritance > Composition

### 4.3.3 Vẽ sơ đồ lớp

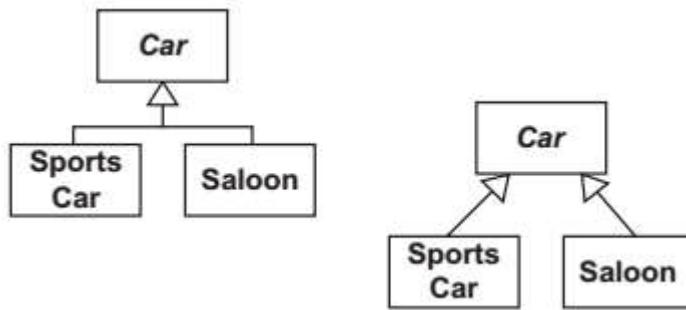
Trong sơ đồ lớp, mỗi lớp được biểu diễn bởi một hình chữ nhật, bên trong ghi tên lớp ở dạng in đậm. Nếu là lớp trừu tượng, tên được viết in nghiêng hoặc được đánh dấu bằng từ khóa {abstract} phía trên hoặc bên trái tên lớp thay vì chữ in nghiêng. Mọi quan hệ giữa các lớp được thể hiện bằng các đoạn thẳng kèm theo chú thích.



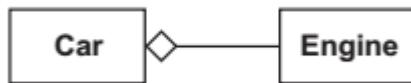
Hình 4.1 Sơ đồ lớp cho hệ thống iCoot

Theo đó, thậm chí một người không có kiến thức nhiều về UML cũng có thể dễ dàng nhận biết được các thông tin quan trọng từ sơ đồ lớp dựa vào các đoạn văn bản mô tả. Ví dụ, từ sơ đồ lớp hình 4.1, ta có thể thấy “Xe (Car) được thuê theo lượt (Rental)”, “Một lượt thuê (Rental) được thực hiện bởi một khách hàng (Customer)”, ...

Mặc dù các mối quan hệ trong sơ đồ lớp được vẽ giữa các lớp nhưng trên thực tế, tại thời điểm thực thi thì nó chính là mối quan hệ giữa các đối tượng (hay các thể hiện) của các lớp đó. Chẳng hạn, theo sơ đồ trên, tại thời điểm thực thi, một thể hiện của lớp Car sẽ kết nối tới (hay có quan hệ kết hợp với) một thể hiện của lớp Rental.



Hình 4.2 Thể hiện quan hệ kế thừa trong UML



Hình 4.3 Thể hiện quan hệ tụ hợp trong UML



Hình 4.4 Thể hiện quan hệ hợp thành trong UML



Hình 4.5 Thể hiện quan hệ kết hợp trong UML

Trong sơ đồ lớp, quan hệ kế thừa được thể hiện bằng một đường mũi tên liền nét, đầu mũi tên hình tam giác rỗng, hướng từ lớp con tới lớp cha như hình 4.2. Để nhấn mạnh tính phân cấp của các lớp con, các mũi tên có thể được kết hợp lại với nhau như hình bên trái.

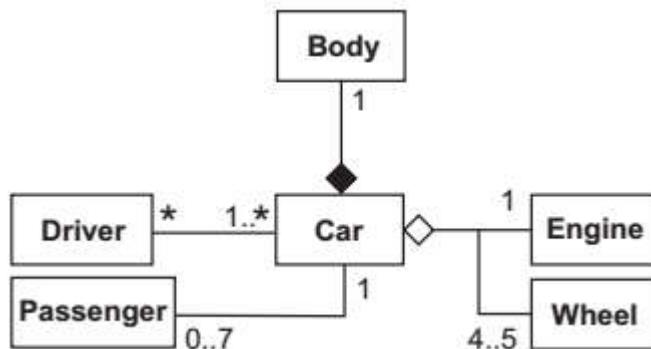
Quan hệ tụ hợp được chỉ ra bởi một đoạn thẳng nối giữa hai lớp với một đầu hình thoi rỗng ở phía lớp tổng thể như hình 4.3. Ta đọc mỗi quan hệ trong hình này như sau: “Động cơ (Engine) là một bộ phận của xe (Car)”. Quan hệ hợp thành được biểu diễn tương tự nhưng với hình thoi đặc, nằm ở phía lớp tổng thể như được chỉ ra trong hình 4.4. Hình này nói lên rằng: “Khung xe (Body) luôn luôn là một bộ phận của chỉ một chiếc xe (Car)”. Cuối cùng, mối kết hợp được thể hiện bằng một đoạn thẳng liền nét nối giữa hai lớp như hình 4.5. Hình này cho biết: “Một tài xế có quan hệ kết hợp với (lá) một chiếc xe” nhưng “Tài

xế không phải là một thành phần hay bộ phận không thể tách rời của chiếc xe” và “Tài xế cũng không phải luôn luôn là một bộ phận hay lái một chiếc xe duy nhất”.

Trừ quan hệ kế thừa, các mối quan hệ còn lại thường đi kèm với bản số. Bản số là một hoặc một khoảng giá trị cho phép một đối tượng của một lớp có thể tham gia bao nhiêu lần vào mỗi kết hợp với các đối tượng của lớp khác. Bản số có thể ở các dạng sau:

- n: chính xác n đối tượng tham gia vào mối quan hệ.
- m..n: một số bất kỳ nằm trong đoạn [m..n].
- p..\*: có ít nhất là p đối tượng tham gia vào mối quan hệ.
- \*: dạng rút gọn của 0..\*
- 0..1: tùy chọn, có thể có hoặc không có đối tượng tham gia.

Với mối quan hệ hợp thành, bản số ở phía lớp tổng thể luôn là một, vì vậy, không cần phải ghi bản số. Trong các trường hợp còn lại, nếu không có bản số, ta phải giả định rằng nó chưa được chỉ ra hoặc đơn giản là chưa biết, chưa xác định được ở thời điểm phân tích. Sẽ là sai lầm nếu cho rằng giá trị của bản số bị thiếu là 1.



Hình 4.6 Thể hiện bản số trong UML

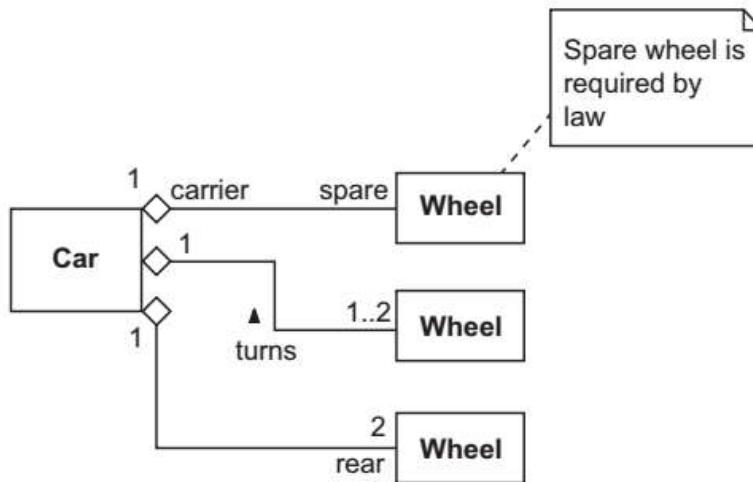
Ví dụ, từ hình 4.6, ta có thể rút ra được những thông tin sau:

- Một chiếc xe có một động cơ.
- Một động cơ là một bộ phận của một chiếc xe.
- Một chiếc xe có 4 hoặc 5 bánh xe.
- Mỗi bánh xe là một bộ phận của một chiếc xe.
- Một chiếc xe luôn luôn được tạo thành từ một khung xe.
- Một khung xe luôn là một bộ phận của một chiếc xe và nó hỏng khi xe hỏng.

- Một chiếc xe có thể lái bởi nhiều tài xế.
- Một tài xế có thể lái ít nhất một chiếc xe.
- Một chiếc xe chỉ có thể chở tối đa 7 hành khách tại mỗi thời điểm.
- Một hành khách chỉ có thể ở trong một chiếc xe tại mỗi thời điểm.

Sự khác biệt giữa quan hệ tụ hợp và hợp thành cũng rất khó nhận thấy. Trong hình 4.6, tại sao động cơ (Engine) lại có quan hệ tụ hợp trong khi khung xe (Body) lại có quan hệ hợp thành? Điểm khác biệt phụ thuộc vào việc chia sẻ đối tượng và vòng đời của đối tượng. Cần nhớ rằng, một đối tượng hợp thành (composed object) không bao giờ là một thành phần của nhiều đối tượng tổng thể và nó sẽ mất đi khi đối tượng chứa nó (đối tượng tổng thể) mất đi. Ngược lại, một đối tượng thành phần (aggregated object) có thể được chia sẻ và vẫn sống khi đối tượng tổng thể mất đi. Mặc dù khi xuất xưởng, mỗi xe có một động cơ mới hoàn toàn nhưng sau đó, động cơ có thể bị thay thế do bị hư hỏng. Vì thế, động cơ không nhất thiết phải chết theo xe. Ngược lại, khung (thân) xe là một thành phần không thể tách rời của xe, nó là linh hồn của chiếc xe. Nói một cách khác, ta không thể phá hủy xe mà không hủy đi phần khung xe nhưng lại có thể lấy động cơ ra trước rồi phá hủy.

Ngoài bùn số, các dạng quan hệ kết hợp có thể đi kèm với một nhãn để chỉ ra tên hay bản chất của mối kết hợp. Nếu cần thể hiện rõ ràng cách đọc tên của mối kết hợp thì ta có thể sử dụng một đầu mũi tên đặc, màu đen như hình 4.7.



Hình 4.7 Nhãn và vai trò của mối kết hợp trong UML

Tương tự như tên mối kết hợp, ta cũng có thể chỉ ra vai trò của đối tượng trong mối kết hợp bằng cách ghi tên vai trò nằm gần với đối tượng cần mô tả. Ví dụ, hình 4.7 chỉ ra các vai trò sau:

- Một chiếc xe có một bánh xe để dự phòng.
- Một bánh xe dự phòng được mang (chở) bởi một chiếc xe.
- Một chiếc xe có 2 bánh ở phía sau.

Hình 4.7 cũng cho thấy một mẫu chú thích trong UML. Chú thích là một đoạn văn bản ngắn được đóng khung - dạng như một mảnh giấy – và được nối tới phần muốn chú thích trong sơ đồ bởi một đoạn thẳng nét đứt. Chú thích có thể được thêm vào bất kỳ loại sơ đồ nào để bổ sung thêm thông tin nhằm làm rõ nghĩa của các thành phần mà các ký hiệu UML khác chưa thể hiện được.

#### 4.3.4 Bổ sung thuộc tính

Một thuộc tính là một đặc điểm của đối tượng, chẳng hạn như kích thước, tên, giá của một mặt hàng, lãi suất, thời hạn của một tài khoản vay, ... Mỗi thuộc tính thường đi kèm với kiểu dữ liệu, có thể là các kiểu dữ liệu cơ bản như số, chuỗi, ... hoặc một lớp. Kiểu dữ liệu được thêm vào sau tên thuộc tính và phân cách nhau bởi dấu hai chấm. Tuy nhiên, không nên lạm dụng việc đưa kiểu dữ liệu của thuộc tính vào sơ đồ lớp ở giai đoạn phân tích. Bởi vì ở giai đoạn này, ta không muốn phụ thuộc vào một ngôn ngữ lập trình hay tập thư viện hàm nào cụ thể.

Engine
capacity
horsePower
manufacturer:String
numberOfCylinders
fuelInjection:boolean

Hình 4.8 Thể hiện các thuộc tính trong UML

Trong sơ đồ lớp, các thuộc tính của một lớp được đặt trong một hình chữ nhật nằm ngay bên dưới tên lớp đó. Nếu cần thiết, có thể tạo một tài liệu để mô tả chi tiết về các thuộc tính. Hình 4.8 cho thấy các thuộc tính của một động cơ (Engine), bao gồm: dung tích (capacity), công suất (tính theo mã lực – horsePower), hãng sản xuất (manufacturer), số lượng xi-lanh (numverOfCylinders), phun nhiên liệu (fuelInjection). Sơ đồ này chỉ ra rằng, kiểu dữ liệu của hãng sản xuất là **String**, của fuelInjection là **Boolean**.

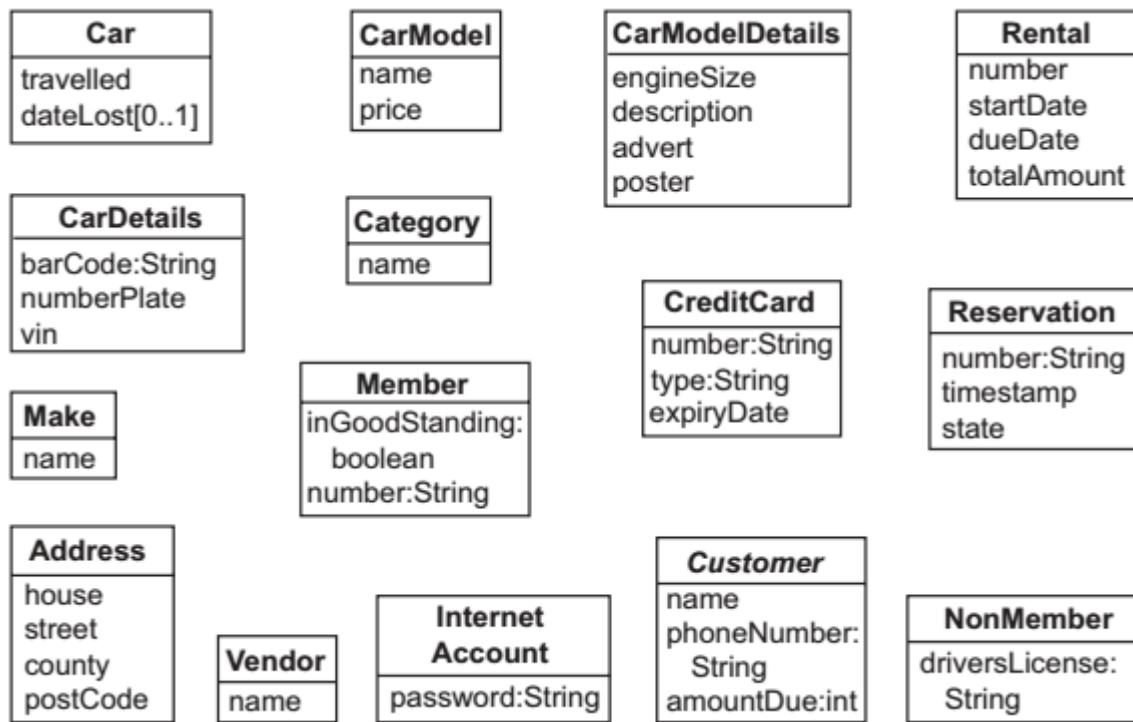
Việc sử dụng kiểu dữ liệu đi kèm tên thuộc tính đồng nghĩa với việc mở ra một loạt vấn đề phức tạp, chẳng hạn: String là gì? Boolean là gì? Nếu kiểu dữ liệu là tên của một trong các lớp trong sơ đồ lớp thì không thành vấn đề. Nhưng nếu không, ta không muốn bị bó buộc vào một ngôn ngữ lập trình hoặc tập thư viện hàm cụ thể. Do vậy, tốt hơn hết là nên

sử dụng các kiểu dữ liệu cơ bản - có mặt trong hầu hết các ngôn ngữ lập trình - như int, float, Boolean, String. Ngoài ra, cũng nên tránh sử dụng kiểu dữ liệu mảng, mặc dù nó được hỗ trợ bởi hầu hết các ngôn ngữ lập trình. Thay vì vậy, ta nên dùng các kiểu tập hợp List hoặc Set một cách tường minh, như vậy sẽ giúp cho các lớp trông có vẻ rõ ràng hơn.

Để đơn giản, các thuộc tính dẫn xuất cũng không nên đưa vào sơ đồ lớp. Chẳng hạn, các thuộc tính của một hình tròn gồm có: bán kính, đường kính, chu vi và diện tích. Tuy nhiên, chỉ cần biết giá trị của một trong các thuộc tính trên thì có thể tính ra giá trị của các thuộc tính còn lại. Vì vậy, chỉ cần chọn lọc và biểu diễn một trong bốn thuộc tính đó để đưa vào sơ đồ lớp. Trong trường hợp này, bán kính có vẻ là lựa chọn tốt nhất vì xác suất nó được truy cập là lớn hơn so với 3 thuộc tính còn lại và các giá trị khác có thể tính toán được thông qua phép nhân (nhanh hơn nhiều so với phép chia).

Trong UML, các mối kết hợp (cả 3 dạng) đều được xem là thuộc tính của một lớp. Nói cách khác, mọi đặc điểm của đối tượng đều có thể được biểu diễn bởi một thuộc tính hoặc một mối kết hợp, trong đó, tên của thuộc tính chính là tên (hoặc vai trò) của mối kết hợp. Để chuyển từ mối kết hợp sang một thuộc tính, ta có thể thêm bùn số vào các thuộc tính, đặt nằm sau tên kiểu dữ liệu hoặc tên thuộc tính. Ký hiệu \* được dùng cho thuộc tính đa trị và ký hiệu [0..1] được dùng cho thuộc tính tùy chọn. Tuy vậy, cách này hiếm khi được dùng.

Hình 4.9 cho thấy tập thuộc tính đầy đủ của các đối tượng đã phân tích ở bước trước khi khảo sát các use case hệ thống của iCoot. Thuộc tính dateLost trong lớp Car là thuộc tính tùy chọn, được chỉ ra bởi bùn số [0..1]. Nếu chiếc xe bị mất, ta ghi nhận ngày mất, ngược lại, không cần lưu giá trị gì cả. Khi được cài đặt trong một ngôn ngữ lập trình, ta có thể dùng một con trỏ **null** để chỉ ra rằng xe chưa bị mất. Nếu thuộc tính tùy chọn có kiểu cơ bản – chẳng hạn như kiểu int – ta phải dành riêng một giá trị nào đó để làm giá trị đánh dấu xe chưa bị mất, ví dụ như 0 hoặc -1.



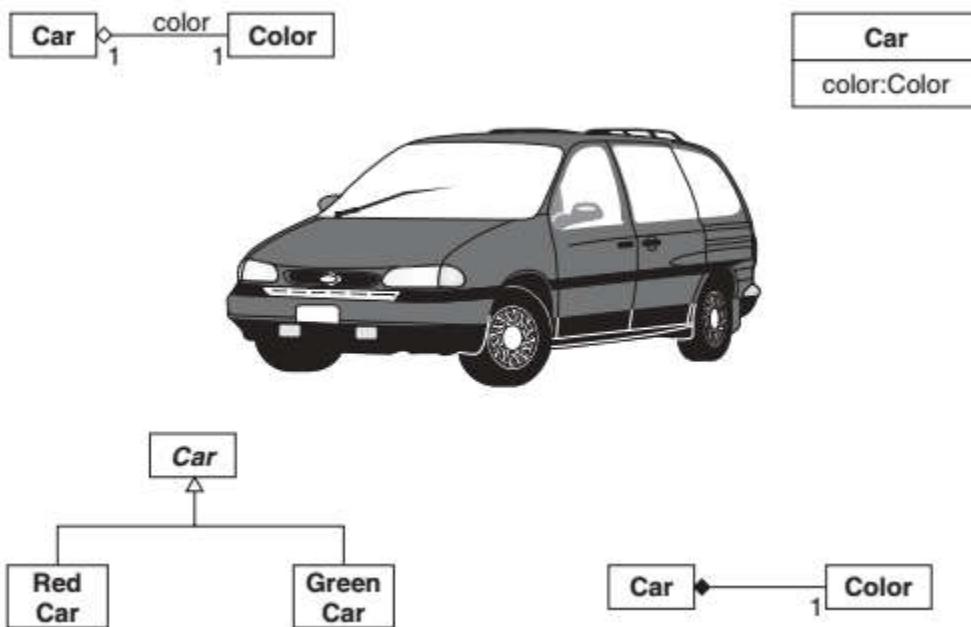
Hình 4.9 Các thuộc tính của các lớp trong hệ thống iCoot

Một vấn đề khá quan trọng khi bổ sung thuộc tính là lựa chọn giữa việc thêm một thuộc tính hay tạo ra một mối quan hệ. Chẳng hạn, làm thế nào để biểu diễn màu sắc của một chiếc xe để khách hàng dễ lựa chọn? Hình 4.10 cho thấy 4 cách khác nhau có thể dùng:

- Tạo thêm một lớp Color và bổ sung mối quan hệ tự hợp giữa Car và Color.
- Thêm một thuộc tính color có kiểu Color vào lớp Car.
- Tạo thêm các lớp con của lớp Car ứng với mỗi màu sắc.
- Tạo thêm mối quan hệ hợp thành giữa Car và Color.

Cả bốn cách đều có ý nghĩa, có thể dùng nhưng làm thế nào để chọn được cách phù hợp nhất? Vấn đề trọng tâm là lựa chọn nào phù hợp với tình huống, ngữ cảnh hiện tại nhất? Hay nói khác hơn là biểu diễn một cách tự nhiên nhất. Với tùy chọn thứ nhất, có vẻ hơi quá nếu nói rằng màu sắc là một bộ phận hay thành phần của chiếc xe. Lựa chọn thứ 2 có vẻ khá phù hợp: trong các mối quan tâm của khách hàng, màu sắc chỉ là một thuộc tính của chiếc xe. Lựa chọn thứ 3 là hơi dư thừa, liệu có nhất thiết phải tạo ra một loại xe mới cho mỗi màu sắc, đặc biệt là có cả loạt màu khác nhau? Tùy chọn thứ tư có vẻ hợp lý hơn một tí so với phương án 1. Trong trường hợp này, nó nói lên rằng, sau khi xuất xưởng, thậm chí nếu ta thay đổi màu sơn thì màu sơn ban đầu của xe vẫn sẽ còn nằm lại bên dưới lớp

sơn mới. Như vậy, nhìn chung, phương án 2 vẫn là thích hợp nhất trong ngữ cảnh phục vụ nhu cầu của người mua xe.



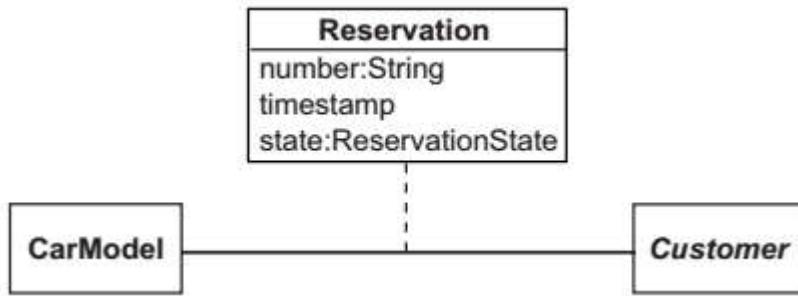
Hình 4.10 Lựa chọn giữa thuộc tính và mối quan hệ

Tuy nhiên, sẽ là một lựa chọn khác nếu ta đứng ở vai trò nhà sản xuất và cần mô hình hóa việc tạo ra chiếc xe. Khi đó, nhà sản xuất sơn và màu sơn có thể trở nên quan trọng hơn. Vì vậy, ta cần mô hình hóa màu sắc thành một lớp riêng biệt, có chứa thuộc tính và các mối quan hệ riêng của nó. Trong trường hợp này, phương án thứ 4 có vẻ là lựa chọn tốt nhất.

Nói chung, trong những tình huống như thế này, không có câu trả lời chính xác. Các nhà phân tích phải dựa vào kiến thức, kinh nghiệm và trực giác của mình để lựa chọn phương án phù hợp nhất với ngữ cảnh.

#### 4.3.5 Bổ sung các lớp kết hợp

Thỉnh thoảng, một mối kết hợp cũng có những thông tin hoặc thao tác riêng. Khi đó, một lớp kết hợp sẽ được tạo ra từ mối kết hợp đó. Thông thường, những lớp này sẽ được tạo ra từ các lớp có quan hệ nhiều – nhiều. Ví dụ, hình 4.11 chỉ ra rằng, một mẫu xe (CarModel) có thể được đặt hàng bởi nhiều khách hàng (Customer) và ngược lại, một khách hàng có thể đặt hàng nhiều mẫu xe khác nhau. Trong trường hợp này, sẽ có thêm một lớp Reservation với các thuộc tính số lượng, thời gian đặt hàng và trạng thái đơn hàng để mô tả chi tiết hơn về mối kết hợp.



Hình 4.11 Một lớp kết hợp từ hệ thống iCoot

Một lớp kết hợp biểu diễn các thuộc tính và phương thức chỉ tồn tại do có sự tồn tại của mỗi kết hợp. Các thuộc tính và phương thức đó không được đưa vào 2 lớp đầu cuối của mỗi kết hợp. Lớp kết hợp chỉ đặc biệt hữu dụng trong giai đoạn phân tích. Sang giai đoạn thiết kế, ta sẽ phải thay thế các lớp kết hợp bằng các lớp cụ thể hơn bởi vì chúng không được hỗ trợ trực tiếp bởi hầu hết các ngôn ngữ lập trình.

Ở ví dụ trên, khi khách hàng tạo ra một đặt chỗ, một kiên kết mới giữa hai lớp Customer và CarModel được tạo ra tại thời điểm thực thi. Trong quá trình thu thập và phân tích yêu cầu, tất cả các thông tin như mã số đặt chỗ, thời gian và trạng thái đều phải được ghi nhận. Tuy nhiên, việc đưa những thông tin này vào lớp Customer hay CarModel đều không phù hợp vì chúng nằm đâu đó ở giữa. Vì vậy, sử dụng lớp kết hợp là thích hợp nhất.

#### 4.4 Xây dựng sơ đồ cộng tác

Việc phân tích khía cạnh động của hệ thống được thực hiện vì những lý do sau đây:

- Để chắc chắn rằng sơ đồ lớp được thiết kế đầy đủ và chính xác. Từ đó, ta có thể khắc phục những sai sót bằng cách thêm, xóa hoặc sửa đổi các lớp, các mối quan hệ, các thuộc tính và các thao tác.
- Để có thể tin tưởng rằng, đến thời điểm này, mô hình có thể được cài đặt vào phần mềm. Không chỉ nhóm phát triển cần phải chắc chắn trước khi thiết kế mà điều quan trọng là khách hàng cũng cần phải xác nhận.
- Để xác thực rằng các chức năng của giao diện người dùng sẽ xuất hiện trong hệ thống cuối cùng.

Phần quan trọng nhất trong việc phân tích khía cạnh động của hệ thống chính là việc hiện thực hóa các use case. Nghĩa là, cần làm cho các use case trở thành hiện thực bằng cách chỉ ra cách chúng được thực thi khi các đối tượng cộng tác với nhau. Việc hiện thực hóa use case tuân thủ theo các bước sau:

- Với mỗi use case hệ thống, mô phỏng các thông điệp được gửi giữa các đối tượng và ghi nhận kết quả bằng các sơ đồ giao tiếp (sơ đồ cộng tác).
- Bổ sung các thao tác vào các đối tượng nhận thông điệp.
- Thêm các lớp biểu diễn biên (giao diện) của hệ thống và các lớp điều khiển (nơi chứa các xử lý nghiệp vụ phức tạp hoặc để tạo, nhận đối tượng mới) nếu thấy cần thiết.

Khi mô phỏng các thông điệp được gửi giữa các đối tượng phân tích, ta cần ghi lại kết quả. Trong UML, có hai loại sơ đồ được thiết kế cho mục đích này, bao gồm: sơ đồ giao tiếp (sơ đồ cộng tác) và sơ đồ tuần tự. Mặc dù cả hai loại sơ đồ này đều có thể ghi nhận cùng một thông tin nhưng sơ đồ giao tiếp phù hợp hơn cho việc hiện thực hóa use case. Lý do là nó đơn giản hơn và chú trọng vào các đối tượng, sự cộng tác động giữa các đối tượng thay vì thứ tự gửi thông điệp như trong sơ đồ tuần tự.

Một sơ đồ giao tiếp có thể cho thấy:

- Các tác nhân tương tác với biên hệ thống (giao diện người dùng).
- Biên của hệ thống tương tác với các đối tượng khác bên trong hệ thống.
- Các đối tượng bên trong hệ thống tương tác với biên của các hệ thống bên ngoài.

Thay vì sử dụng các tương tác 2 chiều, sơ đồ giao tiếp sử dụng kiểu tương tác hướng máy tính (computer-oriented), khách hàng-nhà cung cấp (client-supplier): tác nhân khởi tạo tương tác tới đối tượng biên, đối tượng biên khởi tạo tương tác với các đối tượng hệ thống, đối tượng hệ thống lại khởi tạo tương tác với các đối tượng khác trong hệ thống và biên của hệ thống khác.

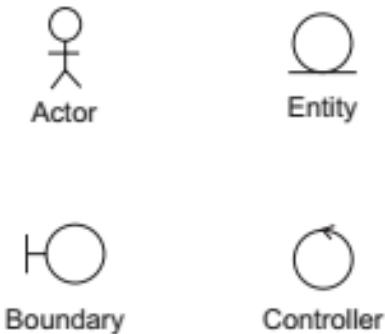
Trong sơ đồ này, ta không cần phải biểu diễn các đối tượng nghiệp vụ nằm bên ngoài hệ thống và cũng không cần đưa các tác nhân không có tương tác trực tiếp với hệ thống vào sơ đồ.

#### 4.4.1 Các đối tượng trong sơ đồ giao tiếp

Trong sơ đồ giao tiếp, các đối tượng được thể hiện bởi các biểu tượng như trong hình 4.12. Việc sử dụng các biểu tượng nhằm tăng thêm tính trực quan, biểu cảm và chỉ ra được bản chất của đối tượng.

- Tác nhân (actor): một người (thông thường) hoặc hệ thống (hiếm khi) nằm bên ngoài hệ thống.

- Biên (boundary): một đối tượng tại cạnh (edge) của hệ thống, nằm giữa hệ thống và tác nhân. Đối với các tác nhân là hệ thống, biên cung cấp một đường truyền thông. Đối với các tác nhân con người, biên có nghĩa là một giao diện người dùng (có thể là toàn bộ cửa sổ hoặc chỉ một phần), nhận lệnh hoặc truy vấn từ người dùng và hiển thị các phản hồi hoặc kết quả xử lý. Mỗi đối tượng biên thường tương ứng với một use case hoặc một nhóm các use case có liên quan với nhau.



Hình 4.12 Các biểu tượng trong sơ đồ giao tiếp

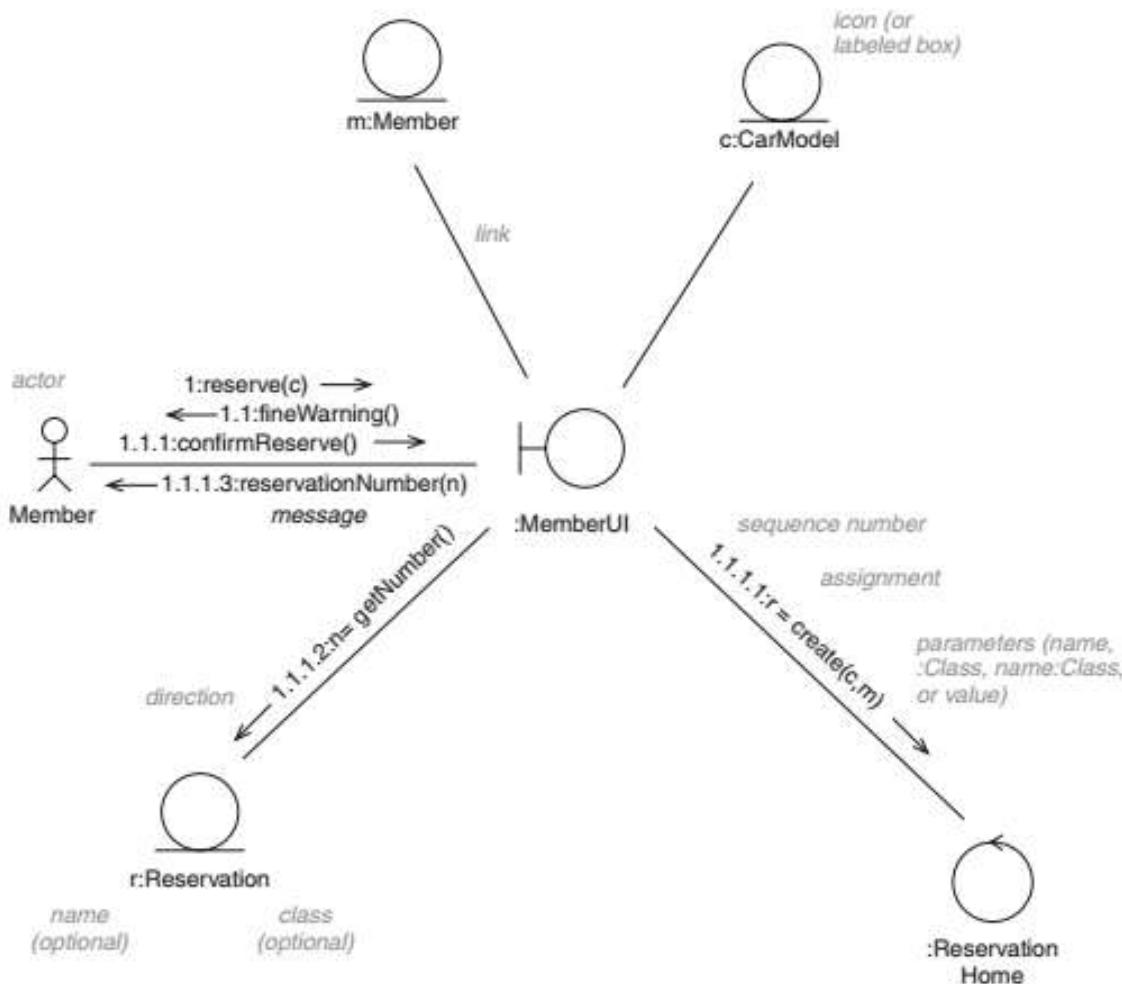
- Thực thể (entity): một đối tượng bên trong hệ thống, biểu diễn một khái niệm nghiệp vụ như khách hàng, xe, mẫu xe và chứa những thông tin hữu ích. Thông thường, các thực thể được sử dụng, xử lý bởi các đối tượng biên và đối tượng điều khiển. Các lớp thực thể là những lớp xuất hiện trong sơ đồ lớp mức phân tích. Hầu hết các thực thể đều tồn tại xuyên suốt quá trình thiết kế.
- Lớp điều khiển (controller): một đối tượng bên trong hệ thống thực hiện việc đóng gói một quy trình phức tạp hoặc lộn xộn. Một lớp điều khiển đóng vai trò cung cấp các dịch vụ sau: điều khiển một phần hoặc mọi xử lý của hệ thống, tạo các thực thể mới, lấy các thực thể đã có. Nếu không có lớp điều khiển, các thực thể có thể trở thành một khối lộn xộn. Các lớp điều khiển chỉ mang lại lợi ích trong giai đoạn phân tích, vì vậy, có thể chúng không tồn tại đến giai đoạn thiết kế ngoại trừ một lớp điều khiển đặc biệt gọi là **home**. Home là một lớp điều khiển được dùng để tạo ra các thực thể mới và lấy các thực thể hiện có. Một home cũng có thể có các thông điệp tiện ích, chẳng hạn `carModelHome.findEngineSizes()`.

#### 4.4.2 Các thành phần của sơ đồ giao tiếp

Hình 4.13 cho thấy một sơ đồ giao tiếp trong giai đoạn phân tích. Tên của các thành phần trong sơ đồ được đính kèm bên cạnh mỗi biểu tượng và có màu xám nhạt. Ý nghĩa của các thành phần như sau:

- Tác nhân: được thể hiện giống như trong sơ đồ use case.

- Các đối tượng được thể hiện bằng các biểu tượng hoặc các hộp có gán nhãn.
- Đoạn thẳng nối giữa hai đối tượng chỉ ra một liên kết, tượng tự như sơ đồ đối tượng.
- Một thông điệp gồm có số thứ tự (chỉ ra thứ tự của thông điệp trong quá trình giao tiếp), tên của thông điệp và danh sách thao só (đặt trong ngoặc đơn).
- Mũi tên với đầu mũi tên hở cho biết hướng của thông điệp đang được gửi đi. Mũi tên này có thể đặt ở sau hoặc phía dưới tên thông điệp.
- Các nhãn được dùng để nhận biết đối tượng và tham số. Chúng có thể được viết ở một trong các dạng sau: tên, tên:kiểu, :kiểu hoặc một giá trị cụ thể. Chẳng hạn: carModel (tên), m:CarModel (tên:kiểu), :CarModel (:kiểu), 10 hoặc “abc”.
- Phép gán một giá trị trả về cho một tên được thể hiện như sau: n = getNumber().
- Một thông điệp điều kiện được thể hiện bởi một lính canh (điều kiện được đặt trong ngoặc vuông), tiếp theo là thông điệp. Ví dụ: 4: [Only on Saturday] readPaper().
- Vòng lặp được chỉ ra bởi dấu \* bên cạnh số thứ tự.

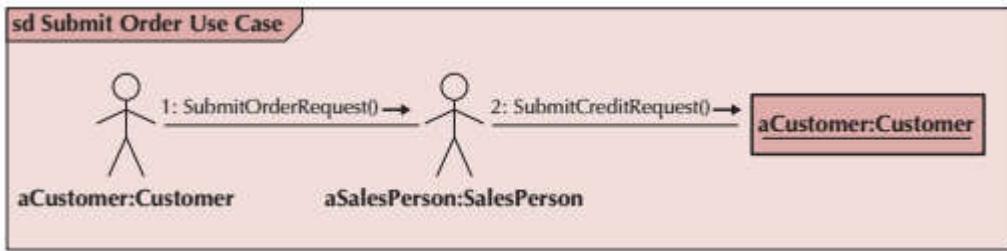


Hình 4.13 Các thành phần của một sơ đồ giao tiếp

Mặc dù trên thực tế, tác nhân không thực sự “gửi thông điệp” đến đối tượng biên, nhưng trên sơ đồ, các thông điệp hình thức là một cách thuật tiện để biểu diễn một tương tác.

Ban đầu, một thông điệp được gửi tới một đối tượng, làm cho một phương thức nào đó trong đối tượng được thực thi. Bên trong phương thức đó, các thông điệp khác lại có thể được gửi đi. Vì vậy, các số thứ tự trong thông điệp cho biết mức tương tác. Ví dụ, thông điệp 1 làm cho phương thức 1 thực thi. Thông điệp đầu tiên trong phương thức 1.1, thông điệp thứ 2 là 1.2, và cứ thế tiếp tục. Thông điệp đầu tiên trong phương thức 1.2 được gửi sẽ có số thứ tự là 1.2.1. Để thể hiện việc xử lý song song trên cùng một sơ đồ, ta có thể đặt tên cho mỗi giao tiếp. Tên này được xem là một phần của số thứ tự. Chẳng hạn, ta có thể sử dụng a và b để đặt tên cho hai tiêu trình riêng biệt. 2.2a và 2.2b có thể xảy ra tại cùng thời điểm trong khi 2.3a sẽ xảy ra sau, bên trong tiêu trình a.

Việc đọc sơ đồ giao tiếp tương đối đơn giản, bắt đầu từ tác nhân kích hoạt tương tác. Ví dụ, Hình 4.13 ở trên mô tả những thông tin sau: Một tác nhân Member yêu cầu đối tượng biên MemberUI thực hiện việc đặt hàng một mẫu xe CarModel cụ thể. Member nhận được cảnh báo rằng sẽ bị phạt nếu không nhận hàng khi xe được giao. Một khi Member xác nhận rằng họ thực sự có mong muốn đặt hàng, MemberUI sẽ yêu cầu đối tượng điều khiển ReservationHome tạo một đối tượng Reservation mới rồi gửi đến đối tượng CarModel và Member. Cuối cùng, MemberUI lấy mã số của đơn đặt hàng mới và hiển thị cho Member thấy.



Hình 4.14 Ví dụ về ngữ cảnh của sơ đồ giao tiếp

#### 4.4.3 Các bước tạo sơ đồ giao tiếp

Bước thứ nhất là xác định ngữ cảnh của sơ đồ giao tiếp. Ngữ cảnh ở đây có thể là một hệ thống, một use case hoặc một tình huống của use case. Ngữ cảnh của sơ đồ được thể hiện bằng một hình chữ nhật được gán nhãn bao quanh sơ đồ như trong hình 4.14.

Bước thứ hai là xác định các đối tượng, tác nhân và mối kết hợp liên kết giữa các đối tượng (và tác nhân) tham gia vào sự cộng tác động. Nhớ rằng, các đối tượng tham gia vào sự cộng tác là các thể hiện của các lớp đã được xác định trong khi xây dựng sơ đồ lớp.

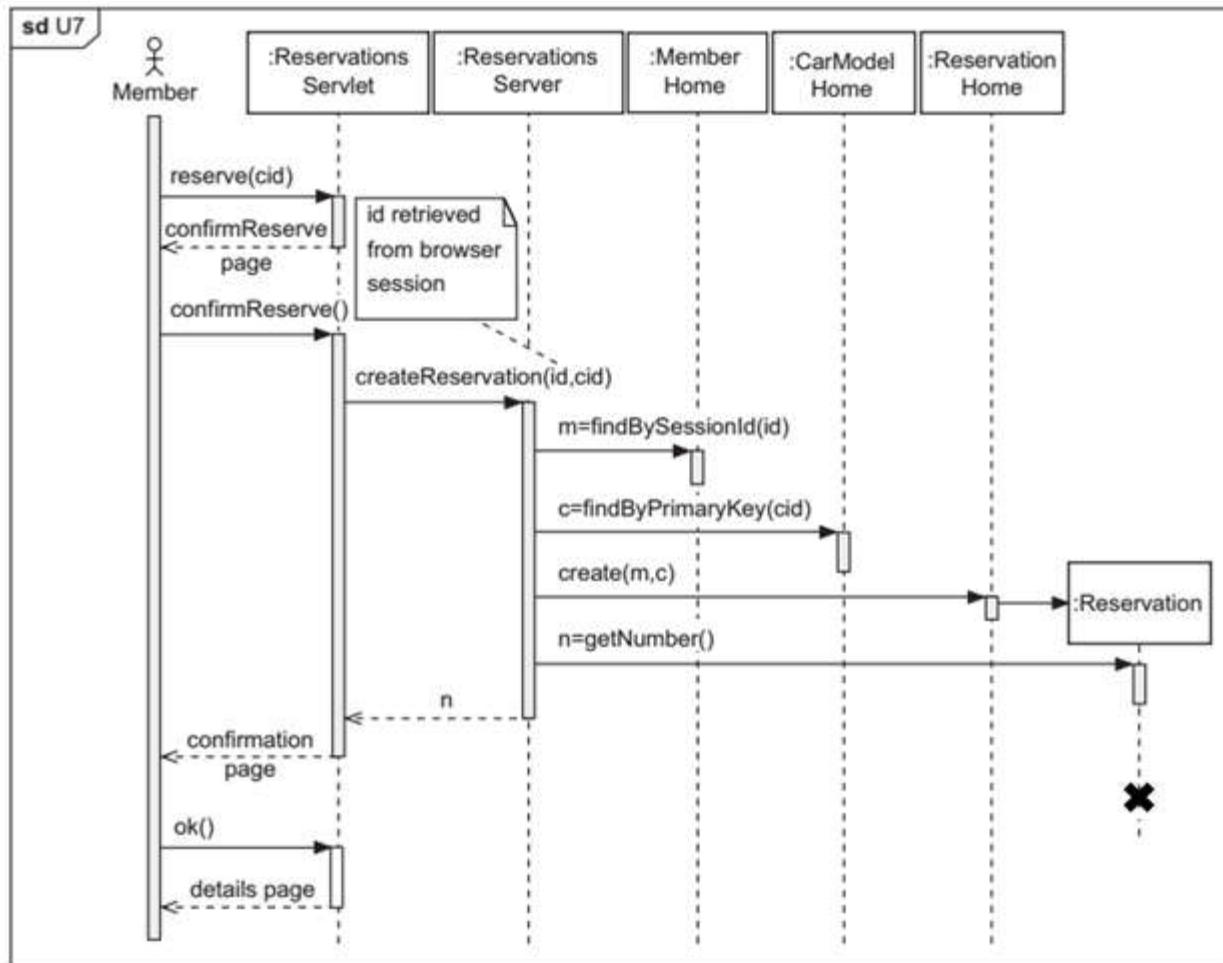
Thứ ba là bố trí các đối tượng (và tác nhân) cùng với mối kết hợp giữa chúng lên sơ đồ giao tiếp bằng cách đặt chúng gần nhau dựa trên mối kết hợp. Việc chú trọng vào mối kết hợp giữa các đối tượng (và tác nhân) và tối thiểu hóa số lượng nét vẽ thể hiện mối kết hợp sẽ làm tăng tính rõ ràng, dễ hiểu cho sơ đồ.

Bước thứ năm là bổ sung các thông điệp vào các mối kết hợp giữa các đối tượng. Việc này được thực hiện bằng cách thêm tên của thông điệp vào các đường vẽ mỗi kết hợp giữa các đối tượng và một mũi tên để chỉ hướng mà thông điệp được gửi đi. Mỗi thông điệp có một số thứ tự đi kèm để thể hiện trình tự thời gian gửi thông điệp.

Bước cuối cùng là đánh giá lại sơ đồ giao tiếp. Mục đích của bước này là bảo đảm rằng sơ đồ giao tiếp mô tả đầy đủ các xử lý bên dưới. Bước này được thực hiện bằng cách rà soát và đảm bảo tất cả các bước trước đều đã được thể hiện trên sơ đồ.

## 4.5 Xây dựng sơ đồ tuần tự

Sơ đồ tuần tự là một trong hai loại sơ đồ tương tác. Sơ đồ tuần tự minh họa các đối tượng tham gia vào một use case và các thông điệp được gửi giữa các đối tượng theo thời gian. Mỗi sơ đồ tuần tự được dùng cho *một* use case. Đây là một dạng mô hình động cho thấy rõ chuỗi thông điệp được gửi giữa các đối tượng trong một tương tác xác định. Vì các sơ đồ tuần tự nhấn mạnh thứ tự thời gian của hoạt động được thực hiện trong một tập các đối tượng nên chúng đặc biệt hữu ích trong việc hiểu rõ các đặc tả thời gian thực và các use case phức tạp.



Hình 4.15 Sơ đồ tuần tự cho use case đặt hàng (Make Reservation)

Sơ đồ tuần tự có thể được vẽ theo dạng tổng quát để cho thát tất cả các tình huống khả dĩ của một use case. Tuy nhiên, các nhà phân tích thường vẽ một tập các sơ đồ tuần tự cụ thể cho từng tình huống trong use case để mô tả chi tiết và hiểu rõ hơn dòng công việc của tình huống đó theo thứ tự thời gian. Dạng sơ đồ này sẽ được sử dụng xuyên suốt cả hai giai đoạn phân tích và thiết kế. Tuy nhiên, các sơ đồ ở mức thiết kế sẽ rất phụ thuộc vào cách

thúc cài đặt và thường sẽ có thêm các đối tượng sơ sở dữ liệu hoặc các thành phần giao diện người dùng cụ thể.

Hình 4.15 cho thấy một ví dụ về sơ đồ tuần tự. Sơ đồ này thể hiện các đối tượng và thông điệp trong use case đặt hàng (Make Reservation) một mẫu xe khi xem thông tin chi tiết mẫu xe đó. Trong sơ đồ này, các tác nhân và đối tượng tham gia gồm có: Member, ReservationsServlet, ReservationsServer, MemberHome, CarModelHome, ReservationHome.

#### 4.5.1 Các thành phần của một sơ đồ tuần tự

Các tác nhân (actor) và các đối tượng (object) tham gia vào sơ đồ tuần tự được đặt ở trên đầu của sơ đồ. Ký hiệu actor giống như trong sơ đồ use case và ký hiệu đối tượng giống như trong sơ đồ đối tượng. Với mỗi đối tượng, ta thường viết tên đối tượng sau là dấu hai chấm ":" và tên lớp hoặc chỉ có dấu ":" và tên lớp như trong hình 4.15.

Đường thẳng nét đứt chạy dọc bên dưới mỗi actor và đối tượng dùng để ký hiệu thời gian sống (đường sống - lifeline) của actor và đối tượng. Đôi khi, một đối tượng tạo ra một đối tượng tạm thời, trong trường hợp như vậy, ở cuối đường sống sẽ có thêm một dấu X để đánh dấu đối tượng đã bị hủy. Ví dụ, các ứng dụng Web thương mại điện tử thường sử dụng một đối tượng ShoppingCart (giỏ hàng) để lưu trữ tạm các mặt hàng được chọn mua. Một khi đơn hàng đã được xác nhận thì không cần đến giỏ hàng nữa. Vì vậy, đối tượng giỏ hàng cần phải được hủy để giải phóng tài nguyên. Trong trường hợp này, dấu X được đặt vào đường sống của đối tượng ShoppingCart tại thời điểm giỏ hàng bị hủy. Nếu các đối tượng vẫn tiếp tục tồn tại trong hệ thống sau khi chúng đã được sử dụng thì đường sống sẽ chạy cho đến cuối (bên dưới) sơ đồ.

Hình chữ nhật mỏng nằm chồng lên đường sống được gọi là đoạn thực thi (execution occurrence). Nó cho biết thời điểm các lớp gửi, nhận thông điệp và thời gian thực hiện. Một thông điệp là một sự giao tiếp giữa các đối tượng nhằm truyền tải thông tin với mong muốn một hoạt động sẽ xảy ra sau đó. Có nhiều loại thông điệp khác nhau có thể được dùng trong sơ đồ tuần tự. Tuy nhiên, trong trường hợp dùng sơ đồ tuần tự để mô hình hóa use case, chỉ có hai loại thông điệp thường được dùng: lời gọi hàm (operation call) và thông điệp trả về (return). Thông điệp *lời gọi hàm* giữa các đối tượng được thể hiện bằng một mũi lên liền nét, hướng từ đối tượng gửi thông điệp đến đối tượng cần gọi hàm. Giá trị của các đối số cho thông điệp được đặt trong cặp dấu ngoặc đơn nằm sau tên thông điệp. Thứ tự của thông điệp đi từ trên xuống dưới. Như vậy, các thông điệp nằm ở vị trí cao hơn (phía trên của sơ đồ) sẽ được thực hiện trước và ngược lại. Thông điệp trả về được thể hiện bằng một mũi tên nét đứt hướng đến đối tượng nhận kết quả trả về. Thông tin trả về được gán làm nhãn của mũi tên. Tuy nhiên, việc thêm các thông điệp trả về thường làm cho sơ đồ

trở nên lộn xộn và phức tạp. Vì thế, chỉ nên dùng trong trường hợp các thông điệp trả về bổ sung thêm nhiều thông tin vào sơ đồ hoặc thật sự cần thiết, nếu không thì không cần thể hiện thông điệp trả về trong sơ đồ tuần tự. Thay vào đó, ta sử dụng phép gán giá trị trả về cho một biến ngay trong thông điệp lời gọi hàm như hình 4.15.

Trong một số trường hợp, thông điệp chỉ được gửi khi phải thỏa mãn một điều kiện nào đó. Khi đó, điều kiện được đặt trong dấu ngoặc vuông nằm trước tên của thông điệp. Chẳng hạn, [aOrder is confirmed] DeleteShoppingCart(). Khi dùng sơ đồ tuần tự để mô hình một tình huống cụ thể, các điều kiện thường không được thể hiện trong một sơ đồ duy nhất mà chỉ có thể hiểu ngầm thông qua các sơ đồ tuần tự khác.

Cũng có lúc, thông điệp được lặp lại nhiều lần. Ký hiệu dấu \* được dùng trước tên thông điệp để thể hiện điều đó. Một đối tượng cũng có thể gửi thông điệp đến chính nó, gọi là tự ủy nhiệm (self-delegation). Nếu một đối tượng tạo ra một đối tượng khác, thông điệp sẽ được gửi trực tiếp đến đối tượng đó thay vì đến đường sống.

#### 4.5.2 Các nguyên tắc khi vẽ sơ đồ tuần tự

- Có gắng sắp xếp actor và các đối tượng từ trái sang phải và theo thứ tự mà chúng tham gia vào tình huống của use case. Đặt chúng ở trên cùng của sơ đồ. Theo đó, các thông điệp không chỉ chạy tuần tự từ trên xuống dưới mà còn từ trái qua phải. Cách này giúp đọc sơ đồ dễ hơn.
- Nếu có một actor và một đối tượng giống nhau về mặt khái niệm nhưng một khái niệm ở trong phần phần, khái niệm kia ở thế giới thực thì cũng đặt chúng cùng một tên. Điều này có thể xảy ra khi khái niệm đó tồn tại trong cả sơ đồ use case (vai trò actor) lẫn sơ đồ lớp (vai trò là một lớp). Thoạt nghe thì có vẻ hơi rối rắm nhưng nếu chúng cùng biểu diễn một ý niệm thì cứ đặt tên chúng giống nhau. Ví dụ, một tác nhân khách hàng (Customer) tương tác với hệ thống và hệ thống lưu trữ thông tin về khách hàng. Trong trường hợp này, chúng thực sự biểu diễn cùng một ý niệm.
- Đối tượng khởi đầu hoạt động (tình huống) – có thể là actor hoặc object – phải được đặt nằm bên trái nhất của sơ đồ.
- Khi có nhiều đối tượng cùng kiểu tham gia vào cùng một sơ đồ tuần tự, phải thêm tên của đối tượng vào trước dấu hai chấm và tên lớp.
- Chỉ thể hiện các giá trị trả về trên sơ đồ nếu chúng thật sự cần thiết hoặc không phải giá trị hiển nhiên. Việc biểu diễn tất cả các giá trị trả về sẽ làm cho sơ đồ tuần tự phức tạp hơn và có thể là khó hiểu. Trong nhiều trường hợp, càng gọn thì càng tốt. Vì thế, chỉ nên đưa thông điệp trả về vào sơ đồ khi nó có bổ sung thêm thông tin cho người đọc.

- Đặt tên của thông điệp và giá trị trả về nằm gần về phía bên trái của mũi tên. Điều này giúp người đọc dễ dàng hiểu và diễn dịch các thông điệp và giá trị trả về của chúng.

#### 4.5.3 Các bước để xây dựng sơ đồ tuần tự

Để tạo một sơ đồ tuần tự, thông thường, ta phải thực hiện sáu bước: xác định ngữ cảnh, nhận biết các tác nhân và các đối tượng tham gia, vẽ đường sống của mỗi đối tượng, thêm các thông điệp vào sơ đồ, đặt các đoạn thực thi trên đường sống của các đối tượng và cuối cùng là đánh giá lại sơ đồ tuần tự.

Bước đầu tiên trong quá trình này là xác định ngữ cảnh của sơ đồ tuần tự. Ngữ cảnh có thể là một hệ hóng, một use case hoặc chỉ là một tình huống của một use. Ngữ cảnh của sơ đồ được thể hiện bằng một khung hình chữ nhật có gán nhãn bao quanh sơ đồ. Trường hợp hay gặp nhất là ngữ cảnh biểu diễn một tình huống của use case. Tại thời điểm phân tích và biểu diễn hệ thống, ta cố gắng hiểu hết và hiểu kỹ vấn đề. Vì vậy, việc tạo các sơ đồ tuần tự cho từng tình huống cụ thể thay vì tạo một sơ đồ tuần tự tổng quát cho cả use case sẽ giúp các nhà phát triển hiểu biết đầy đủ hơn về vấn đề đang giải quyết. Một sơ đồ tuần tự cho một tình huống cụ thể thường sẽ dễ hiểu, ngược lại, một sơ đồ tuần tự tổng quát có thể rất phức tạp. Việc kiểm tra một use case nào đó đã hoàn chỉnh hay chưa có thể dễ dàng thực hiện bằng cách kiểm tra và đánh giá tính đầy đủ của một tập các sơ đồ tuần tự cụ thể (chi tiết) thay vì cố gắng đánh giá một sơ đồ tuần tự tổng quát phức tạp.

Bước thứ hai là nhận biết các tác nhân và đối tượng tham gia theo trình tự được mô hình hóa. Nghĩa là phải tìm được các tác nhân và đối tượng tương tác với nhau trong tình huống của use case. Các tác nhân có thể được nhận biết thông qua việc tạo các mô hình chức năng trong khi các đối tượng được xác định thông qua việc phát triển mô hình cấu trúc (sơ đồ lớp). Vì mô hình hóa khía cạnh động của hệ thống là một quá trình lặp nên đừng quá quan trọng hóa việc tìm đầy đủ mọi đối tượng. Thông thường, các sơ đồ tuần tự sẽ được tinh chỉnh sau mỗi lần lặp.

Bước thứ ba là tạo ra đường sống cho mỗi đối tượng. Ở bước này, chỉ cần vẽ các đường thẳng dọc dứt nét vào bên dưới mỗi đối tượng. Nếu một đối tượng nào đó bị hủy sớm, trước các đối tượng còn lại thì cần thêm một dấu X ở cuối đường sống của nó.

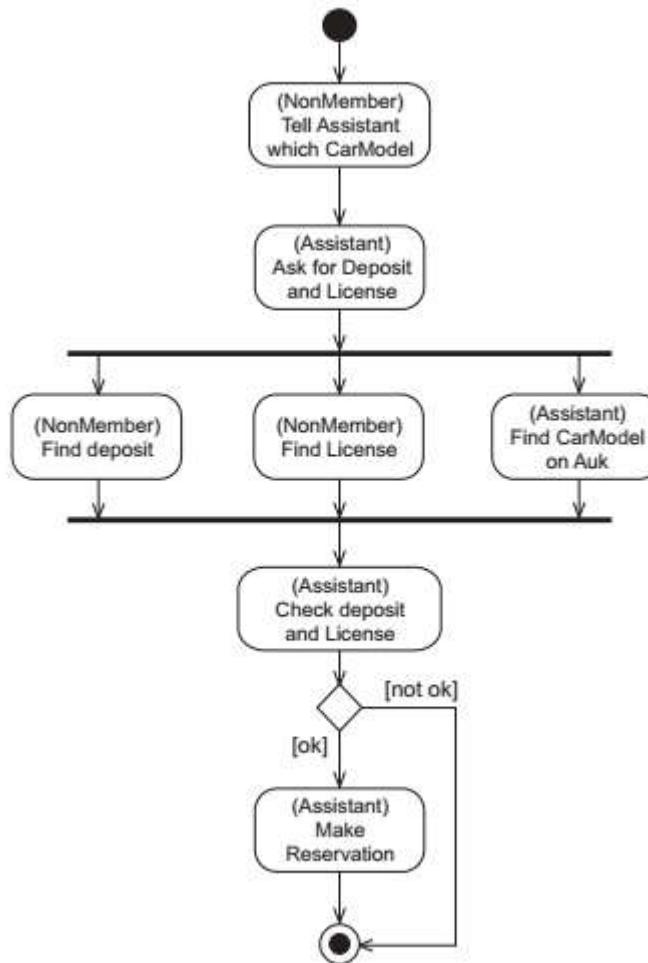
Bước thứ tư là thêm các thông điệp vào sơ đồ bằng cách vẽ các mũi tên nằm ngang hướng từ đối tượng này đến đối tượng khác. Hướng của mũi tên cho biết hướng mà thông điệp được truyền đi. Thông điệp được sắp xếp theo trình tự thời gian, từ trên xuống dưới. Các tham số đi kèm thông điệp phải đặt trong cặp dấu ngoặc đơn nằm sau tên thông điệp.

Bước thứ năm là đặt các đoạn thực thi (execution occurrence) nằm chồng lên đường sống của các đối tượng khi chúng gửi và nhận thông điệp. Các đoạn này được thể hiện bằng các hình chữ nhật hép nằm đè lên đường sống.

Bước cuối cùng là đánh giá sơ đồ tuần tự. Mục đích của bước này là bảo đảm rằng sơ đồ tuần tự biểu diễn đầy đủ quy trình công việc hay nói khác hơn là sơ đồ thể hiện được tất cả các bước của quy trình.

#### 4.6 Xây dựng sơ đồ hoạt động

Sơ đồ hoạt động được dùng để mô hình hóa các hoạt động trong quy trình nghiệp vụ, độc lập với các đối tượng hoặc cung cấp tài liệu mô tả một thuật toán. Sơ đồ hoạt động có thể được xem như một sơ đồ dòng dữ liệu phức tạp được dùng cho việc phân tích cấu trúc. Tuy nhiên, không giống như sơ đồ dòng dữ liệu, sơ đồ hoạt động có các ký hiệu để giải quyết việc mô hình hóa các hoạt động xảy ra đồng thời, song song và các quy trình quyết định phức tạp. Các sơ đồ hoạt động có thể được dùng để mô hình hóa mọi thứ từ dòng công việc nghiệp vụ mức cao liên quan tới nhiều use case cho tới chi tiết của một use case cụ thể. Tóm lại, sơ đồ hoạt động có thể được dùng để mô hình hóa mọi quy trình.



Hình 4.16 đồ họa hoạt động của use case nghiệp vụ NonMember Reserves CarModel

Các sơ đồ hoạt động mô tả các hoạt động chính và mối quan hệ giữa chúng trong một quy trình. Hình 4.16 cho thấy một sơ đồ hoạt động đơn giản biểu diễn use case nghiệp vụ “Khách hàng (không phải là thành viên) đặt hàng một mẫu xe hơi”. Từ sơ đồ này, dễ dàng rút ra được những điểm sau:

- Khách hàng NonMember (không phải là thành viên) nói với người bán hàng (Assistant) mẫu xe (CarModel) mà mình muốn đặt hàng.
- Người bán hàng yêu cầu đặt cọc và xem bằng lái.
- Trong khi khách hàng đang tìm bằng lái và đặt cọc thì người bán hàng tìm mẫu xe trong hệ thống Auk.
- Khi mọi thứ đã hoàn tất, người bán hàng kiểm tra bằng lái và việc đặt cọc.
- Nếu cả hai thứ đều hợp lệ, người bán hàng sẽ tạo ra một phiếu đặt hàng và hoạt động hoàn tất.

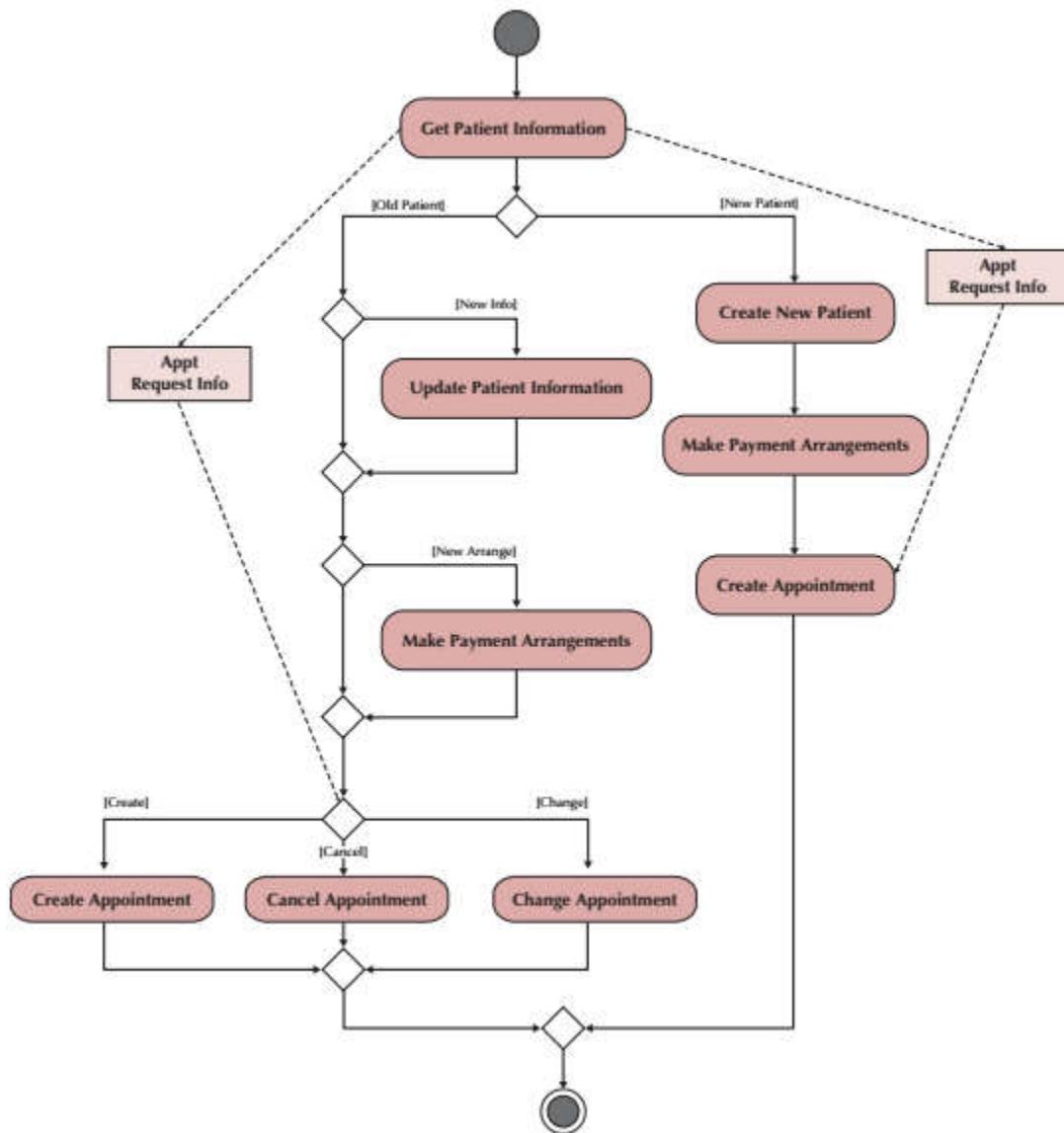
- Ngược lại, hoạt động hoàn tất.

#### 4.6.1 Các thành phần cơ bản của một sơ đồ hoạt động

Trong sơ đồ hoạt động, các *hành động* (action) hay *hoạt động* (activity) có thể mô tả các thao tác được làm bằng tay hoặc được tự động hóa bởi máy tính. Chúng được thể hiện bằng các hình chữ nhật góc tròn như trong hình 4.16. Mỗi hoạt động có một tên và bắt đầu bằng một động từ, kết thúc bằng một danh từ hoặc cụm danh từ. Tên phải ngắn gọn, súc tích, chỉ chứa thông tin vừa đủ để người đọc có thể dễ dàng hiểu chính xác những gì cần làm. Sự khác nhau duy nhất giữa một hành động (action) và một hoạt động (activity) là ở chỗ một hoạt động có thể được phân rã thành một tập các hoạt động nhỏ hơn hoặc hành động. Trái lại, một hành động biểu diễn một thao tác đơn giản không thể phân rã nhỏ hơn. Thông thường, các hoạt động chỉ được dùng trong việc mô hình hóa dòng công việc hoặc quy trình nghiệp vụ. Khi đó, mỗi hoạt động đi kèm với một use case. Chẳng hạn như trong sơ đồ ở hình 4.17, mỗi hoạt động như Update Patient Information, Make Payment Arrangements, Cancel Appointment có thể là một use case.

Trong sơ đồ hoạt động, các hành động và hoạt động thường biến đổi hoặc thay đổi trạng thái của các đối tượng nào đó. Các đối tượng này được mô hình hóa bởi các nút đối tượng (object nodes) và được thể hiện bằng các hình chữ nhật. Tên của lớp đối tượng được viết bên trong hình chữ nhật. Về cơ bản, các nút đối tượng biểu diễn luồng thông tin đi từ một hoạt động này sang một hoạt động khác. Hình 4.17 cho thấy nút đối tượng Appt Request Info (thông tin yêu cầu đặt hẹn) đi từ hoạt động Get Patient Information đến hoạt động Create Appointment.

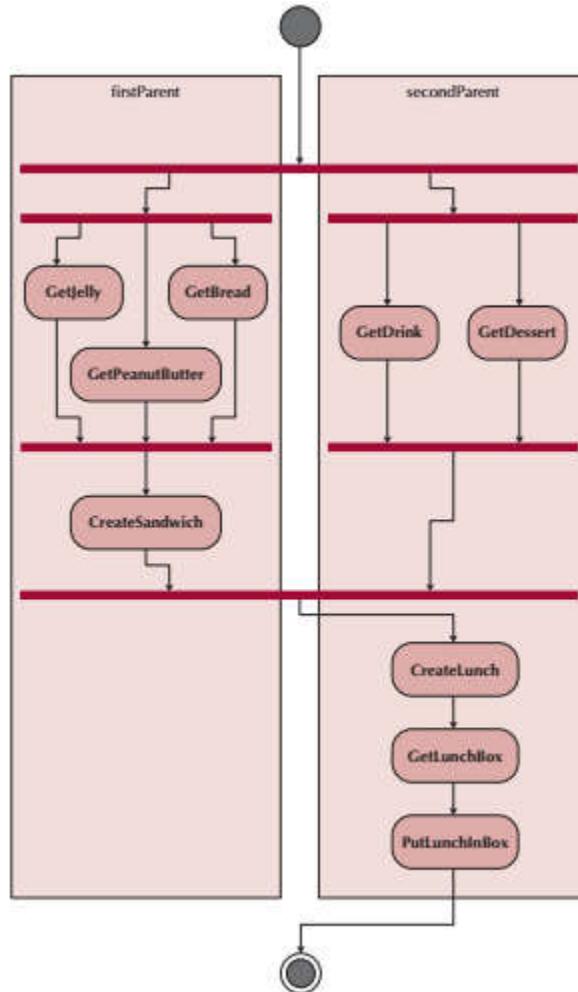
Trong sơ đồ hoạt động, có hai loại dòng thông tin: *dòng điều khiển* và *dòng đối tượng*. Dòng điều khiển (control flow) mô hình hóa luồng thực thi công việc trong quy trình nghiệp vụ. Dòng điều khiển được thể hiện trong sơ đồ bởi một đường mũi tên liền nét để chỉ ra hướng di chuyển của dòng. Dòng điều khiển chỉ được nối giữa các hành động hoặc hoạt động. Dòng đối tượng (object flow) mô hình hóa luồng di chuyển của các đối tượng trong một quy trình nghiệp vụ. Vì các hành động và hoạt động làm thay đổi hay biến đổi các đối tượng nên dòng đối tượng là cần thiết để chỉ ra các đối tượng thực sự đi vào và đi ra khỏi các hành động hay hoạt động. Một dòng đối tượng được thể hiện bởi một mũi tên nét đứt nhằm chỉ hướng di chuyển của đối tượng. Một đầu của mũi tên phải gắn vào một hành động hoặc hoạt động và đầu còn lại được gắn vào một nút đối tượng.



Hình 4.17 Sơ đồ hoạt động cho use case quản lý các cuộc hẹn (của bệnh nhân với bác sĩ)

Có tất cả 7 loại nút điều khiển (control node) trong sơ đồ hoạt động, bao gồm: khởi đầu (initial), kết thúc hoạt động (final-activity), kết thúc dòng (final-flow), quyết định (decision), hợp nhánh quyết định (merge), phân nhánh song song (fork) và hợp nhánh song song (join). Một *nút khởi đầu* mô tả sự khởi đầu một tập các hành động hay hoạt động. Nó được thể hiện bằng một hình tròn nhỏ được tô kín. Nút *kết thúc hoạt động* được dùng để dừng quy trình đang được mô hình hóa. Khi gặp nút kết thúc hoạt động, mọi hành động và hoạt động đều được kết thúc ngay lập tức, không cần biết chúng đã hoàn thành hay chưa. Nút này được biểu diễn bởi một hình tròn bao quanh một hình tròn nhỏ khác được tô kín. Nút *kết thúc dòng* tương tự như nút kết thúc hoạt động, ngoại trừ việc nó chỉ dừng một nhánh thực thi cụ thể trong quy trình nghiệp vụ và cho phép các nhánh khác đang được

thực hiện đồng thời hoặc song song tiếp tục thực hiện công việc của chúng. Nút kết thúc dòng được chỉ ra trên sơ đồ bởi một hình tròn nhỏ có dấu X bên trong.



Hình 4.18 Ví dụ về nút phân nhánh song song và phân tuyến

Các *nút quyết định* và *nút hợp nhánh quyết định* hỗ trợ việc mô hình hóa các cấu trúc quyết định, rẽ nhánh trong quy trình nghiệp vụ. Nút *quyết định* được dùng để biểu diễn một điều kiện kiểm tra để xác định xem sẽ đi tiếp nhánh nào. Các nhánh này có tính chất loại trừ lẫn nhau, nghĩa là chọn nhánh này thì không thể chọn nhánh khác. Trong trường hợp này, mỗi nhánh phải được gán nhãn là một điều kiện đảm bảo (guard condition). Một điều kiện đảm bảo biểu diễn một giá trị của phép kiểm tra. Chẳng hạn, trong hình 4.17, ngay sau hoạt động `Get Patient Information` có một nút quyết định, chia làm hai nhánh: một nhánh ứng với các bệnh nhân cũ đã từng đến khám và một nhánh ứng với các bệnh nhân mới. Nút *hợp nhánh quyết định* được dùng để hợp tất cả các nhánh đã chia trước đó bởi một nút quyết định.

Các *nút phân nhánh song song* và *nút hợp nhánh song song* cho phép mô hình hóa các tiền trình xảy ra đồng thời hoặc song song. Nút phân nhánh song song được dùng để tách một thao tác trong quy trình nghiệp vụ thành nhiều dòng song song hoặc chạy đồng thời. Khác với nút quyết định, các nhánh đi ra từ nút phân nhánh song song không có tính chất loại trừ lẫn nhau. Với trường hợp này, mỗi tiền trình sẽ được thực thi bởi một bộ xử lý riêng biệt. Mục đích của nút *hợp nhánh song song* cũng tương tự như nút hợp nhánh quyết định. Nó hợp tất cả các nhánh song song đã bị tách ra trước đó bởi nút phân nhánh song song.

Sơ đồ hoạt động có thể mô hình hóa một quy trình nghiệp vụ mà không phụ thuộc vào bất kỳ một cách thức thực thi đối tượng nào. Tuy vậy, có nhiều lúc, việc phân chia một sơ đồ hoạt động theo từng đối tượng có thể giúp gán trách nhiệm cụ thể cho từng đối tượng hoặc cá nhân thực hiện hoạt động đó. Điều này đặc biệt hữu ích khi mô hình hóa một dòng công việc nghiệp vụ. Việc chia sơ đồ theo đối tượng thực hiện hoạt động như vậy sẽ tạo ra các *phân tuyến* (swimlane) như trong hình 4.18.

#### 4.6.2 Một số lưu ý khi vẽ sơ đồ hoạt động

Khi vẽ sơ đồ hoạt động, chỉ vẽ một nút khởi đầu và đặt nó ở phía trên hoặc góc trên bên trái, tùy thuộc vào độ phức tạp của sơ đồ. Với hầu hết các quy trình nghiệp vụ, chỉ có duy nhất một nút kết thúc hoạt động. Nút này được đặt ở phía dưới cùng hoặc góc dưới bên phải của sơ đồ. Vì các quy trình nghiệp vụ mức cao có tính tuần tự, không chạy song song nên cần hạn chế sử dụng nút kết thúc dòng.

Khi mô hình hóa các quy trình nghiệp vụ mức cao hoặc dòng công việc, chỉ những quyết định quan trọng mới được thể hiện trong sơ đồ. Trong những trường hợp như thế, các điều kiện đảm bảo ứng với mỗi nhánh đi ra khỏi nút quyết định phải có tính chất loại trừ lẫn nhau. Các dòng đi ra và các điều kiện đảm bảo phải tạo thành một tập đầy đủ. Nghĩa là mọi giá trị khả dĩ của quyết định phải nằm trên các nhánh đi ra. Tương tự, các nút phân nhánh và hợp nhánh song song cũng chỉ nên biểu diễn trong sơ đồ hoạt động nếu các hoạt động song song đó là thực sự cần thiết và quan trọng.

Khi trình bày sơ đồ hoạt động phải hạn chế tối thiểu các đường cắt nhau để đảm bảo sơ đồ rõ ràng, dễ đọc. Các hoạt động nên sắp xếp theo thứ tự từ trái qua phải, từ trên xuống dưới theo trình tự mà chúng được thực hiện.

Các phân tuyến chỉ nên dùng khi muốn làm tăng tính rõ ràng, dễ đọc cho sơ đồ họa động. Hơn nữa, nó phải được sắp xếp một cách hợp lý. Ví dụ, khi sử dụng các phân tuyến được vẽ theo chiều nằm ngang thì phân tuyến trên cùng phải biểu diễn đối tượng hoặc cá nhân quan trọng nhất liên quan tới việc xử lý. Thứ tự của các phân tuyến còn lại được quyết

định để làm sao tối thiểu hóa số lần giao cắt của các dòng điều khiển. Nếu có các dòng đối tượng trong các hoạt động liên quan tới các cá nhân (hoặc phân tuyến) khác nhau thực hiện hoạt động xử lý thì tốt hơn hết là thêm một nút đối tượng vào giữa hai phân tuyến và kết nối dòng đối tượng với nút này.

Những hoạt động không có đầu ra (không cho ra kết quả) – còn gọi là hoạt động lỗ đen (black-hole activity) – và nó thực sự là một điểm để kết thúc trong sơ đồ thì cần phải vẽ một dòng điều khiển từ nó đến nút kết thúc hoạt động hoặc nút kết thúc dòng.

#### 4.6.3 Các bước để xây dựng sơ đồ hoạt động

Việc tạo sơ đồ hoạt động để làm tài liệu mô tả và mô hình hóa quy trình nghiệp vụ thường trải qua năm bước. Đầu tiên, phải chọn một quy trình nghiệp vụ đã xác định rõ để mô hình hóa. Để làm được điều này, cần phải xem lại định nghĩa các yêu cầu và sơ đồ use case đã tạo để biểu diễn các yêu cầu. Ngoài ra, toàn bộ tài liệu được tạo ra trong quá trình thu thập yêu cầu cũng cần phải được xem xét lại. Trong hầu hết các trường hợp thì cách tốt nhất là bắt đầu từ việc xem xét các use case trong sơ đồ use case.

Thứ hai là xác định tập các hoạt động cần thiết để hỗ trợ cho quy trình nghiệp vụ. Việc này được thực hiện bằng cách xem xét lại sơ đồ use case và tìm ra những use case chính liên quan tới quy trình nghiệp vụ.

Thứ ba là xác định các dòng điều khiển và các nút cần thiết để dẫn chứng sự logic của quy trình nghiệp vụ. Các use case thường chỉ được thực hiện khi thỏa một điều kiện nào đó. Vì thế, ta có thể suy ra rằng, sơ đồ hoạt động cũng phải có các nút quyết định và các nút hợp nhánh quyết định. Các nút này cũng có thể phát hiện được thông qua định nghĩa của các yêu cầu.

Thứ tư là xác định các dòng đối tượng và các nút cần thiết hỗ trợ cho sự logic của quy trình nghiệp vụ. Tuy nhiên, các nút và dòng đối tượng chỉ thực sự cần thiết khi thông tin thu thập bởi hệ thống trong một hoạt động này được dùng trong một hoạt động khác được thực thi sau đó (nhưng không phải là hoạt động ngay phía sau).

Cuối cùng, bố trí các thành phần và vẽ sơ đồ hoạt động để cung cấp tài liệu về quy trình nghiệp vụ. Cũng giống như khi vẽ sơ đồ use case, để cho rõ ràng, thẩm mỹ và dễ đọc, cần phải tối thiểu hóa số đường thẳng giao cắt nhau.

### 4.7 Kết chương

Chương này đã trình bày những nội dung chính sau:

- Phân tích yêu cầu phần mềm là gì? Tại sao giai đoạn phân tích lại quan trọng?

- Tổng quan về quá trình phân tích yêu cầu.
- Cách thức xây dựng một mô hình phân tích khía cạnh tĩnh của hệ thống nhằm chỉ ra các đối tượng liên quan đến nghiệp vụ trong hệ thống đề xuất.
- Cách xây dựng một sơ đồ lớp để thể hiện được các lớp đối tượng kém theo thuộc tính, mối quan hệ giữa chúng.
- Phân tích khía cạnh động của hệ thống nhằm cải tiến và xác thực mô hình tĩnh bằng cách dùng các sơ đồ tương tác.
- Xây dựng sơ đồ hoạt động để hiểu rõ hơn quy trình nghiệp vụ.

### Tài liệu tham khảo

1. Mike O'Docherty, “*Object-Oriented Analysis and Design Understanding System Development with UML 2.0*”, John Wiley & Sons, 2005, pp. 130-202.
2. Alan Dennis, Barbara Haley Wixom, David Tegarden, “*System Analysis Design UML Version 2.0: AN OBJECT-ORIENTED APPROACH – 4<sup>th</sup> Edition*”, John Wiley & Sons, 2012, pp. 152-270.
3. Roger S. Pressman, “*Software Engineering: A Practitioner's Approach – 7<sup>th</sup> Edition*”, McGraw-Hill, 2010, pp. 148-215.

## CHƯƠNG 5. THIẾT KẾ PHẦN MỀM

### 5.1 Giới thiệu

#### 5.1.1 Khái niệm

Sau giai đoạn phân tích, khi các yêu cầu cụ thể đối với hệ thống đã được xác định, chúng ta sẽ chuyển qua giai đoạn tiếp theo là thiết kế phần mềm. Đây là một giai đoạn vô cùng quan trọng trong quá trình phát triển dự án phần mềm. Kết quả của giai đoạn thiết kế phần mềm là bản Đặc tả Thiết Kế (Design Specifications) - được xem như là một bản hướng dẫn cho các hoạt động xây dựng phần mềm sau này.

Trong thiết kế phần mềm, chúng ta đi xây dựng mô hình cho một hệ thống, mà phải đáp ứng được mọi yêu cầu, bao gồm cả các yêu cầu phi chức năng và các ràng buộc được đặt ra cho hệ thống đó. Đầu vào cơ bản cho thiết kế phần mềm là kết quả thu được từ quá trình phân tích các yêu cầu. Bản chất thiết kế phần mềm là một quá trình chuyển hóa các yêu cầu phần mềm thành một biểu diễn thiết kế, nó tạo ra một mô hình biểu diễn của một thực thể mà sau này sẽ được xây dựng.

Thiết kế phần mềm là cách chuyển hóa chính xác các yêu cầu thành mô hình thiết kế hệ thống phần mềm cuối cùng, làm cơ sở cho việc triển khai chương trình phần mềm. Nó là tài liệu cung cấp đầy đủ các thông tin cần thiết cho việc bảo trì hệ thống. Một thiết kế tốt là chìa khóa cho một phần mềm hữu hiệu.

#### 5.1.2 Mục tiêu

Xét một cách chi tiết mục tiêu của thiết kế là:

- Thu được sự hiểu biết sâu về các yêu cầu phi chức năng và các ràng buộc có liên quan tới ngôn ngữ lập trình, sử dụng lại thành phần, các hệ điều hành, các công nghệ phân tán, các công nghệ cơ sở dữ liệu, các công nghệ giao diện người dùng, các công nghệ quản lý các giao dịch.
- Tạo ra một đầu vào thích hợp và làm xuất phát điểm cho các hoạt động cài đặt tiếp theo sau bằng cách nắm bắt các yêu cầu về mỗi hệ thống cụ thể, các giao diện, các lớp.
- Có khả năng phân rã việc cài đặt thành các mảng nhỏ dễ quản lý hơn, có thể được phát triển bởi nhiều nhóm khác nhau và có thể tiến hành đồng thời.

- Nắm bắt sớm các giao diện chủ yếu giữa các hệ thống con trong vòng đời của phần mềm. Điều này sẽ có ích khi chúng ta suy luận về kiến trúc hệ thống và khi sử dụng các giao diện như những công cụ đồng bộ giữa các nhóm phát triển khác nhau.
- Trực quan hóa và suy luận thiết kế bằng cách sử dụng một hệ thống các ký pháp chung.
- Tạo ra một sự trừu tượng hóa liên tục của việc cài đặt của hệ thống, tức là cài đặt sự làm mịn dần thiết kế bằng cách đắp “thịt” vào khung xương nhưng không thay đổi cấu trúc của nó.

### 5.1.3 Quá trình thiết kế

Thiết kế phần mềm là quá trình thiết kế cấu trúc phần mềm dựa trên những tài liệu đặc tả. Hoạt động thiết kế bao gồm những công việc chính sau:

- Thiết kế dữ liệu: Cấu trúc dữ liệu được sử dụng để cài đặt hệ thống phải được thiết kế một cách chi tiết và cụ thể.
- Thiết kế xử lý: Chi tiết các hàm xử lý phục vụ cho các chức năng của hệ thống.
- Thiết kế giao diện: với mỗi hệ thống con, các giao diện của nó với những hệ thống con khác phải được thiết kế và tư liệu hoá.
- Xây dựng sơ đồ lớp.
- Xây dựng sơ đồ trạng thái.

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 tiếp sau.

## 5.2 Thiết kế dữ liệu

### 5.2.1 Mục tiêu

Mục tiêu chính của thiết kế dữ liệu là mô tả cách thức tổ chức lưu trữ các dữ liệu của phần mềm. Có hai dạng lưu trữ chính mà người thiết kế cần phải cân nhắc và lựa chọn. Đó là:

- Lưu trữ dưới dạng tập tin
- Lưu trữ dưới dạng cơ sở dữ liệu

Lưu trữ dưới dạng tập tin thường chỉ thích hợp với một số phần mềm đặc thù (cờ tướng, trò chơi nhỏ, v.v.) đặc điểm chung của các phần mềm này là chú trọng rất nhiều vào xử lý,

hình thức giao diện và không chú trọng nhiều đến việc lưu trữ lại các thông tin được tiếp nhận trong quá trình sử dụng phần mềm (thông thường các thông tin này được tiếp nhận và xử lý ngay).

Cách tiếp cận dùng cơ sở dữ liệu rất thông dụng và tỏ ra hiệu quả hơn so với lưu trữ dữ liệu tập tin. Giáo trình này sẽ giới hạn trình bày chi tiết các phương pháp kỹ thuật liên quan đến việc tổ chức lưu trữ dữ liệu dùng cơ sở dữ liệu quan hệ. Giáo trình này sẽ không nhắc lại các khái niệm cơ bản về cơ sở dữ liệu và giả sử rằng người xem đã biết về các khái niệm này. Tuy nhiên chúng ta cũng nên xem lại các bước để hình thành nên mô hình dữ liệu quan hệ trong quá trình thiết kế dữ liệu.

### 5.2.2 Kết quả của thiết kế dữ liệu

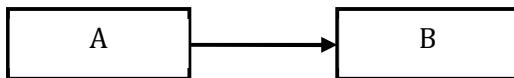
Cách thức tổ chức lưu trữ dữ liệu của phần mềm được mô tả thông qua hai loại thông tin sau:

- Thông tin tổng quát: Cung cấp góc nhìn tổng quan về các thành phần lưu trữ:
  - Danh sách các bảng dữ liệu: Việc lưu trữ cần sử dụng bao nhiêu bảng dữ liệu và đó là các bảng nào?
  - Danh sách các liên kết: Các bảng dữ liệu có quan hệ gì, mối liên kết giữa chúng ra sao?
- Thông tin chi tiết:
  - Danh sách các thuộc tính của từng thành phần: Các thông tin cần lưu trữ của thành phần?
  - Danh sách các miền giá trị toàn vẹn: Các qui định về tính hợp lệ của các thông tin được lưu trữ.

Có nhiều phương pháp, nhiều đề nghị khác nhau về việc mô tả các thông tin trên. Giáo trình này chọn sơ đồ logic để biểu diễn các thông tin tổng quát và bảng thuộc tính để mô tả chi tiết các thành phần trong sơ đồ logic.

- Sơ đồ logic là sơ đồ cho phép thể hiện hệ thống các bảng dữ liệu cùng với mối quan hệ liên kết giữa chúng. Các ký hiệu được dùng trong sơ đồ rất đơn giản như sau:

Mô tả	Ký hiệu
Bảng: hình chữ nhật	Tên bảng
Liên kết: (xác định duy nhất một): Mũi tên	————→



Hình trên có nghĩa là một phần tử bảng A sẽ xác định duy nhất một phần tử bảng B, ngược lại một phần tử bảng B có thể tương ứng với nhiều phần tử bảng A.

Ví dụ: Với hệ thống quản lý mượn sách của thư viện, có sơ đồ logic sau:



Theo sơ đồ này việc lưu trữ các dữ liệu trong phần mềm quản lý mượn sách của thư viện được tổ chức thành ba bảng: DOC\_GIA, MUON\_SACH, SACH cùng với hai liên kết giữa chúng.

- Bảng thuộc tính cho phép mô tả chi tiết thành phần trong sơ đồ logic theo dạng như sau:

STT	Thuộc tính	Kiểu	Miền giá trị	Ý nghĩa	Ghi chú
1					
2					
...					

Bảng thuộc tính cho phép mô tả chi tiết các thành phần cần lưu trữ và sẽ được dùng trong báo cáo về thiết kế dữ liệu của phần mềm. Tuy nhiên cách mô tả trên khá dài dòng, ta có cách trình bày cô đọng hơn theo dạng lược đồ quan hệ. Dạng trình bày này gồm có tên bảng và các thuộc tính đi kèm, các thuộc tính khóa chính được gạch chân nét liền, các thuộc tính khóa ngoại được gạch chân nét đứt.

**Ví dụ:**

DOC\_GIA (MaDG, LoaiDG, HoTen, NgaySinh, NgayLapThe, GioiTinh, DiaChi, SoDT)

SACH(MaSach, TenSach, TheLoai, NgayNhap, TacGia, NhaXB, NamXB, NgonNgu)

MUON\_SACH(MaDG, MaSach, NgayMuon, NgayTra)

### 5.2.3 Quá trình thiết kế

Quá trình thiết kế dữ liệu bao gồm 3 bước lớn tương ứng với 3 yêu cầu của phần mềm:

- Thiết kế với tính đúng đắn
  - Đảm bảo đầy đủ và chính xác về mặt ngữ nghĩa các thông tin liên quan đến các công việc trong yêu cầu.
  - Các thông tin phục vụ cho các yêu cầu chất lượng sẽ không được xét đến trong bước thiết kế này.
- Thiết kế với yêu cầu chất lượng
  - Vẫn đảm bảo tính đúng đắn nhưng thỏa mãn thêm các yêu cầu chất lượng khác (tiến hóa, tốc độ nhanh, lưu trữ tối ưu).
  - Cần chú ý bảo đảm tính đúng đắn khi cài tiến sơ đồ logic.
- Thiết kế với yêu cầu hệ thống
  - Vẫn đảm bảo tính đúng đắn và các yêu cầu chất lượng khác nhưng thỏa mãn thêm các yêu cầu hệ thống (phân quyền, cấu hình phần cứng, môi trường phần mềm, v.v)

#### 5.2.3.1 Thiết kế dữ liệu với tính đúng đắn

Các bước thực hiện:

*Bước 1:* Chọn một yêu cầu và xác định sơ đồ logic cho yêu cầu đó.

*Bước 2:* Bổ sung thêm một yêu cầu và xem xét lại sơ đồ logic

- Nếu sơ đồ logic vẫn đáp ứng được thì tiếp tục bước 3
- Nếu sơ đồ logic không đáp ứng được thì bổ sung vào sơ đồ thuộc tính mới (ưu tiên 1) hoặc thành phần mới (ưu tiên 2) cùng với các thuộc tính và liên kết tương ứng.

*Bước 3:* Quay lại bước 2 cho đến khi đã xem xét đầy đủ các yêu cầu

#### Ghi chú

- Với mỗi yêu cầu phải xác định rõ cần lưu trữ các thông tin gì? (dựa vào luồng dữ liệu đọc/ghi trong sơ đồ luồng dữ liệu tương ứng) và tìm cách bổ sung các thuộc tính để lưu trữ các thông tin này.
- Chỉ xem xét tính đúng đắn.

- Cần chọn các yêu cầu theo thứ tự từ đơn giản đến phức tạp (thông thường yêu cầu tra cứu là đơn giản nhất).
- Với yêu cầu phức tạp có thể phải bổ sung vào sơ đồ logic nhiều thành phần mới. Khóa của các thành phần phải dựa trên ngữ nghĩa tương ứng trong thế giới thực.

**Ví dụ:** Xét phần mềm với các yêu cầu sau:

1. Lập thẻ độc giả.
2. Nhận sách.
3. Cho Mượn/Trả sách.

Bước 1: Xét yêu cầu 1:

- Lập bảng DOC\_GIA.
- Chi tiết bảng: DOC\_GIA(MaDG, LoaiDG, HoTen, NgaySinh, NgayLapThe, GioiTinh, DiaChi, SoDT)
- Sơ đồ logic: 

DOC_GIA
---------

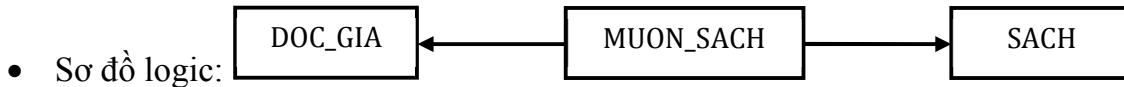
Bước 2: Xét yêu cầu 1, 2: Thông tin mới: Tên sách, Tác giả, Nhà xuất bản, Năm xuất bản, Thể loại, Ngày nhập, Ngôn ngữ.

- Tạo thêm bảng SACH.
- Chi tiết bảng: SACH(MaSach, TenSach, TheLoai, NgayNhap, TacGia, NhaXB, NamXB, NgonNgu)



Bước 3: Xét yêu cầu 1, 2, 3: Thông tin mới: Ngày mượn, Ngày trả, Tiền phạt.

- Tạo thêm bảng MUON\_SACH.
- Chi tiết bảng MUON\_SACH(MaDG, MaSach, NgayMuon, NgayTra, TienPhat)



### 5.2.3.2 Thiết kế dữ liệu với yêu cầu chất lượng

Xem xét đánh giá sơ đồ logic theo các yêu cầu về chất lượng và tiến hành cập nhật lại sơ đồ để bảo đảm các tiêu chuẩn về chất lượng. Ngoài tính đúng đắn cần ưu tiên hàng đầu thì xem xét sự hợp lý hơn nhau giữa các phần mềm chính là mức độ thỏa mãn các tiêu chuẩn chất lượng còn lại (đặc biệt là tính tiến hóa).

#### a. Tính tiến hóa

Để bảo đảm tính tiến hóa, sơ đồ logic sẽ còn bổ sung cập nhật lại nhiều thành phần qua các bước thiết kế chi tiết. Trong các bước đầu tiên là thiết kế dữ liệu, nó sẽ giới hạn xem xét đến các thuộc tính có giá trị rời rạc.

Thuộc tính có giá trị rời rạc là các thuộc tính mà miền giá trị chỉ bao gồm một số giá trị nhất định. Các giá trị này thông thường thuộc về tập hợp có độ biến động rất ít trong quá trình sử dụng phần mềm.

Tuy nhiên cần lưu ý rằng khả năng biến động trên tập hợp giá trị của thuộc tính rời rạc là thấp nhưng không phải là không có. Và khi xảy ra biến động (ví dụ như thêm mới loại độc giả, thêm sách thuộc ngôn ngữ mới) nếu không chuẩn bị trước trong thiết kế thì người dùng sẽ không thể khai báo được các biến động này với phần mềm, và do đó có thể một số chức năng sẽ không thực hiện được (ví dụ như không thể thêm sách mới với ngôn ngữ tiếng Ả Rập).

Để chuẩn bị tốt cho biến động về sau (nếu có) trong tập hợp các giá trị của thuộc tính rời rạc. Chúng ta sẽ tách các thuộc tính này thành một thành phần trong sơ đồ logic. Khi đó người dùng trong quá trình sử dụng hoàn toàn có thể cập nhật lại tập hợp các giá trị này tương ứng với các biến động thực tế trong thế giới thực.

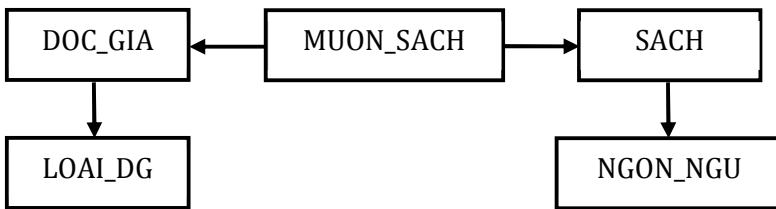
#### Ví dụ:

Trong bảng DOC\_GIA: LoaiDG là thuộc tính có khả năng thêm mới là rất thấp.

Trong bảng SACH: NgonNgu là thuộc tính có khả năng thêm mới là rất thấp.

Tuy nhiên, để đảm bảo tính tiến hóa, ta sẽ tiến hành tách các thuộc tính rời rạc này thành các thành phần độc lập.

Kết quả khi tách các thuộc tính rời rạc như sau:



Chi tiết các bảng:

DOC\_GIA(MaDG, MaLDG, HoTen, NgaySinh, NgayLapThe, GioiTinh, DiaChi, SoDT)

LOAI\_DG(MaLDG, TenLDG, GhiChu)

SACH(MaSach, TenSach, TacGia, TheLoai, MaNN, NhaXB, NamXB, NgayNhap)

NGON\_NGU(MaNN, TenNN)

MUON\_SACH(MaDG, MaSach, NgayMuon, NgayTra, TienPhat)

### b. Tính hiệu quả về tốc độ xử lý

Phạm vi xem xét:

- Chỉ giới hạn xem xét việc tăng tốc độ thực hiện của phần mềm bằng cách bổ sung thêm các thuộc tính vào các bảng dùng lưu trữ các thông tin đã tính toán trước (theo qui tắc nào đó từ các thông tin gốc đã được lưu trữ)
- Các thông tin này phải được tự động cập nhật khi có bất kỳ thay đổi thông tin gốc liên quan

Các bước tiến hành:

- Bước 1: Chọn một yêu cầu và xem xét cần bổ sung thông tin gì trên bộ nhớ phụ để tăng tốc độ thực hiện của xử lý liên quan (các thông tin xử lý phải đọc mà không cần thực hiện việc tính toán)
- Bước 2: Quay lại bước 1 cho đến khi đã xem xét đầy đủ các yêu cầu

Ghi chú:

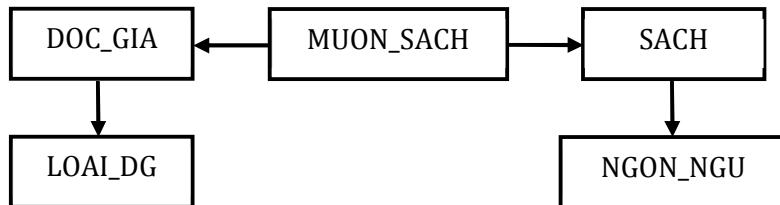
- Sau mỗi bước nhất thiết phải lập bảng danh sách các thuộc tính tính toán cùng với thông tin liên quan, bao gồm:
  - Thông tin gốc

- Xử lý tự động cập nhật thông tin gốc (chi tiết về các xử lý này sẽ được mô tả trong phần thiết kế xử lý)
- Nếu thông tin gốc thường xuyên bị thay đổi, việc bổ sung thuộc tính tính toán để tăng tốc độ thực hiện sẽ mất ý nghĩa (thậm chí gây phản tác dụng).
- Việc tăng tốc độ truy xuất có thể sẽ dẫn đến việc lưu trữ không tối ưu.
- Thứ tự xem xét các yêu cầu theo thứ tự từ đầu đến cuối.

**Ví dụ: Bổ sung thêm các yêu cầu sau**

1. Mỗi thẻ độc giả có thời hạn 03 năm.
2. Cho biết trạng thái sách là đang được mượn hay không.

Với sơ đồ logic cũng như ví dụ trước:



- Bước 1: Xét yêu cầu 1: Thêm thuộc tính Ngày hết hạn cho bảng **DOC\_GIA**:  
**DOC\_GIA(MaDG, LoaiDG, HoTen, NgaySinh, NgayLapThe, NgayHetHan, GioiTinh, DiaChi, SoDT)**
- Bước 2: Xét yêu cầu 2: Thêm thuộc tính Tình trạng mượn cho bảng **SACH**  
**SACH(MaSach, TenSach, TacGia, TheLoai, NgonNgu, NhaXB, NamXB, NgayNhap, TinhTrangMuon)**

**c. Tính hiệu quả về lưu trữ thông tin**

Tính hiệu quả trong thiết kế dữ liệu sẽ được xem xét dưới góc độ lưu trữ có tối ưu hay không. Vấn đề đặt ra là xây dựng sơ đồ logic sao cho vẫn bảo đảm lưu trữ đầy đủ thông tin theo yêu cầu nhưng với dung lượng lưu trữ nhỏ nhất có thể có. Vấn đề này đặc biệt quan trọng với các phần mềm phải sử dụng hệ thống lưu trữ lớn và nhiều phát sinh thông tin cần lưu trữ theo thời gian.

Khi đó cần đặc biệt quan tâm đến các thành phần mà dữ liệu tương ứng sẽ được phát sinh nhiều theo thời gian. Chúng ta sẽ tìm cách bố trí lại sơ đồ logic sao cho vẫn đảm bảo thông tin mà dung lượng lưu trữ lại ít hơn.

**Các bước tiến hành:**

Bước 1: Lập danh sách các bảng cần được xem xét để tối ưu hóa việc lưu trữ

- Xem xét và xác định các công việc có tần suất thực hiện thường xuyên và bổ sung vào danh sách các bảng được sử dụng tương ứng của công việc này.
- Xem xét các bảng mà khóa của bảng bao gồm nhiều thuộc tính và bổ sung bảng này vào danh sách được chọn.

Bước 2: Tối ưu hóa việc lưu trữ các bảng có khối lượng dữ liệu lưu trữ lớn thông qua việc tối ưu hóa lưu trữ từng thuộc tính trong bảng.

- Xác định các thuộc tính mà việc lưu trữ chưa tối ưu. Ưu tiên xem xét các thuộc tính có kiểu chuỗi.
- Tối ưu hóa việc lưu trữ tùy theo từng trường hợp cụ thể.
- Một trong các trường hợp thông dụng nhất là chuỗi có kích thước lớn và giá trị được sử dụng nhiều lần trong các mẫu tin khác nhau (ví dụ: thuộc tính TacGia, NhaXB trong bảng SACH của phần mềm quản lý sách) .
- Với trường hợp trên việc tối ưu hóa có thể thực hiện thông qua việc bổ sung các bảng mới (bảng TAC\_GIA, NHA\_XB) và tổ chức cấu trúc bảng SACH (thay thuộc tính TacGia bằng MaTG, thay thuộc tính Nha\_XB bằng MaNxb)

Bước 3: Tối ưu hóa các bảng mà khóa của bảng bao gồm nhiều thuộc tính.

- Phân rã bảng đang xét thành hai bảng. Trong đó, một bảng chứa các thuộc tính mà giá trị được lặp lại nhiều lần trong cùng một lần thực hiện công việc tương ứng trong thế giới thực. Bảng này cần có khóa riêng (sẽ được bảng còn lại sử dụng để tham chiếu đến)
- Ghi chú: Việc phân rã giúp cho việc lưu trữ được tối ưu hơn, tuy nhiên tốc độ truy xuất có thể sẽ chậm hơn và việc thực hiện xử lý sẽ khó khăn hơn (do thuật giải sẽ trở nên phức tạp hơn). Cho nên cần cân nhắc trước khi thực hiện việc phân rã.

#### Ví dụ:

Bước 1: Lập danh sách các bảng cần xem xét để tối ưu hóa lưu trữ:

MUON\_SACH(MaDG, MaSach, NgayMuon, NgayTra, TienPhat): việc mượn sách có tần suất thực hiện thường xuyên, khóa của bảng MUON\_SACH gồm có 02 thuộc tính.

Bước 2: Xác định các thuộc tính mà việc lưu trữ chưa tối ưu

Bảng SACH(MaSach, TenSach, TheLoai, NgayNhap, TacGia, NhaXB, NamXB, NgonNgu): có các thuộc tính lưu trữ chưa tối ưu: TheLoai, TacGia, NhaXB.

Bước 3:

- Phân rã bảng MUON\_SACH thành 2 bảng: MUON\_SACH và CT\_MUON (chi tiết mượn):

MUON\_SACH(MaCTM, TienPhat)

CT\_MUON(MaCTM, MaDG, MaSach, NgayMuon, NgayTra)

- Phân rã bảng SACH thành các bảng: SACH, THE\_LOAI, TAC\_GIA, NHA\_XB:

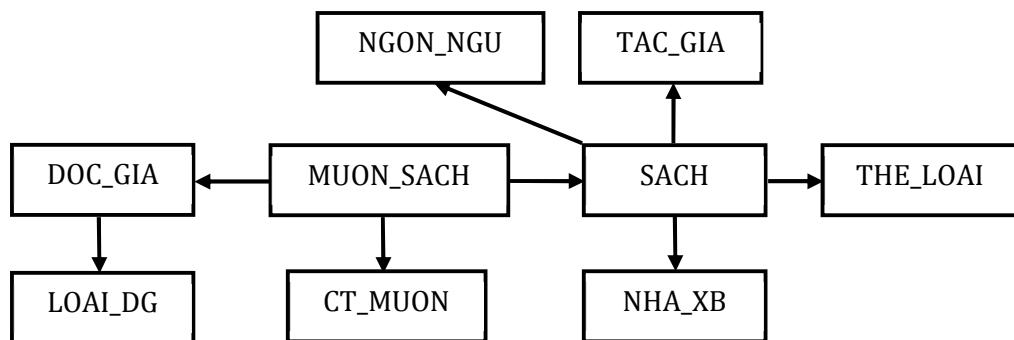
SACH(MaSach, TenSach, MaTG, MaTL, MaNN, MaNXB, NamXB, NgayNhap, TinhTrangMuon)

TAC\_GIA(MaTG, HoTen, DiaChi, SoDT)

THE\_LOAI(MaTL, TenTL)

NHA\_XB(MaNXB, TenNXB, NgayThanhLap)

Sơ đồ logic:



### 5.2.3.3 Thiết kế dữ liệu với yêu cầu hệ thống

Cải tiến phần mềm bằng việc thêm vào các yêu cầu hệ thống (như: phân quyền, cấu hình phần cứng, môi trường phần mềm, v.v), nhưng vẫn đảm bảo tính đúng đắn và các yêu cầu chất lượng.

**Ví dụ:** Yêu cầu phân quyền cho hệ thống phần mềm quản lý thư viện.

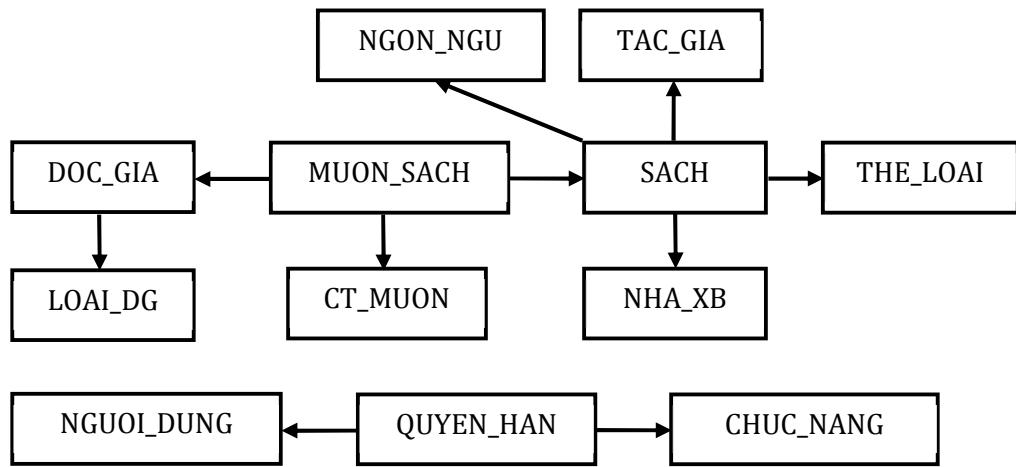
Bổ sung các bảng:

NGUOI\_DUNG(MaND, TenND)

CHUC\_NANG(MaCN, TenCN, GhiChu)

QUYEN\_HAN(MaND, MaCN)

Sơ đồ logic:



### 5.3 Thiết kế xử lý

Thiết kế xử lý là bước xây dựng mô tả các hàm xử lý và hằng, biến, kiểu dữ liệu liên quan tương ứng với các chức năng yêu cầu của dự án. Từ danh sách các yêu cầu ở phần trước, chúng ta có thể lập ra một số danh sách xử lý tùy thuộc vào yêu cầu và quy mô của dự án. Thiết kế xử lý giúp định hình được nội dung xây dựng mã nguồn cần phải làm để phân chia công việc hiệu quả.

#### 5.3.1 Các danh sách thiết kế xử lý

##### 5.3.1.1 Danh sách các kiểu dữ liệu xử lý

Danh sách này liệt kê các kiểu dữ liệu cần thiết để xử lý trong chương trình, kèm theo là ý nghĩa và ghi chú tương ứng.

STT	Kiểu dữ liệu	Ý nghĩa	Ghi chú
...	...	...	...

##### 5.3.1.2 Danh sách các thuộc tính kiểu dữ liệu X

Danh sách này liệt kê các thuộc tính của dữ liệu nào đó với các thông tin chi tiết hơn.

STT	Thuộc tính	Kiểu	Ràng buộc	Giá trị khởi động	Ghi chú
...	...	...	...	...	...

### 5.3.1.3 Danh sách các hàm, phương thức xử lý

Thông thường, các dự án phải lập danh sách các hàm xử lý vì chúng là các hàm quan trọng trong việc tạo sự tương tác kết nối giữa dữ liệu và giao diện chương trình. Các hàm xử lý được định nghĩa trong bảng sau:

ST T	Hàm	Tham số	Kết quả trả về	Thuật giải	Ý nghĩa	Ghi chú
1	InsertStudent	StudentName (string), Birthday (datetime), Address (string)	Kiểu bool với 2 giá trị, "true" là nhập thành công, "false" là nhập không thành công	Lấy các tham số, kiểm tra kiểu dữ liệu và nhập vào hệ thống	Nhập 1 sinh viên vào cơ sở dữ liệu	
2	DeleteStudent	StudentID (int)	Kiểu bool với 2 giá trị, "true" là xóa thành công, "false" là xóa không thành công	Lấy tham số StudentID, kiểm tra dữ liệu và thực hiện thao tác xóa	Xóa 1 sinh viên dựa theo StudentID	

Ví dụ phương thức **InsertStudent** thì cần 3 tham số đầu vào là StudentName, Birthday, Address và kết quả trả về là kiểu bool. Chúng ta có thể hình dung mã giả khi xây dựng chương trình như sau:

```
bool InsertStudent(string StudentName, DateTime Birthday, string Address)
{
    //Kiểm tra các dữ liệu đầu vào
    if (nhập thành công) trả về true
    else trả về false
}
```

#### 5.3.1.4 Danh sách các biến

Sau khi có danh sách các hàm xử lý thì chúng ta có thể liệt kê danh sách các biến sử dụng trong các hàm và chương trình. Các biến lập trong danh sách này là biến toàn cục, dùng chung cho hệ thống, các biến cục bộ thông thường là các tham số của hàm như danh sách các hàm xử lý phần trước. Các biến có thể lập theo danh sách sau:

STT	Biến	Kiểu	Ý nghĩa	Ghi chú

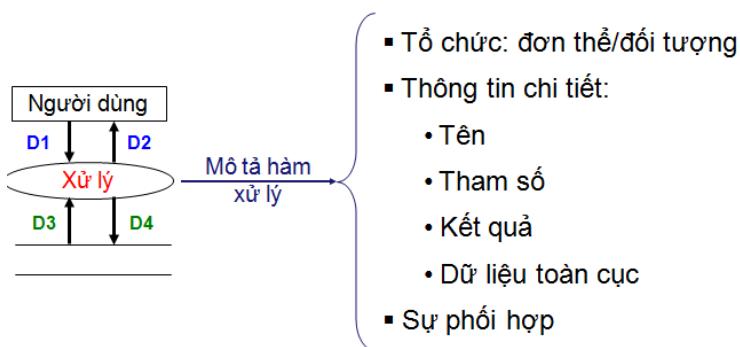
#### 5.3.1.5 Danh sách các hằng

Tương tự danh sách các biến, chúng ta cũng có thể lập danh sách các hằng số (biến không đổi) cho chương trình. Các hằng cũng được xem là dạng biến toàn cục, sử dụng chung cho toàn bộ chương trình phần mềm.

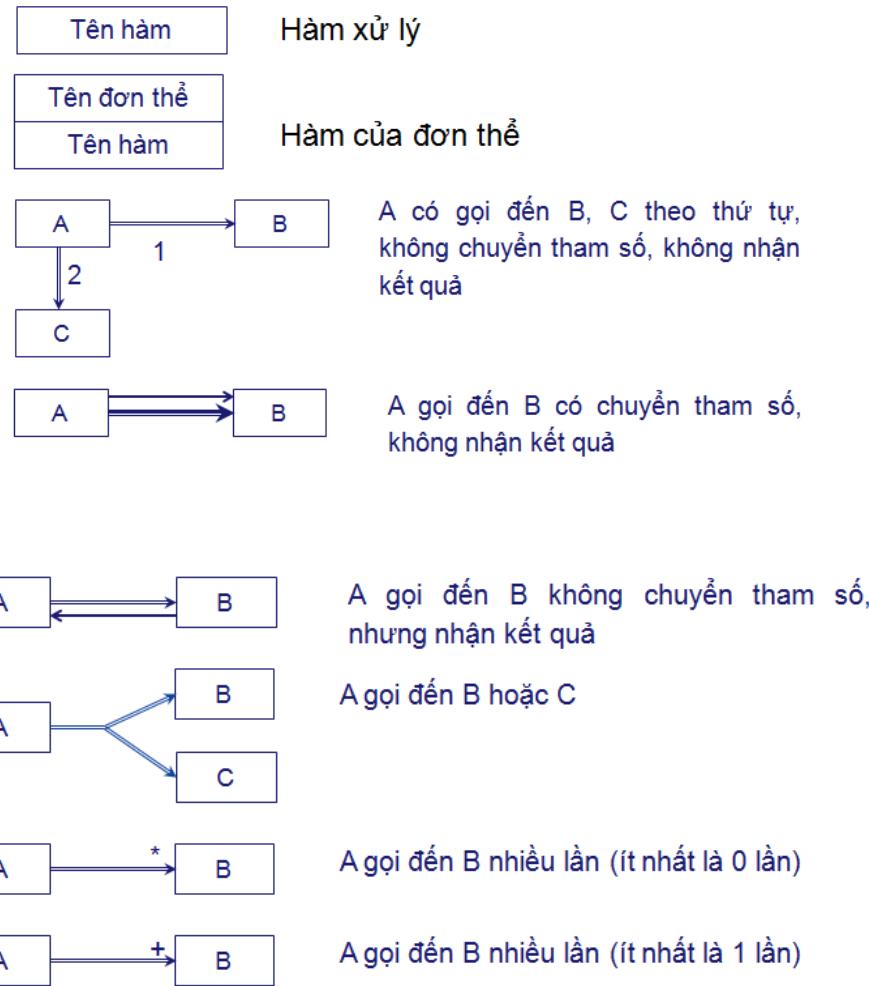
STT	Hằng	Kiểu	Giá trị	Ý nghĩa	Ghi chú

#### 5.3.2 Mô tả hàm xử lý

Sau khi lập danh sách các xử lý, chúng ta phải mô tả chi tiết các xử lý và các sơ đồ phối hợp này để có cái nhìn tổng quát hơn.



Yêu cầu cho xử lý phải đảm bảo tính đúng đắn, dễ bảo trì, tái sử dụng và di chuyển tốt. Một số ký hiệu trong việc mô tả xử lý như sau:



Chú ý nếu có n biến cỗ cần xử lý thì tương ứng phải có n sơ đồ phối hợp.

**Ví dụ:** Xét đến màn hình tiếp nhận một học sinh mới như sau

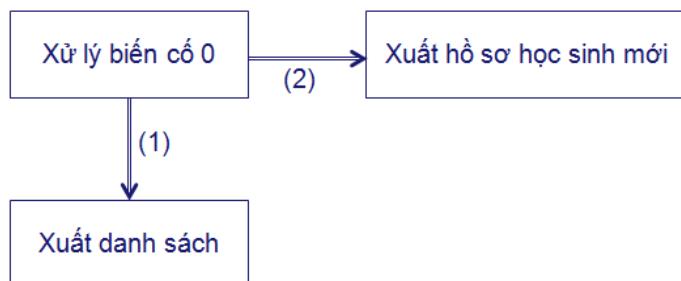
Tiếp nhận học sinh				
Họ tên	<input type="text"/>	Nam <input checked="" type="checkbox"/>		
Ngày sinh	<input type="text"/>	Lớp <input type="text"/>		
Địa chỉ	<input type="text"/>			<input type="button" value="Ghi"/>
Danh sách học sinh đã tiếp nhận				
STT	Mã HS	Tên HS	Giới tính	Ngày sinh
...	...	...	...	...

Đầu tiên, chúng ta hãy mô tả các biến cő (các trường hợp kết quả có thể) cho màn hình trên và lập danh sách chi tiết biến cő như sau:

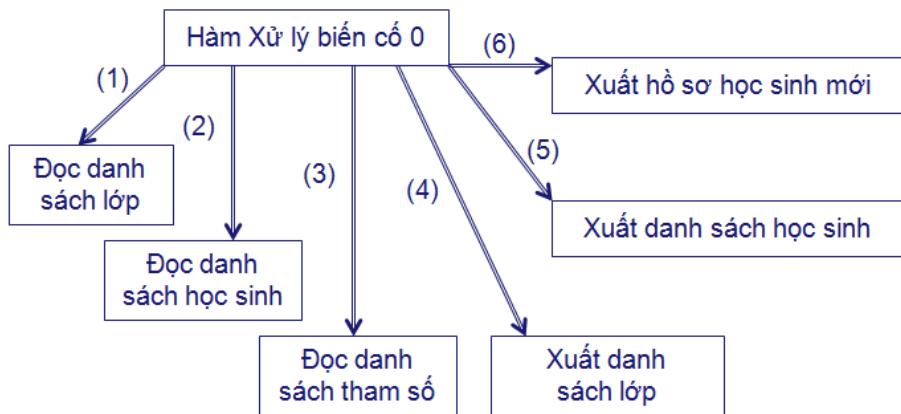
- Biến cő 0: Khởi động màn hình
- Biến cő 1: Kiểm tra tuổi học sinh hợp lệ (tuổi từ 15 đến 20)
- Biến cő 2: Khi chọn một lớp học trên combobox
- Biến cő 3: Kiểm tra dữ liệu hợp lệ và ghi

Biến cő	Điều kiện kích hoạt	Xử lý	Ghi chú
0	Khởi động màn hình	Đọc danh sách lớp, danh sách học sinh, tham số Xuất danh sách lớp, danh sách học sinh, hồ sơ học sinh mới	
1	Kết thúc nhập ngày sinh	Kiểm tra ngày sinh hợp lệ và xuất thông báo lỗi nếu không hợp lệ	Tuổi theo qui định từ 15 đến 20
2	Kết thúc chọn lớp	Ghi nhận vị trí của lớp được chọn trong danh sách lớp	Chuẩn bị ghi tên hồ sơ
3	Nút nút ghi	Kiểm tra hồ sơ hợp lệ. Nếu hợp lệ thì nhập hồ sơ học sinh và ghi hồ sơ học sinh. Xuất thông báo	Mã và tên phải khác rỗng

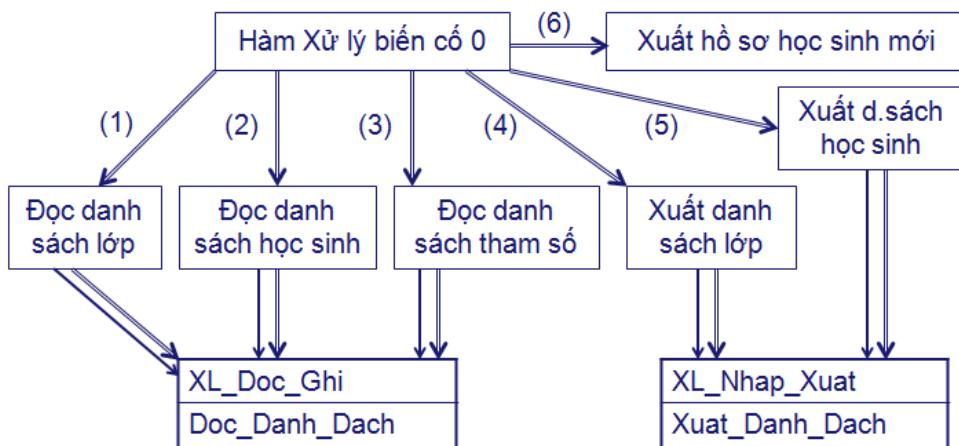
Tiếp đến, chúng ta sẽ mô tả sơ đồ chi tiết từng biến cő. Các kỹ thuật thiết kế có thể dùng là phân rã/tích hợp, tham số hóa và đối tượng hóa. Với biến cő 0 phân tích ở trên, chúng ta có thể lập sơ đồ phôi hợp như sau:



Khi phân rã hàm từ biến cő 0, chúng ta lại có sơ đồ



Chi tiết hơn nữa là:



Tùy thuộc vào kiểu thiết kế xử lý, chúng ta có các mô tả biến cō khác nhau. Vì vậy, từ một hàm xử lý chúng ta có nhiều cách để thể hiện mô hình xử lý miễn sao đảm bảo đúng chức năng của hàm xử lý đó.

#### 5.4 Thiết kế giao diện

Thiết kế giao diện là quá trình xây dựng giao diện để người dùng có thể tương tác với chương trình ứng dụng thông qua các chức năng, đồng thời thông tin từ chương trình có thể được biểu diễn hiển thị ở giao diện. Giao diện chương trình được thiết kế để sử dụng ở màn hình các thiết bị như máy tính, laptop, các thiết bị di động, hay nhiều thiết bị điện tử khác.

Giao diện giúp người dùng có thể nhìn, nghe, chạm, nói chuyện với hay đọc hiểu với chương trình. Giao diện người dùng có 2 thành phần chủ yếu: đầu vào (input) và đầu ra (output). Đầu vào là cách một người có thể đưa ra các yêu cầu thực thi của mình với máy tính. Một số thành phần đầu vào phổ biến là chuột, bàn phím, giọng nói, ngón tay (đối với

màn hình đa chạm). Đầu ra là cách một máy tính đưa các kết quả tính toán của nó và các yêu cầu với người dùng. Ngày nay, các cơ chế đầu ra phổ biến của máy tính là màn hình hiển thị, âm thanh, giọng nói. Một số đầu ra khác như cảm nhận mùi và chuyển động thiết bị tương tác với các bộ phận cơ thể con người (chẳng hạn ghế xoay khi xem video 4D) vẫn còn nhiều tiềm năng cần khám phá. Một thiết kế giao diện thích hợp phải cung cấp một sự pha trộn giữa một đầu vào thiết kế tốt và các cơ chế đầu ra thỏa mãn được nhu cầu, khả năng của con người đến mức có thể.

Với sự phát triển vượt bậc của Internet, việc giao diện thiết kế trở nên là nhu cầu thực sự cần thiết để đáp ứng xu hướng hội nhập, kết nối thông tin giữa con người với nhau trên thế giới.

#### 5.4.1 Người dùng

Thông thường, người dùng không quan tâm đến cấu trúc bên trong hệ thống mà chỉ quan tâm đến giao diện sử dụng. Họ dựa vào giao diện để đánh giá một cách chủ quan về chất lượng hệ thống. Vì vậy, nếu người dùng đánh giá giao diện sử dụng chưa phù hợp thì hoàn toàn có nguy cơ dẫn đến sự thất bại của toàn bộ dự án.

Giao diện phải được thiết kế làm sao để cho người dùng cảm thấy thoải mái, thuận tiện và sử dụng các chức năng hiệu quả. Người dùng được xem là trọng tâm của việc thiết kế. Bố cục và phong cách giao diện cũng ảnh hưởng đến người dùng theo nhiều cách khác nhau. Nếu giao diện chưa thể hiện rõ thông tin chức năng, người dùng có thể thường xuyên nhầm lẫn và gây ra lỗi khi sử dụng. Người dùng có xu hướng tẩy chay hoặc từ chối đối với các thiết kế có thẩm mỹ xấu hoặc nội dung nghèo nàn, bô cục lêch lạc. Trái lại, các giao diện được thiết kế tốt và phù hợp thì lại tăng hiệu quả làm việc của người dùng, từ đó tiết kiệm được thời gian sử dụng chương trình và mang lại nhiều hiệu quả phụ kèm theo.

Thời gian làm việc tăng thêm trên mỗi màn hình tính theo giây (s)	Thời gian tính theo năm tăng thêm nếu thực thi 4.8 triệu màn hình mỗi năm (year)
1	.7
5	3.6
10	7.1
20	14.2

Bảng 5.1 Tác động của thiết kế giao diện thiếu hiệu quả đến thời gian xử lý công việc

Trong bảng 5.1, chúng ta thấy với chỉ cần 1s tăng thêm, nếu sử dụng trên 4.8 triệu màn hình phải mất thêm 0.7 năm mỗi năm. Con số này không đáng kể với các công ty nhỏ có

nhu cầu sử dụng màn hình (máy tính) khiêm tốn, tuy nhiên với các công ty không lồ thì đây là thời gian tiêu tốn đáng phải cân nhắc.

Để thiết kế giao diện đạt nhu cầu người dùng, chúng ta phải nhận thức được sự tác động của người dùng với giao diện chương trình, đồng thời phải nắm vững các nguyên tắc thiết kế, phân loại khả năng tương tác giữa người dùng và thiết kế hiển thị giao diện, cuối cùng là thể hiện nội dung thông tin phù hợp với trình độ của các nhóm người dùng khác nhau.

Ngoài ra, khả năng trí nhớ tức thời của con người bị hạn chế, con người chỉ có thể nhớ một vài loại thông tin, vì vậy giao diện thiết kế cũng không nên biểu diễn quá nhiều dữ liệu khiến người dùng không nhớ hết và gây ra lỗi. Người thiết kế cũng nên chú ý đến những giới hạn vật lý và tinh thần của người dùng và cần hiểu rằng con người luôn có thể gây ra lỗi cho dù hệ thống có thể hiện giao diện và kèm theo hướng dẫn tốt đến đâu.

### 5.4.2 Các nguyên tắc thiết kế giao diện

Các nguyên tắc mang tính cơ bản ảnh hưởng đến thiết kế và thực thi của tất cả giao diện, bao gồm giao diện phần mềm và giao diện Web, được liệt kê dưới đây.

#### 5.4.2.1 Nguyên tắc thẩm mỹ

Chúng ta phải mang đến cho người dùng một giao diện mang tính thẩm mỹ, trong đó phải:

- Cung cấp sự tương phản có ý nghĩa giữa các thành phần trên màn hình
- Thực hiện gom nhóm các thành phần giao diện có cùng chức năng
- Canh chỉnh các thành phần giao diện và các nhóm giao diện
- Cung cấp thể hiện 3 chiều nhằm mang tính trực quan nếu có thể
- Sử dụng hiệu ứng màu sắc và đồ họa đơn giản, hợp lý

Giao diện thẩm mỹ luôn cuốn hút, gây chú ý đến người dùng, kèm theo việc truyền tải thông tin rõ ràng và kịp thời. Một thiết kế giao diện đẹp giúp người dùng sử dụng chương trình thỏa mái, dễ lôi cuốn và tương tác hiệu quả với các chức năng. Trái lại, một thiết kế xấu làm người dùng phân tâm, hoặc giảm sự chú ý, che lấp mục đích và ý nghĩa các chức năng, do đó gây bối rối với người dùng. Giao diện đóng vai trò quan trọng vì đa số mọi người tương tác với chương trình máy tính thông qua thị giác.

#### 5.4.2.2 Nguyên tắc rõ ràng

Giao diện phải trực quan, các khái niệm và ngôn ngữ phải minh bạch, đơn giản với người sử dụng. Các thành phần giao diện phải dễ hiểu, có sự liên quan đến khái niệm và chức năng của con người trong thế giới thực.

#### 5.4.2.3 Nguyên tắc tương thích

Thiết kế giao diện phải tương thích với người dùng, công việc, nhiệm vụ, và sản phẩm.

- **Tương thích người dùng:** Giao diện phải thích hợp và tương thích với các nhu cầu của người dùng. Người thiết kế phải hiểu yêu cầu và sử dụng quan điểm người dùng khi xây dựng giao diện. Một lỗi sai chung của một nhà thiết kế là xem các quan điểm người dùng đều như nhau hoặc luôn áp đặt quan điểm thiết kế của mình cho tất cả người dùng. Ví dụ, một người thiết kế thích nền màu đỏ, chữ màu xanh, tuy nhiên đa số mọi người đều lại thích thiết kế đơn giản hơn như nền màu trắng, chữ màu xanh chẳng hạn. Để thực hiện việc tương thích với người dùng, các nhà thiết kế nên khảo sát, thu thập ý kiến người để đánh giá, lựa chọn giao diện thiết kế phù hợp.
- **Tương thích công việc, nhiệm vụ:** Một hệ thống phải phù hợp với nhiệm vụ mà một người dùng cần phải làm. Cấu trúc và luồng chức năng phải cho phép dễ dàng chuyển tiếp giữa các nhiệm vụ. Tuyệt đối không ép buộc người dùng theo sử dụng nhiều ứng dụng hoặc nhiều màn hình để thực thi các nhiệm vụ khi sử dụng chương trình phần mềm. Ví dụ, một chương trình cho phép nhập thông tin sinh viên kèm theo là ảnh của sinh viên (kích thước 200x300 pixels), tuy nhiên các ảnh đầu vào đều không đúng kích thước. Do đó, người dùng phải dùng một chương trình xử lý ảnh khác để xử lý ảnh trước khi nhập thông tin sinh viên. Đây là điều không nên, chương trình phải tích hợp thêm chức năng điều chỉnh kích ảnh cho phù hợp.
- **Tương thích sản phẩm:** Khi thiết kế giao diện chương trình thì phải xây dựng chương trình đó tương thích với các loại máy tính hay phiên bản hệ điều hành khác nhau. Nếu giao diện chương trình được hiển thị trên các dòng máy không tương thích, giao diện có thể bị lệch, méo, một số chức năng bị thiếu dẫn đến việc không sử dụng được chương trình, giảm hiệu quả làm việc.

#### 5.4.2.4 Nguyên tắc dễ hiểu

Giao diện thiết kế phải dễ hiểu với người dùng theo khía cạnh:

- Nên nhìn cái gì
- Nên làm gì

- Khi nào làm một việc gì đó
- Nơi nào để làm việc đó
- Tại sao phải làm việc đó
- Cách để làm việc đó

Một hệ thống phải dễ hiểu, theo trật tự rõ ràng, có ý nghĩa. Các bước hoàn thành một nhiệm vụ phải đơn giản. Các chú thích phải được diễn giải ngắn gọn.

#### 5.4.2.5 Nguyên tắc cấu hình

Giao diện thiết kế phải cho phép người dùng tùy ý cá nhân hóa và tùy biến cấu hình để tăng khả năng kiểm soát chương trình, thúc đẩy vai trò cá nhân trong việc hiểu, cho phép việc sử dụng giao diện dựa mức độ kinh nghiệm của người dùng. Điều này mang đến sự thỏa mãn tốt hơn với người dùng.

#### 5.4.2.6 Nguyên tắc bền vững

Một giao diện phải có cái nhìn, hoạt động nhất quán xuyên suốt để người dùng dễ sử dụng và nắm bắt. Một số thành phần nên giống nhau như phong cách giao diện (màu sắc, font chữ, ...), tính năng và cách hoạt động tương tự nhau. Cùng một chức năng phải luôn luôn cho ra một kết quả giống nhau. Các tính năng và vị trí các thành phần giao diện phải không thay đổi. Các giao diện thiết kế không nhất quán sẽ gây ức chế, bối rối lớn với người dùng.

Ví dụ, với một chương trình soạn thảo văn bản thì các chức năng soạn thảo đơn giản như thanh công cụ dưới đây (Hình 5.1) phải luôn ở trên đầu chương trình và không nên bị thay đổi vị trí, màu sắc hay biểu tượng ở các module khác trong chương trình.



Hình 5.1 Thanh công cụ soạn thảo văn bản

#### 5.4.2.7 Nguyên tắc kiểm soát

Người dùng phải kiểm soát được các tương tác với chương trình thông qua:

- Các chức năng phải cho ra kết quả từ các yêu cầu rõ ràng
- Các chức năng có thời gian thực thi nhanh
- Các chức năng phải được ngắn hay trì hoãn không thực hiện trong trường hợp người dùng muốn
- Người dùng không nên bị dừng tương tác bởi các lỗi sinh ra trong hệ thống

#### 5.4.2.8 Nguyên tắc hiệu quả

Người thiết kế cũng phải chú ý đến số lượng các chuyển động mắt và tay của người dùng. Các chuyển động này không nên được lãng phí. Người dùng hầu hết sử dụng thị giác để nhận biết thông tin, vì vậy các thành phần giao diện phải có tính dự đoán, hiển nhiên và ngắn gọn. Việc chuyển tiếp thủ công giữa các điều khiển hệ thống hay màn hình phải ngắn nhất có thể. Tránh việc sử dụng thường xuyên chuột, bàn phím chỉ để thực hiện thao tác.

Người thiết kế phải luôn dự đoán nhu cầu người dùng. Ở mỗi bước trong 1 quy trình, phải trình bày tất cả các thông tin và công cụ cần thiết để hoàn thành quy trình đó. Hạn chế việc đòi hỏi người dùng phải tìm kiếm, thu thập thông tin hay công cụ cần thiết. Ví dụ, khi người dùng nhập vào ngày, tháng, năm sinh thì chương trình phải tự tính tuổi của người đó, tránh việc yêu cầu người dùng phải nhập vào tuổi.

#### 5.4.2.9 Nguyên tắc thân thiện

Chúng ta phải thể hiện trên giao diện các nội dung, khái niệm thông qua ngôn ngữ quen thuộc với người dùng, giữ giao diện tự nhiên, bắt chước các hành vi người dùng. Các khái niệm dễ hiểu sẽ làm người dùng làm quen hệ thống nhanh hơn.

#### 5.4.2.10 Nguyên lý mềm dẻo

Một hệ thống phải mềm dẻo thể hiện các nhu cầu khác nhau của người dùng, cho phép một mức độ và dạng nền tảng dựa trên:

- Kỹ năng và kiến thức của người dùng
- Kinh nghiệm người dùng
- Sở thích người dùng
- Thói quen người dùng
- Các điều kiện hiện tại

#### 5.4.2.11 Nguyên tắc dự đoán

Giao diện chương trình phải có tính dự đoán tự nhiên dựa trên từng chức năng để:

- Cung cấp các thành phần giao diện nhận biết được và rõ ràng
- Cung cấp các gợi ý cho kết quả hoặc dữ liệu mà người dùng mong muốn

Các nhiệm vụ, giao diện và chuyển động thông qua hệ thống phải được dự đoán dựa trên kinh nghiệm và kiến thức người dùng trước đó. Tất cả tương tác nên đưa đến kết quả

mà người dùng mong muốn. Việc dự đoán sẽ giảm thiểu lỗi, cho phép thực thi các công việc nhanh hơn. Tất cả các gợi ý phải thống nhất giống nhau và hoàn chỉnh.

#### 5.4.2.12 Nguyên tắc phục hồi

Hệ thống phải có khả năng ra lệnh bãi bỏ hay đảo ngược các thao tác, lệnh đã thực thi, quay về thời điểm nào đó khi lỗi phát sinh. Tương tự, ở giao diện thiết kế, người dùng phải có chức năng lưu trữ thông tin tự động lúc đang làm việc khi gặp sự cố. Việc phục hồi giúp người dùng tránh được lỗi gấp trước đó hoặc lấy lại được thông tin bị mất, giảm thiểu thời gian và công sức phải thực hiện công việc từ đầu.

### 5.4.3 Biểu diễn bố cục giao diện

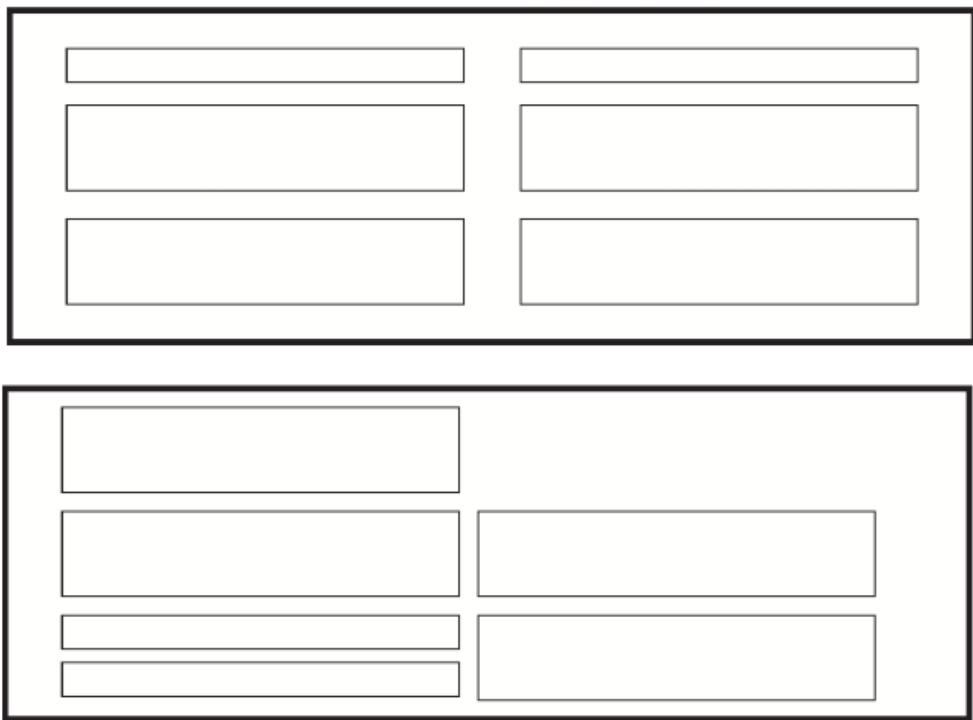
Bố cục giao diện thường được biểu diễn theo các tính chất sau:

- Tính cân bằng
- Tính đối xứng
- Tính đều đặn
- Tính dự đoán
- Tính liên tục
- Tính kinh tế
- Tính thống nhất
- Tính cân xứng theo tỉ lệ
- Tính đơn giản
- Tính gom nhóm

Các nghiên cứu về thị giác chỉ ra rằng khi tiếp xúc với giao diện, con người bị ảnh hưởng bởi tính cân bằng, độ đậm nhạt của tiêu đề, đồ họa, và văn bản. Cơ chế nhận thức của con người luôn tìm kiếm những thứ có trật tự và ý nghĩa, và có gắng áp đặt cấu trúc giao diện mặc định trong tư duy khi đối chiếu các giao diện có bố cục lộn xộn. Một chương trình với giao diện có cấu trúc bố cục tốt luôn giúp người dùng làm quen nhanh hơn và sử dụng cảm thấy thoải mái hơn.

#### 5.4.3.1 Tính cân bằng

Bố cục cân bằng có các thành phần giao diện tương đối bằng nhau theo các phía trái, phải, trên, dưới. Hình sau là giao diện cân bằng, các thành phần giao diện đối xứng với nhau qua trọng tâm ở giữa.



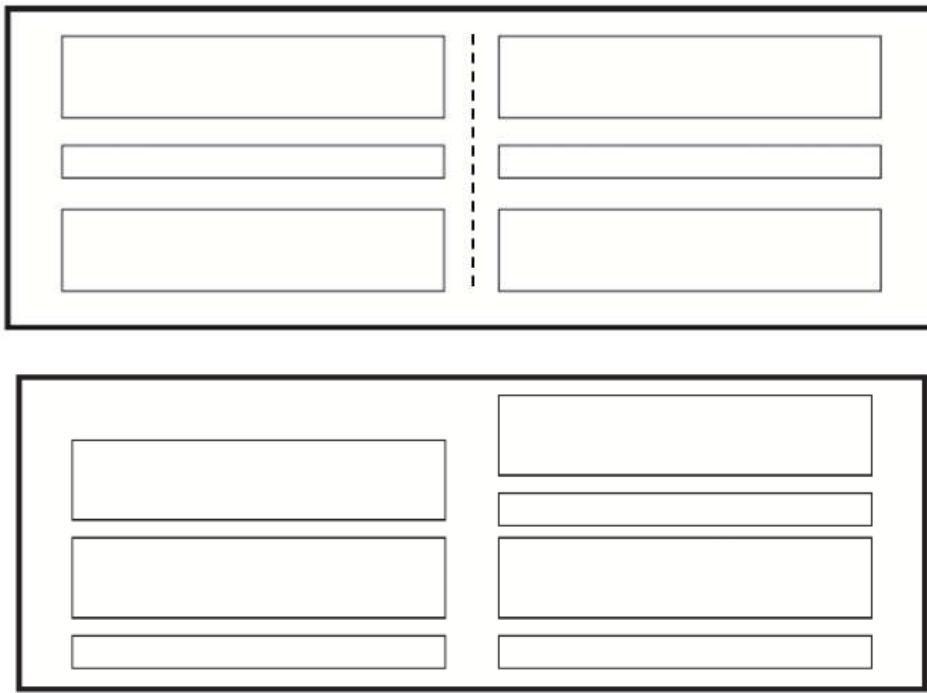
Hình 5.2 Giao diện cân bằng (trên) và giao diện chưa cân bằng (dưới)

Hình dưới là giao diện không cân bằng với các số lượng thành phần bên trái là 4, bên phải là 2, hơn nữa các thành phần 2 bên không đối xứng nhau từng đôi một nếu xét theo chiều ngang.

Người dùng sẽ cảm thấy khó chịu với giao diện chưa cân bằng, tương tự như khi họ nhìn một bức tranh bị treo xiên 1 bên trên tường. Thông thường, giao diện không nhất thiết phải đảm bảo tính cân bằng tuyệt đối mà chỉ nên tương đối, với 1 số sự sai khác nhỏ như hình dạng, vị trí, màu sắc, kích thước.

#### 5.4.3.2 Tính đối xứng

Nếu kẻ một đường dọc nét đứt ở giữa màn hình làm đường trực thì nếu các thành phần giao diện phải đối xứng nhau thông qua đường này thì gọi giao diện này là giao diện đối xứng.

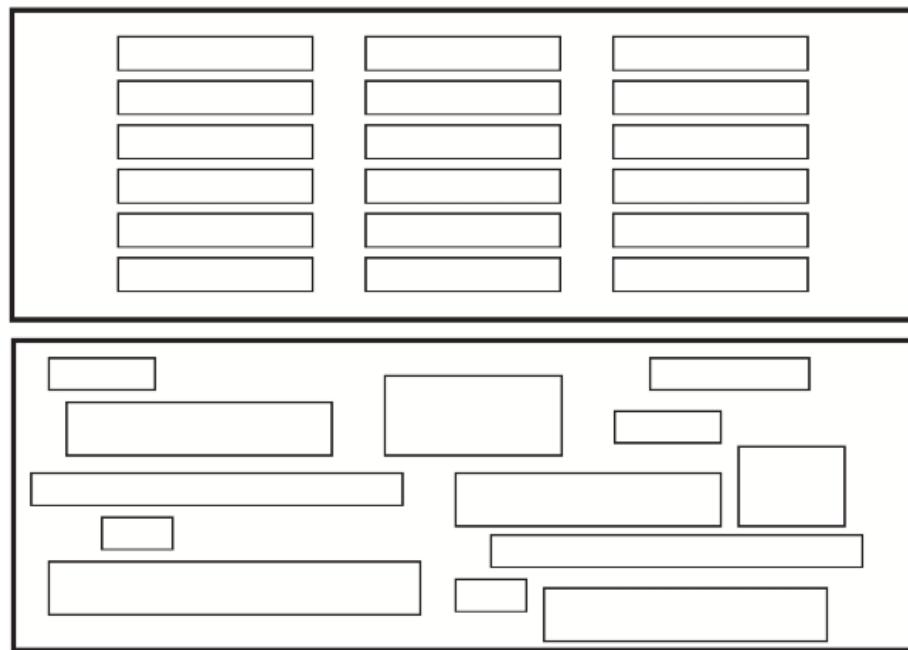


*Hình 5.3 Giao diện đối xứng (trên) và giao diện không đối xứng (dưới)*

Tuy nhiên, chúng ta có giao diện không đối xứng khi các thành phần không đối xứng nhau từng đôi một.

#### **5.4.3.3 Tính đều đặn (regularity)**

Để tạo một bố cục đều đặn, nhà thiết kế phải thiết lập các tiêu chuẩn và các điểm canh lề ngang, dọc đồng nhất và sử dụng các thành phần kích thước, hình dạng, màu sắc và khoảng trống tương tự nhau.

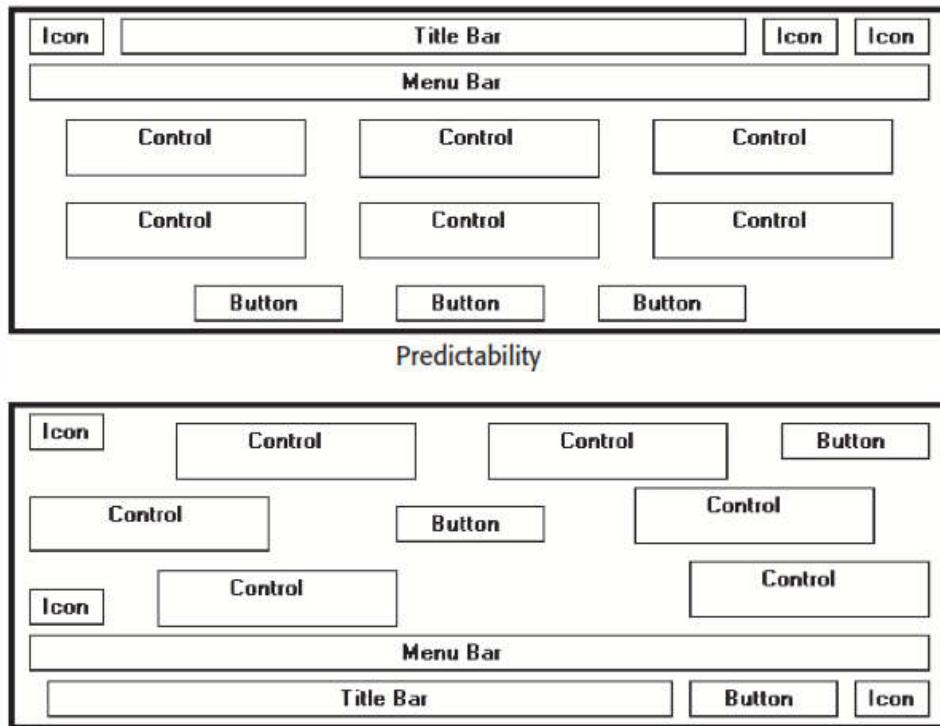


Hình 5.4 Giao diện đều đặn (trên) và giao diện không đều đặn (dưới)

Trong hình, giao diện đều đặn có các thành phần đều nhau theo từng vị trí, được chia làm 3 cột, cùng sử dụng ô vuông giống nhau và khoảng cách giữa các đối tượng ô vuông cũng cân bằng. Trái lại, hình dưới là giao diện không đều đặn, với kích thước các thành phần khác nhau, được bố trí lộn xộn.

#### 5.4.3.4 Tính dự đoán

Giao diện phải được tạo theo trật tự hoặc thứ tự sắp xếp nhất định.



Hình 5.5 Giao diện dự đoán (trên) và giao diện tự phát (dưới)

Trong hình, giao diện dự đoán được sắp xếp theo bố cục và kế hoạch cho trước. Khi xem một màn hình có thứ tự, người dùng cũng có thể dự đoán các màn hình khác cũng có thứ tự tương tự. Tương tự, nếu chỉ cần xem 1 phần giao diện, người dùng cũng có thể dự đoán được toàn bộ giao diện. Giao diện tự phát là giao diện được thiết kế không theo kế hoạch vì vậy bố cục không theo thứ tự.

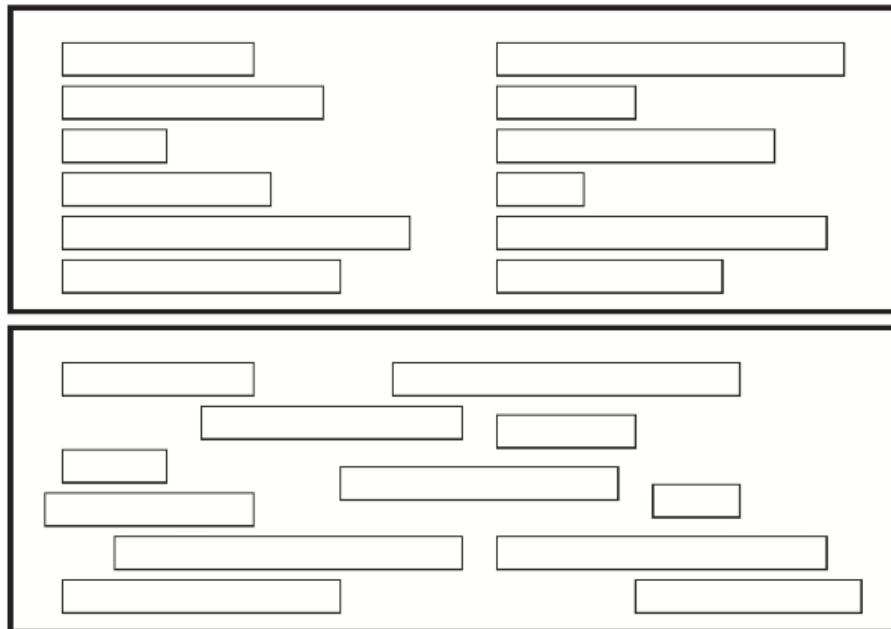
#### 5.4.3.5 Tính liên tục

Giao diện có tính liên tục là giao diện chứa các thành phần được sắp xếp giúp người dùng có thể theo dõi theo một trật tự rõ ràng, hợp lý và hiệu quả. Thị giác người sử dụng thường bị cuốn hút bởi:

- Các đối tượng nổi bật về ánh sáng so với đối tượng ít nổi bật hơn
- Các thành phần riêng biệt so với các nhóm thành phần
- Hiệu ứng đồ họa hơn so với văn bản
- Màu sắc hơn so với trắng, đen
- Vùng tối hơn so với vùng sáng
- Thành phần có kích thước lớn hơn so với thành phần có kích thước nhỏ

- Hình dạng bất thường so với hình dạng bình thường

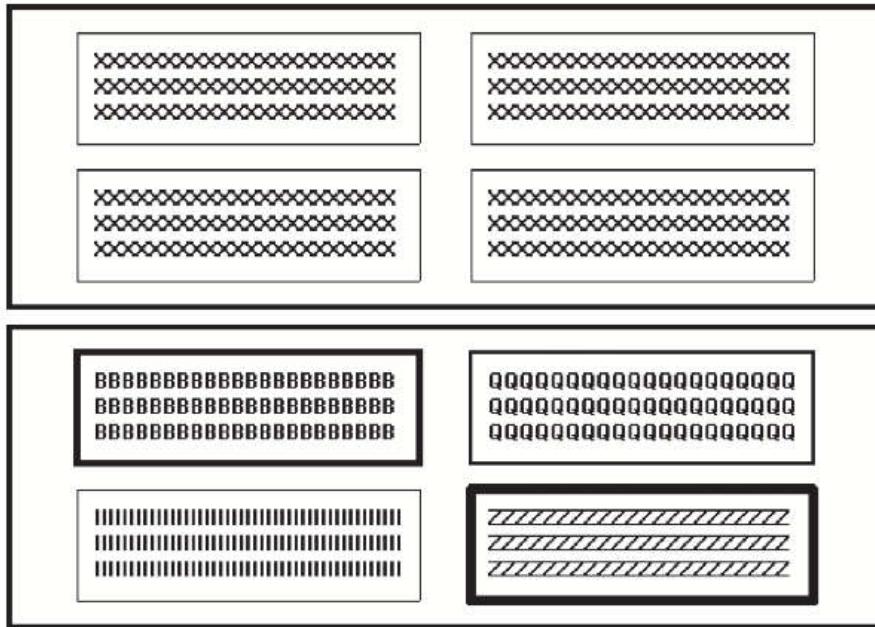
Trong hình phía dưới, giao diện liên tục hướng thị giác người dùng theo trật tự từ trên xuống dưới và từ trái sang phải khi thực hiện các thao tác tương tác. Ngược lại, giao diện ngẫu nhiên làm người dùng bối rối không biết phải thao tác theo thứ tự như thế nào.



Hình 5.6 Giao diện có tính liên tục (trên) và giao diện ngẫu nhiên (dưới)

#### 5.4.3.6 Tính kinh tế

Giao diện phải mang tính kinh tế khi sử dụng phong cách, kỹ thuật hiển thị và màu sắc ít nhất có thể.

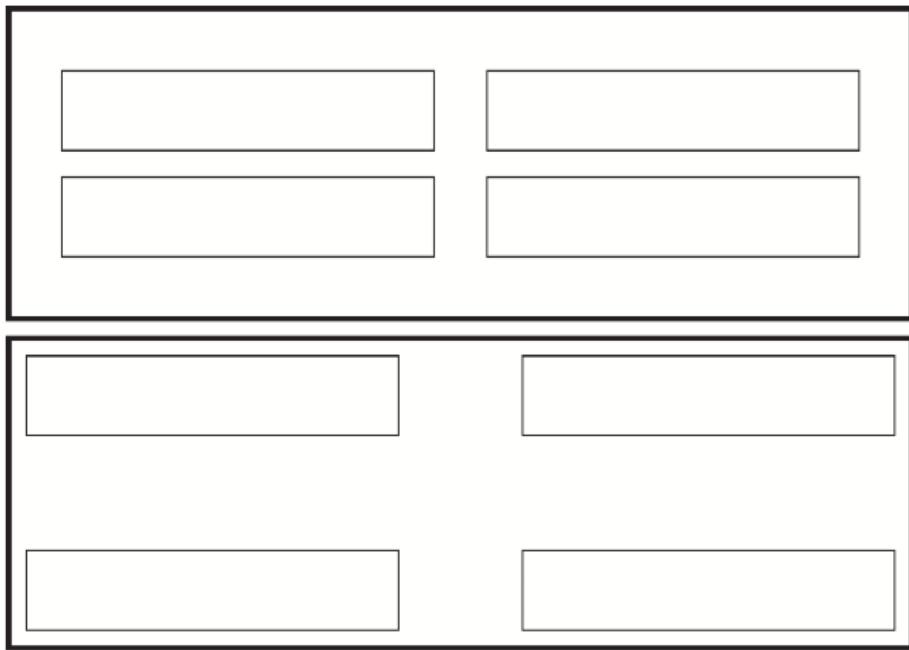


Hình 5.7 Giao diện có tính kinh tế (trên) và giao diện phúc tạp (dưới)

Trong hình trên, giao diện kinh tế sử dụng phong cách đồng nhất, đơn giản để hiển thị các thành phần dữ liệu. Trái lại, giao diện phức tạp sử dụng nhiều phong cách hiển thị khác nhau (độ đậm nhạt đường viền, văn bản). Do đó, nếu thiết kế giao diện phức tạp, chi phí bỏ ra cho các nhân viên thiết kế có thể tăng thêm mà chương trình phần mềm vẫn không đạt hiệu quả nhưng mong muốn.

#### 5.4.3.7 Tính thống nhất

Giao diện mang tính thống nhất khi sử dụng thông tin có kích thước, màu sắc, hình dạng tương tự nhau và ít có khoảng trống giữa các thành phần hơn so với khoảng trống với các lề màn hình.



Hình 5.8 Giao diện có tính thống nhất (trên) và giao diện phân mảnh (dưới)

Trong hình, giao diện có tính thống nhất có cùng đối tượng thành phần là các ô chữ nhật với kích thước giống nhau, khoảng cách giữa các thành phần nhỏ hơn so với khoảng cách giữa các thành phần với lề màn hình. Trái lại, giao diện phân mảnh thì khoảng cách giữa các thành phần rộng hơn rất nhiều với khoảng cách giữa các thành phần với lề màn hình.

#### 5.4.3.8 Tính cân xứng theo tỷ lệ

Khi thiết kế giao diện, chúng ta có thể chọn một số kích thước chuẩn cho các thành phần giao diện. Một số tỉ lệ chuẩn được Marcus định nghĩa trong nghiên cứu năm 1992 của ông là:

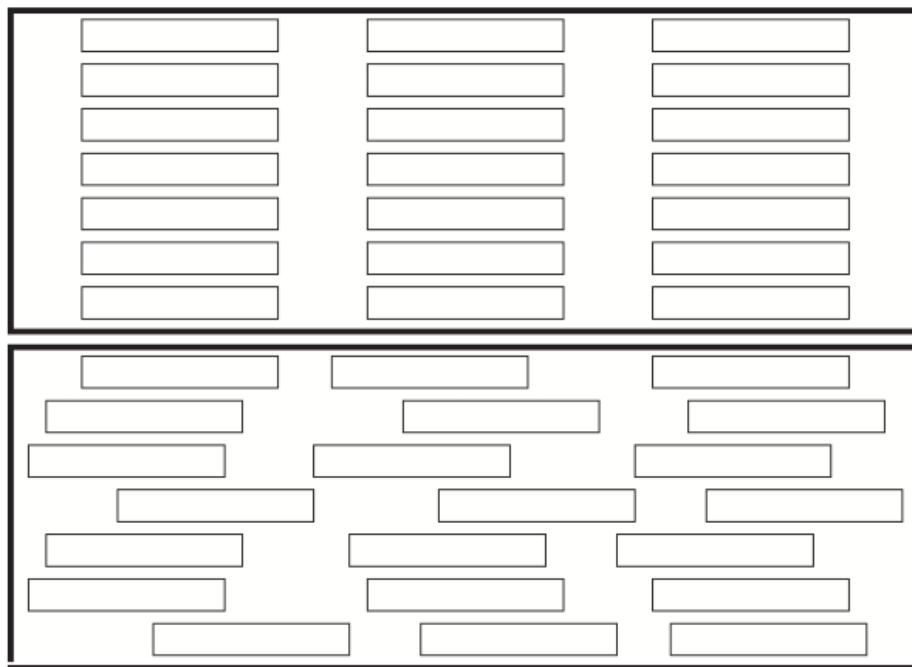
- Hình vuông (1:1)
- Tỉ lệ căn bậc 2 (1:1.414)
- Hình chữ nhật vàng (1:1.618)
- Tỉ lệ căn bậc 3 (1:1.732)
- Tỉ lệ 1:2



Hình 5.9 Các tỉ lệ chuẩn cho các thành phần giao diện

#### 5.4.3.9 Tính đơn giản

Người thiết kế phải tối ưu số lượng thành phần sử dụng trên giao diện và giảm thiểu các điểm canh lề, đồng thời xây dựng một chuẩn lưới canh lề dọc, ngang để giao diện đều, đơn giản, rõ ràng.



Hình 5.10 Giao diện đơn giản (trên) và giao diện phức tạp (dưới)

Trong hình, giao diện đơn giản phía trên với các thành phần được canh lề theo ma trận lưới đều nhau, còn giao diện phức tạp lại chứa nhiều thành phần có điểm canh lề so le nhau.

#### 5.4.3.10 Tính gom nhóm

Cuối cùng, giao diện thiết kế phải có tính gom nhóm, tức là gom các thành phần có cùng chức năng và ý nghĩa vào cùng 1 nhóm để người dùng dễ dàng nhận biết và tương tác tốt hơn với giao diện. Nhà thiết kế sử dụng các đường viền để khoanh vùng các nhóm và

giữa các nhóm khác nhau đều có 1 khoảng cách nhất định. Trong hình dưới, các thành phần có cùng ý nghĩa được gom thành các nhóm, mỗi nhóm có đường viền bao quanh, giữa các nhóm có khoảng trắng phù hợp, cân xứng.

<b>TIFF File format</b>	<b>Image size</b>
Version # 42	Horizontal 1888 6.3
Byte order 11	Vertical 1656 5.5
NewSubFile Type 1	Units Pixel Inch
<b>Spatial Configuration</b>	
Samples per pixel 1	<b>Image resolution</b>
Bits per sample 1	Horizontal res. 300
Planar config. 1	Vertical res. 300
	Units dpi

Hình 5.11 Giao diện có tính gom nhóm

#### 5.4.4 Biểu diễn thông tin

Thông tin từ hệ thống được hiển thị tới người dùng sử dụng thông qua giao diện thiết kế. Thông tin khác nhau có kiểu dữ liệu khác nhau vì vậy biểu diễn thông tin có thể chuyển thành nhiều cách hiển thị như dạng đồ họa, âm thanh. Chúng ta có thể chia thông tin được thể hiện bằng 2 loại:

- Thông tin tĩnh: là dạng được khởi tạo ở đầu của mỗi phiên làm việc trong chương trình. Những thông tin này không thay đổi trong suốt phiên làm việc đó và có thể là ở dạng số hoặc dạng văn bản.
- Thông tin động là những thông tin thay đổi liên tục trong cả phiên sử dụng với sự quan sát của người dùng.

Một số nhân tố có tác động đến việc hiển thị thông tin như sau:

- Sở thích của người sử dụng về việc hiển thị thông tin, một phần thông tin hay quan hệ dữ liệu.
- Tốc độ thay đổi dữ liệu thông tin nhanh hay chậm
- Các thao tác cần làm của người dùng để thay đổi dữ liệu
- Thể hiện thông tin ở các kiểu dữ liệu khác nhau
- Màu sắc trong thiết kế

Khi thể hiện bằng thông tin bằng màu sắc, chúng ta cũng nên chú ý các nguyên tắc:

- Thay đổi màu khi thay đổi trạng thái của hệ thống

- Giới hạn số lượng màu được sử dụng và không nên lạm dụng việc sử dụng màu.
- Sử dụng màu để hỗ trợ cho những nhiệm vụ mà người dùng đang có găng thực hiện.
- Sử dụng màu một cách thống nhất và cẩn thận.
- Cẩn thận khi sử dụng các cặp màu.

Các thông tin là dạng thông báo lỗi phải có cấu trúc, ngắn gọn, xúc tích và thống nhất cho tất cả các giao diện trong hệ thống. Nếu thông báo lỗi không rõ ràng có thể làm cho người dùng chán nản dẫn đến việc từ bỏ việc sử dụng chương trình. Người thiết kế phải đánh giá đúng kỹ năng và kinh nghiệm của người dùng khi thiết kế thông báo lỗi.

#### 5.4.5 Quy trình thiết kế giao diện

Người thiết kế phải lập ra một số danh sách như danh sách các biến cỗ và danh sách các thành phần giao diện để hiểu rõ cần phải thiết kế giao diện như thế nào.

Danh sách các biến cỗ

STT	Điều kiện kích hoạt	Xử lý	Ghi chú
...	...	...	...

Danh sách các thành phần giao diện

STT	Tên	Kiểu	Ý nghĩa	Miền giá trị	Giá trị mặc định	Ghi chú
...	...	...	...	...	...	...

**Ví dụ:** thiết kế chức năng tiếp nhận học sinh mới của phần mềm Quản Lý Học Sinh như sau:

Tiếp nhận học sinh	
Họ tên:.....	Giới tính:.....
Ngay sinh:.....	Địa chỉ:.....
Lớp:.....	
<b>Qui định:</b> Họ tên phải có. Tuổi từ 15-20. Trường có 20 lớp và 3 khối. Khối 10 có 8 lớp, Khối 11 có 7 lớp, Khối 12 có 5 lớp	

Đầu tiên chúng ta phải lập danh sách các thành phần giao diện như sau:

STT	Tên	Kiểu	Ý nghĩa	Miền giá trị	Giá trị mặc định	Ghi chú
1	Lb_Tieu_de	Label	Tiêu đề màn hình			
2	Lb_Hoten	Label	Tiêu đề họ tên			
3	Txt_Hoten	TextBox	Text box nhập họ tên			
4	Ch_Phai	Checkbox				
5	...					
6	...					

Sau đó, tiến hành xây dựng giao diện nháp như sau:

**Tiếp nhận học sinh**

Họ tên	<input type="text"/>	Nam <input checked="" type="checkbox"/>
Ngày sinh	<input type="text"/>	Lớp <input type="text"/>
Địa chỉ	<input type="text"/>	
<b>Ghi</b>		

**Danh sách học sinh đã tiếp nhận**

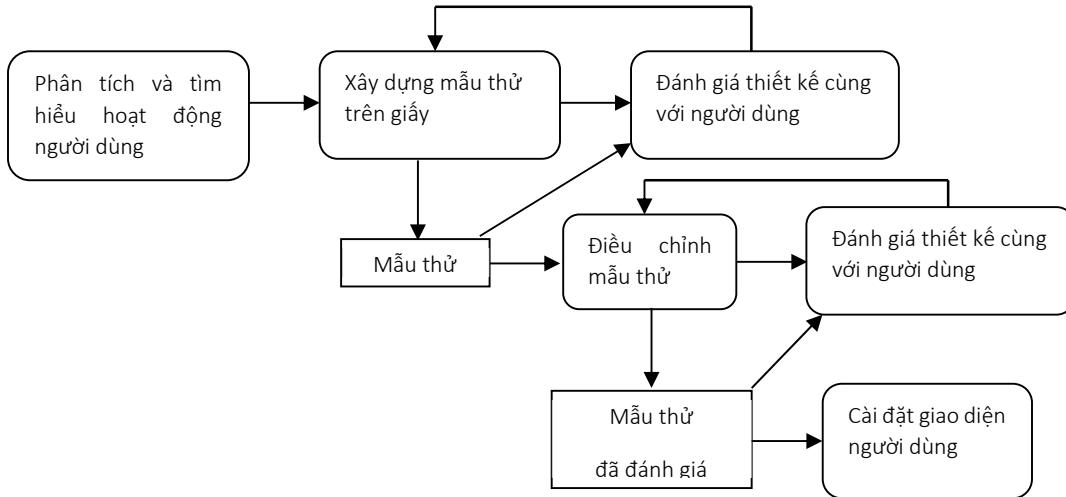
STT	Mã HS	Tên HS	Giới tính	Ngày sinh
...	...	...	...	...

#### 5.4.6 Khảo sát người dùng và phân tích, đánh giá giao diện

Để thiết kế giao diện hiệu quả, thông thường thì phải cần một quy trình lặp lại với sự cộng tác giữa người dùng và người thiết kế. Quy trình này gồm các hoạt động chính như:

- Khảo sát người sử dụng: tìm hiểu các yêu cầu của người sử dụng mong muốn đối với chương trình.
- Lập mẫu thử hệ thống: xây dựng một tập các ví dụ thử để thử nghiệm đánh giá hệ thống.
- Đánh giá giao diện: thử nghiệm các mẫu thử cùng với người sử dụng.

Chúng ta phải tiến hành khảo sát, lấy ý kiến người dùng để hiểu rõ cần phải làm gì với giao diện. Việc khảo sát phải được mô tả bằng thuật ngữ đơn giản, dễ hiểu để người dùng có thể hiểu và tiến hành khảo sát thuận lợi. Mô hình chung về quá trình khảo sát người, phân tích và đánh giá giao diện như sau:



Hình 5.12 Mô hình khảo sát, phân tích, đánh giá giao diện

Trong hình trên, bước đầu tiên là chúng ta phải tiến hành việc phân tích và tìm hiểu hoạt động người dùng bằng cách xây dựng các mẫu thử trên giấy hoặc bằng các thể hiện khác như giao diện demo. Tiếp đến, chúng ta phải dùng mẫu thử này thử nghiệm với người dùng. Người dùng sẽ góp ý, đánh giá mẫu thử để khâu xây dựng cho ra mẫu thử tốt hơn. Quá trình này cứ lặp đi lặp lại. Trong trường hợp mẫu thử có điều chỉnh thì tương tự chúng ta cũng đưa cho người dùng đánh giá và điều chỉnh mẫu thử mới điều chỉnh. Các quá trình này cứ lặp đi lặp lại cho đến khi nào cho ra mẫu thử tốt nhất thì chúng ta sẽ dùng mẫu thử đó cài đặt giao diện người dùng. Quy trình này tương tự như việc xuất bản phần mềm, nhà sản xuất đưa ra phiên bản Beta để thử nghiệm và lấy ý kiến khách hàng trong khoảng thời gian nào đó. Sau đó, nhà sản xuất gom ý kiến người dùng và chuyên gia để nâng cấp phần mềm và cho ra bản chính thức.

Để tăng chất lượng thiết kế, ngoài việc lấy ý kiến khách hàng, chúng ta cũng nên lấy ý kiến chuyên gia trong lĩnh vực thiết kế phần mềm để có cái nhìn kỹ thuật sâu sắc và phong phú hơn.

## 5.5 Xây dựng sơ đồ lớp

## **5.6 Xây dựng sơ đồ trạng thái**

## **5.7 Kết chương**

### **Tài liệu tham khảo**

1. Slide bài giảng Công nghệ Phần mềm, Đại học Công nghệ Thành phố Hồ Chí Minh
2. Wilbert O. Galitz, “*The Essential Guide to User Interface Design*”, John Wiley & Sons, Inc., 2002.

## CHƯƠNG 6. XÂY DỰNG PHẦN MỀM

### 6.1 Kiến trúc phần mềm

#### 6.1.1 Các kiểu kiến trúc phần mềm

Kiểu kiến trúc phần mềm là tập hợp các nguyên tắc để hình thành một framework (khung) chung cho các hệ thống phần mềm. Kiểu kiến trúc phần mềm giúp cải tiến các thành phần cấu trúc của phần mềm bằng việc cung cấp các giải pháp để giải quyết các vấn đề xảy ra lặp lại nhiều lần trong quá trình phát triển phần mềm. Hiểu được các kiểu kiến trúc phần mềm giúp các thành viên trong nhóm phát triển có một ngôn ngữ chung trong việc thảo luận các nền tảng kỹ thuật áp dụng trong dự án của mình.

Danh mục	Kiểu kiến trúc
Truyền thông - Communication	Kiến trúc hướng dịch vụ (SOA), Message Bus
Triển khai - Deployment	Client/Server, N-Tier, 3-Tier
Lĩnh vực - Domain	Domain Driven Design
Cấu trúc - Structure	Kiến trúc dựa trên các thành phần, Kiến trúc hướng đối tượng, kiến trúc hướng dịch vụ

Bảng 6.1 Phân loại danh mục các kiểu kiến trúc phần mềm

Kiểu kiến trúc	Mô tả
Client/Server	Chia hệ thống làm 2 phần: Client và Server, ứng dụng tại client sẽ gửi các yêu cầu đến Server. Trong nhiều trường hợp, Server là nơi tập trung cơ sở dữ liệu với ứng dụng logic như Stored Procedures
Kiến trúc dựa trên các thành phần	Phân tách các thiết kế ứng dụng thành các chức năng có thể tái sử dụng hoặc các thành phần logic

(Component-Based)	với các thành phần giao tiếp được xác định rõ ràng
Domain Driven Design	Có các đặc điểm của mô hình hướng đối tượng nhưng tập trung vào mô hình một lĩnh vực kinh doanh và xác định đối tượng kinh doanh dựa trên các thực thể trong các lĩnh vực kinh doanh.
Kiến trúc phân tầng	Phân các thành phần liên quan của các ứng dụng thành các nhóm xếp chồng lên nhau (tầng – lớp)
Message Bus	Một kiểu kiến trúc quy định việc sử dụng một hệ thống phần mềm có thể nhận và gửi tin nhắn sử dụng một hoặc nhiều các kênh giao tiếp, vì vậy mà các ứng dụng có thể tương tác mà không cần phải biết chi tiết cụ thể về những cái khác
N-Tier / 3-Tier	Chia các tính năng của ứng dụng thành các phần theo cùng một cách như phân tầng, tuy nhiên các phần này được đặt trên các thiết bị vật lý riêng biệt
Kiến trúc hướng đối tượng	Mô hình thiết kế dựa trên việc phân chia ứng dụng các đối tượng, các thực thể có thể tái sử dụng, những đối tượng này chứa dữ liệu và các hành vi liên quan đến đối tượng
Kiến trúc hướng dịch vụ	Sử dụng trong các ứng dụng được sử dụng như một dịch vụ cho các hệ thống khác, thông thường sẽ thông qua môi trường web.

Bảng 6.2 Mô tả ngắn về các kiểu phần mềm

### 6.1.2 Sự kết hợp các kiểu kiến trúc

Thông thường một phần mềm không bao giờ giới hạn trong một kiểu kiến trúc đơn lẻ, ta có thể kết hợp nhiều kiểu thiết kế khác nhau trong cùng một hệ thống. Ví dụ có thể có một thiết kế hướng dịch vụ bao gồm các dịch vụ sử dụng các kiến trúc phân tầng và kiểu kiến trúc hướng đối tượng để tạo và thực thi các đối tượng trong ứng dụng.

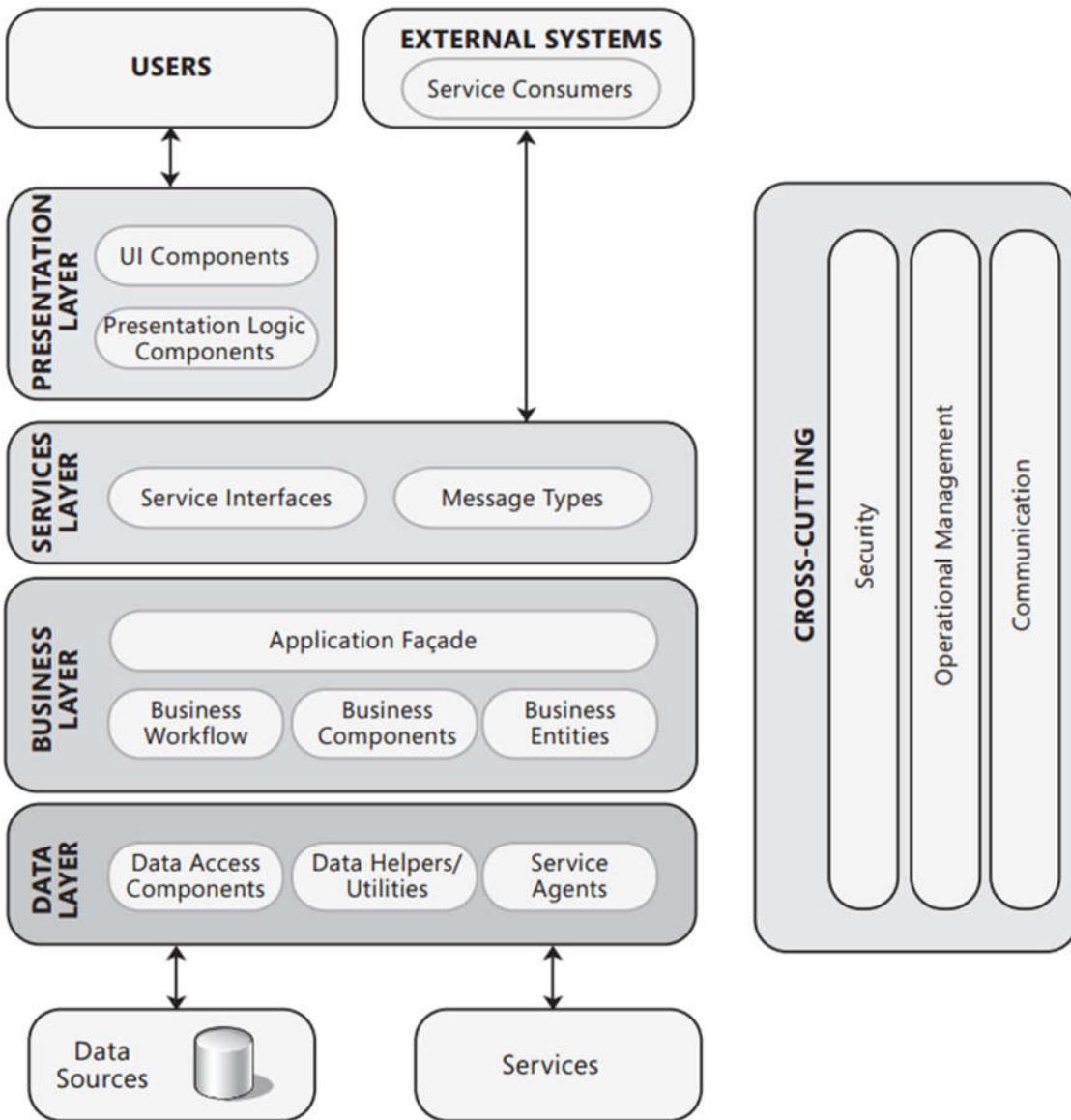
Việc kết hợp các kiểu kiến trúc sẽ hữu ích nếu ta xây dựng một ứng dụng truy cập trên môi trường web. Ở đây ta có thể phân chia các thành phần liên quan bằng việc sử dụng kiểu kiến trúc phân tầng. Tức là ta sẽ có các lớp Presentation Logic, lớp Business Logic và lớp Data Access Logic. Trong khi đó, với yêu cầu bảo mật của ứng dụng a có thể triển khai ứng dụng với kiểu kiến trúc 3-Tier hoặc N-Tier. Trong đó, tầng Presentation Tier có thể được triển khai cho mạng nội bộ bên trong doanh nghiệp, bên trong tầng này ta có thể sử dụng các kiến trúc như MVC( Model-View-Controller - Nằm trong kiểu kiến trúc phân tầng) cho mô hình tương tác. Hoặc ta cũng có thể sử dụng kiểu kiến trúc hướng dịch vụ và thực thi các giao tiếp dựa trên các thông điệp giữa Web Server và Application Server.

Đối với một ứng dụng desktop được lựa chọn xây dựng, có thể có yêu cầu của client gửi đến một ứng dụng trên server. Trong trường hợp này, ta có thể phát triển ứng dụng dựa trên việc lựa chọn kiểu kiến trúc client/server và kiểu kiến trúc dựa trên các thành phần để phân tách các phần của ứng dụng một cách độc lập để đáp ứng cho các yêu cầu từ các giao diện người dùng.Thêm vào đó việc sử dụng kiểu kiến trúc hướng đối tượng cho các thành phần sẽ nâng cao khả năng tái sử dụng, kiểm tra lỗi và tính mềm dẻo cho ứng dụng.

Nhiều yếu tố ảnh hưởng tới việc lựa chọn kiểu kiến trúc sử dụng. Các yếu tố này bao gồm khả năng của doanh nghiệp cho việc thiết kế và thực thi; Khả năng và kinh nghiệm của người phát triển, về cơ sở hạ tầng và những hạn chế của tổ chức etc.

### 6.1.3 Phân tầng ứng dụng

Không phân biệt mô hình chi tiết các kiểu kiến trúc ứng dụng đang được xây dựng với các thiết kế giao diện người dùng hoặc các dịch vụ sử dụng, ta có thể phân tách thiết kế phần mềm thành các nhóm logic cho các thành phần của ứng dụng. Các nhóm logic này được gọi là các tầng. Các tầng này giúp phân biệt giữa các nhóm chức năng khác nhau của các thành phần, giúp tạo ra một thiết kế mềm dẻo, linh hoạt hơn hỗ trợ việc tái sử dụng các thành phần trong việc xây dựng phần mềm. Mỗi tầng logic chứa một số nhóm các chức năng nhỏ hơn nhằm chuyên biệt hóa vai trò của nó trong kiến trúc chung. Cũng như hỗ trợ phân chia công việc cụ thể cho từng thành viên trong quá trình xây dựng ứng dụng.



Hình 6.1 Mô hình phân tầng logic các thành phần của ứng dụng

#### 6.1.3.1 Tầng trình bày (Presentation Layer)

Sử dụng các tính năng cung cấp bởi tầng business để xây dựng giao diện các chức năng cho người dùng cũng như quản lý tương tác của người dùng với hệ thống. Tầng này bao gồm các tính năng trực quan giúp người dùng nhập dữ liệu, thông tin điều khiển hệ thống, và hiển thị kết quả ra cho người dùng. Nó bao gồm các thành phần giao diện (UI Components) và các thành phần xử lý giao diện (UI Process Components).

### 6.1.3.2 Tầng dịch vụ (Service Layer)

Tầng này xuất hiện khi ứng dụng phần mềm xây dựng sẽ cung cấp các tính năng của mình thông qua các dịch vụ cho các hệ thống khác. Tầng này sẽ thực hiện tương tác và sử dụng các đối tượng được định nghĩa trong tầng Business. Bên trong tầng này, ta sẽ định nghĩa và thực thi các giao diện dịch vụ (Service interfaces) và các dữ liệu cung cấp hay nói cách khác là loại thông điệp (Message Types). Kiến trúc thực thi dịch vụ được sử dụng phổ biến cho tầng này chính là REST và SOAP. Ta có thể sử dụng công nghệ ASP.NET Webservices (ASMX) hoặc WCF cho việc cung cấp các dịch vụ.

### 6.1.3.3 Tầng Business (Business Layer)

Đây là tầng xử lý chính của hệ thống với việc định nghĩa các đối tượng dữ liệu cũng như xây dựng các thao tác xử lý cho các đối tượng. Tầng này chịu trách nhiệm biến đổi các đối tượng dữ liệu và làm cho nó phù hợp với tầng trình bày và tầng truy cập dữ liệu. Trong tầng này ta có thể áp dụng các mẫu thiết kế để tối ưu hóa việc mã hóa và tái sử dụng các thành phần được dùng nhiều lần. Trong tầng này gồm các nhóm các chức năng như:

Application Façade – cung cấp một giao tiếp đơn giản cho các thành phần trong tầng business, thường là kết nối các thao tác xử lý đơn giản thành các thao tác xử lý phức tạp hay làm giảm sự phụ thuộc giữa các thành phần với nhau. B

Business Component bao gồm:

- Business Workflow: Định nghĩa các luồng thực thi cho các đối tượng từ việc tiếp nhận dữ liệu từ tầng trình bày, kiểm tra tính hợp lệ của dữ liệu, rồi đưa nó vào các phương thức xử lý và truyền xuống tầng truy cập dữ liệu etc.
- Business Entity: hoặc còn gọi Business Object, đây chính là các đối tượng được đóng gói dựa trên dữ liệu được lấy từ tầng truy cập dữ liệu và các thao tác xử lý chuyên biệt.

### 6.1.3.4 Tầng dữ liệu (Data Layer)

Tầng này cung cấp các chức năng truy cập các cơ sở dữ liệu trên cùng một máy thực thi hoặc tại một máy chủ dữ liệu khác. Nó bao gồm các thành phần truy cập dữ liệu (Data Access Component) và Service Agents được sử dụng để truy cập dữ liệu trên các dịch vụ cung cấp bảo các hệ thống bên ngoài. Hiện nay có nhiều công nghệ được sử dụng tầng truy cập dữ liệu tùy thuộc vào ngôn ngữ lập trình sử dụng. Ví dụ với ngôn ngữ C# nền tảng để truy cập dữ liệu là công nghệ ADO.NET và hiện nay nó vẫn còn được sử dụng phổ biến. Tuy nhiên để tiết kiệm thời gian xây dựng tầng này từ đầu người ta đã phát triển các mô hình ánh xạ dữ liệu tự động từ các bảng dữ liệu sang các đối tượng (Object/Relational

Mapping – ORM). ORM mạnh nhất hiện nay là Entity Framework được phát triển bởi Microsoft dựa trên nền công nghệ ADO.NET.

#### 6.1.4 Sự chọn lựa công nghệ sử dụng

Việc lựa chọn loại ứng dụng phát triển và công nghệ sử dụng dựa trên yêu cầu của phần mềm sẽ giúp ta xem xét việc chọn kiểu kiến trúc sử dụng trong xây dựng phần mềm. Hiện nay có nhiều loại hình ứng dụng khác nhau có thể được sử dụng để xây dựng một ứng dụng phần mềm quản lý trên các nền tảng hệ điều hành khác nhau từ máy chủ, người dùng cá nhân hoặc trên các thiết bị di động, cụ thể:

- **Ứng dụng di động (Mobile Applications):** Ứng dụng dạng này thích hợp cho các phần mềm nhỏ sử dụng đa phần bởi các cá nhân trên những màn hình hạn chế về kích thước. Có 3 nhóm ngôn ngữ lập trình sử dụng phổ biến cho loại ứng dụng này: Java với công cụ Android Studio, Swift dùng lập trình cho IOS và C# với Visual Studio.
- **Ứng dụng Desktop (Desktop Applications):** Sử dụng cho các thiết bị máy tính màn hình lớn, thường dùng cho các phần mềm liên quan đến quản lý trong các doanh nghiệp với các hệ cơ sở dữ liệu. Một số công nghệ có thể sử dụng như Window Forms, hoặc WPF với ngôn ngữ C#.
- **Ứng dụng Web (Web Applications):** Với ứng dụng dạng này phần mềm có thể chạy trên nhiều nền tảng hệ điều hành khác nhau. Một đặc trưng của ứng dụng dạng này là nó có thể được truy cập sử dụng bởi nhiều người dùng cùng một lúc. Do vậy mà kiến trúc xây dựng thường phức tạp hơn các loại hình ứng dụng khác. Có nhiều công nghệ cho việc xây dựng ứng web: ASP.NET với C#, JSP với Java, ngôn ngữ PHP dùng cho việc lập trình phía Server. Phía client đa phần là sử dụng các ngôn ngữ web như HTML, CSS và Javascript.
- **Ứng dụng cung cấp dịch vụ (Service Applications):** Ứng dụng cung cấp dịch vụ (các API tính toán, trích lọc, thống kê) cho các ứng dụng desktop, web, di động. Thông thường nó không được sử dụng trực tiếp bởi nhóm người dùng thông thường, nó được nhúng như là một tầng trong các mô hình kiến trúc phân tầng và được sử dụng bởi các thành phần liên quan trong kiến trúc chung. Công nghệ cho ứng dụng dạng này như WCF, Restful, SOAP etc.

## 6.2 Một số công cụ hỗ trợ xây dựng phần mềm

Trong phần này, một số công cụ hỗ trợ cho người phát triển phần mềm sẽ được giới thiệu. Ở đây những công cụ hỗ trợ cho quá trình phát triển trong một nhóm gồm nhiều

thành viên. Ta không đi vào giới thiệu các công cụ phát triển phục vụ dựa trên các ngôn ngữ lập trình như Visual Studio nếu phát triển ứng dụng với ngôn ngữ C# hoặc C++ MFC, Eclipse hoặc Netbean với Java etc.

### 6.2.1 Quản lý mã nguồn với công cụ Git

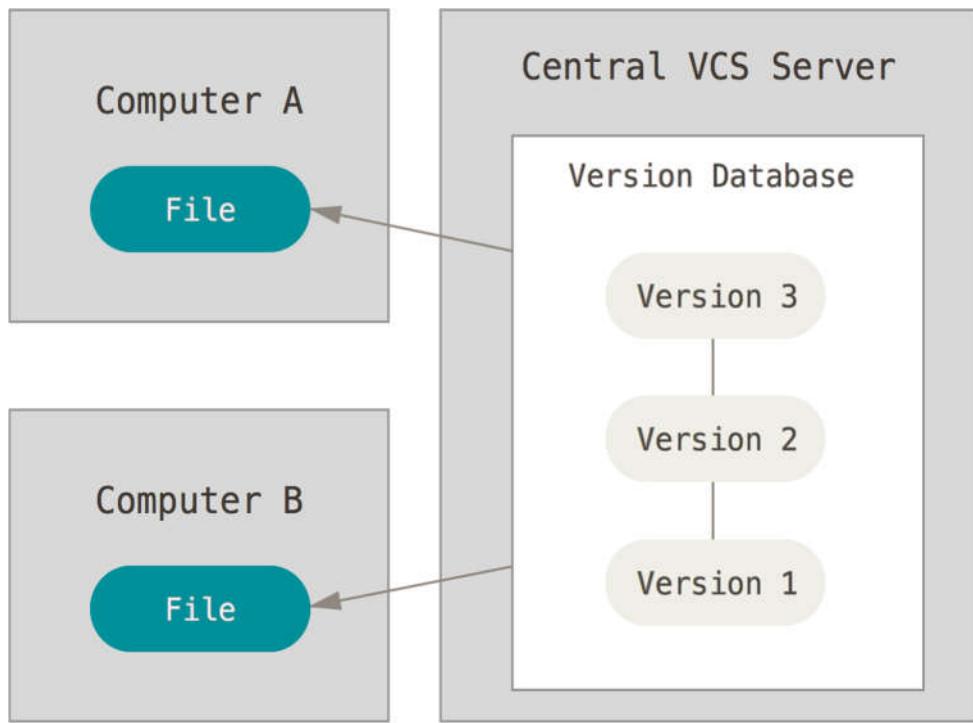
Khi tiến hành xây dựng chương trình được phân chia module cho nhiều thành viên trong một nhóm của một dự án bất kỳ từ một dự án nhỏ vài người đến các dự án lớn vài chục người, chắc chắn rằng chúng ta sẽ gặp phải những vấn đề sau:

- Ai đã chỉnh sửa tập tin X, trước đây nó hoạt động tốt không có lỗi nhưng hiện tại nó lại xuất hiện lỗi?
- Salomé, bạn có thể giúp tôi làm việc trên tập tin X trong khi tôi làm việc trên tập tin Y? Hãy cẩn thận không can thiệp vào tập tin Y bởi vì nếu chúng ta làm việc trên cùng tập tin Y trong cùng một thời điểm có thể mã lệnh của tôi và bạn bị đè lên nhau!
- Ai đã thêm dòng mã này trong tập tin này? Nó không dùng để làm gì cả?
- Những tập tin mới được thêm vào nhằm mục đích gì và ai đã thêm chúng vào dự án?
- Ai đã thực hiện những thay đổi gì để giải quyết lỗi?

Nếu có những vấn đề trên đây, một công cụ quản lý mã nguồn nên sử dụng . Công cụ này giúp các thành viên làm việc cùng với nhau trong cùng một dự án trên cùng mã nguồn. Ngay cả khi bạn là người làm việc duy nhất của dự án, công cụ này sẽ cung cấp cho bạn nhiều tiện ích ví dụ như lưu giữ lịch sử mỗi lần chỉnh sửa trên các tập tin mã nguồn etc.

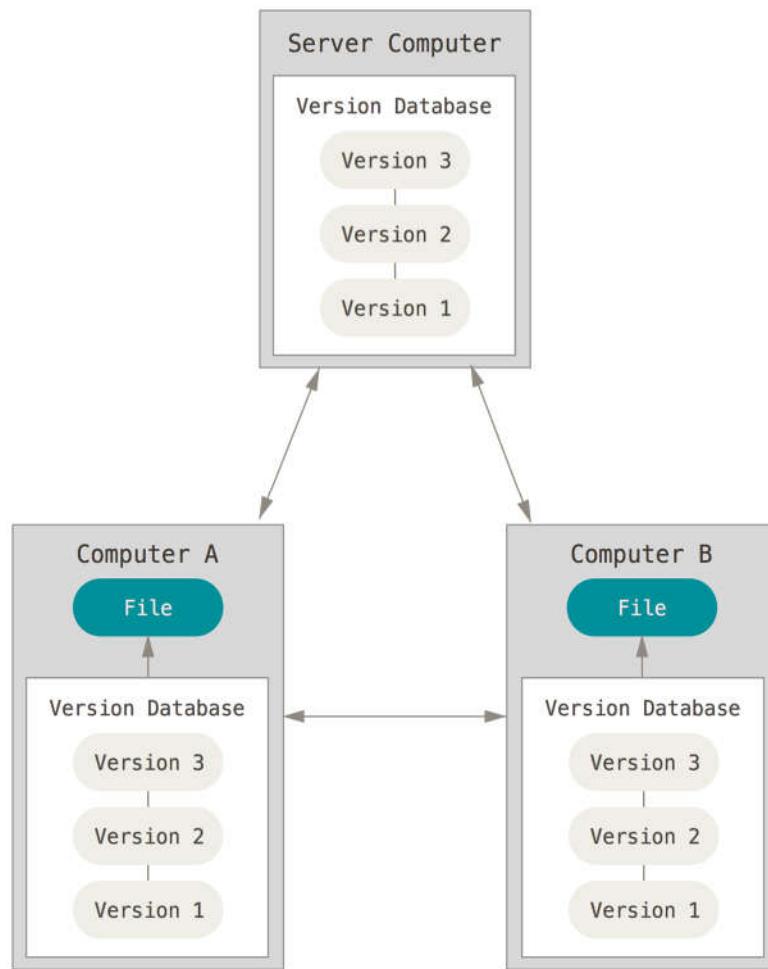
Vậy công cụ quản lý mã nguồn là gì? Công cụ loại này được phát triển nhằm được sử dụng chủ yếu bởi những người phát triển trong các dự án phát triển phần mềm. Người phát triển có thể quản lý mã nguồn của mình cũng như theo dõi sự thay đổi của các tập tin cũng như lưu giữ các phiên bản cũ hơn của các tập tin mã nguồn được viết bởi các thành viên khác nhau hoặc tổ chức phân các nhánh phát triển của phần mềm khác nhau cho các thành viên trong nhóm, giúp chuyên biệt hóa vai trò của từng thành viên. Nếu hai người làm việc đồng thời trên cùng một tập tin, nó có khả năng hợp nhất các sửa đổi để tránh công việc của một trong hai người bị ghi đè lên lẫn nhau.

Công cụ này được phân thành 2 nhóm chính: tập trung và phân tán. Với nhóm công cụ tập trung: một máy chủ lưu giữ các phiên bản cũ của tập tin và kết nối với nó cho các nhà phát triển để đọc các tập tin đã được sửa đổi bởi người khác cũng như để gửi các thay đổi của họ trên các tập tin.



Hình 6.2 Quản lý mã nguồn tập trung

Nhóm các công cụ phân tán: có thể không cần máy chủ, mỗi người đều có lịch sử của sự thay đổi của mỗi tập tin tại máy cục bộ của mình rồi sau đó đồng bộ lên máy chủ (nếu có). Những người phát triển được truyền trực tiếp các thay đổi của các tập tin, theo mô hình giống như peer-to-peer.



Hình 6.3 Quản lý mã nguồn phân tán

Hiện nay có nhiều công cụ khác nhau cho phép quản lý mã nguồn và phiên bản, ví dụ như SVN (Subversion), Mercurial hoặc Git. Trong giáo trình này ứng dụng Git được lựa chọn bởi các tính năng hỗ trợ mạnh mẽ cũng như sự phổ biến của nó trong cộng đồng các nhà phát triển phần mềm hiện nay.

Công cụ	Loại	Mô tả	Dự án sử dụng
CVS	Tập trung	Đây là ứng dụng quản lý mã nguồn lâu đời nhất. Vẫn còn được sử dụng bởi một vài dự án.	OpenBSD
SVN	Tập	Công cụ được sử dụng phổ biến hiện tại,	Apache, Remine,

(Subversion)	trung	nó khá đơn giản để sử dụng. Sử dụng trên window với Tortoise SVN, trước đây được sử dụng bởi google code.	Struts
Mercurial	Phân tán	Công cụ có các chức năng hoàn thiện và hỗ trợ mạnh mẽ, nó xuất hiện sau Git một thời gian ngắn và thường được so sánh với Git	Mozilla, Python, OpenOffice.org
Bazaar	Phân tán	Được phát hành gần đây, chức năng đầy đủ, giống như Mercurial, Bazaar được phát triển bởi Canonical (nhà phát hành Ubuntu). Nó tập trung vào tính tiện dụng và tính mềm dẻo.	Ubuntu, MySQL, Inkscape
Git	Phân tán	Các chức năng hỗ trợ mạnh mẽ, được tạo ra bởi Linus Torvalds (người phát triển Linux). Ưu điểm về tốc độ và khả năng quản lý về các nhánh của phần mềm. Cho phép người phát triển làm việc song song với các tính năng mới của dự án phát triển.	Phát triển Kernel của Linux, Debian, VLC, Android, Gnome, Qt ...

Bảng 6.3 Một số công cụ quản lý mã nguồn

### 6.2.2 Những mô hình làm việc với Git

Git được đánh giá cao ở khả năng tùy biến của nó. Cụ thể, nó cho phép chính các nhà phát triển tạo và quản lý các nhánh mã nguồn của mình một cách dễ dàng. Điều này mang lại khả năng sử dụng Git với rất nhiều mô hình phân nhánh (branching) khác nhau.

Bên cạnh đó, khả năng cho phép làm việc độc lập trên chính máy cục bộ cũng tạo ra rất nhiều kiểu làm việc với Git. Có nhà phát triển có thể tạo những branch (nhánh) khác nhau ở máy cục bộ đợi đến khi làm xong hoàn toàn mới đồng bộ lên server. Cũng có người

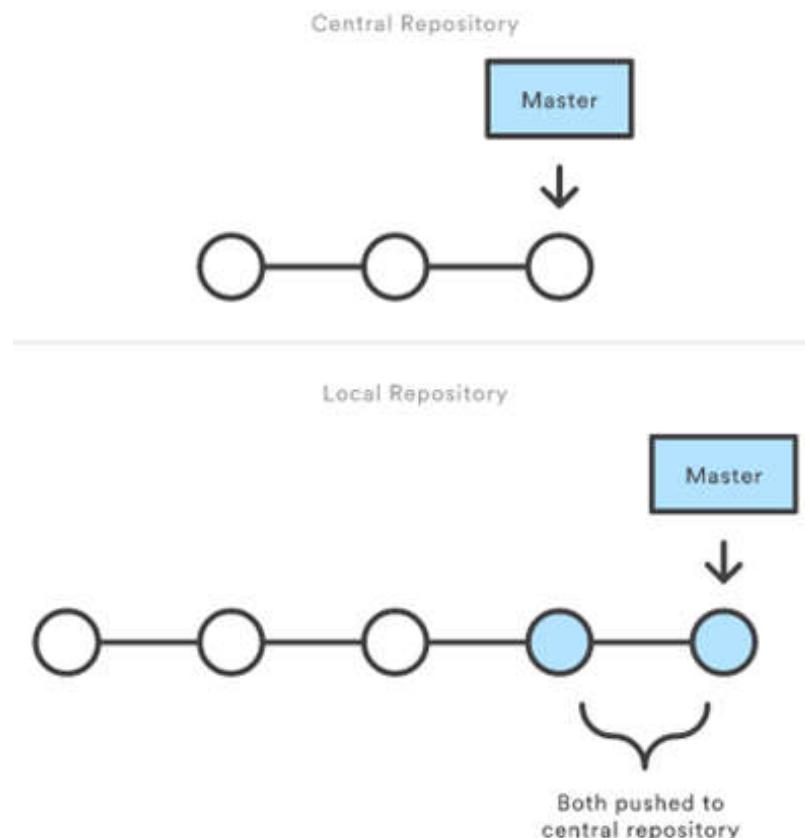
thích đồng bộ hoá mã nguồn của nhánh mình đang làm lên máy chủ một cách định kì hàng ngày ...

Chính vì Git tạo ra nhiều mô hình làm việc đa dạng như vậy – nên để làm việc với Git thật sự hiệu quả, ta nên hiểu rõ về những khái niệm cơ bản của Git cho đến các mô hình làm việc khác nhau.

#### 6.2.2.1 Mô hình tập trung (*Centralized workflow*)

Đây là mô hình khá giống với cách ta làm việc với các hệ thống quản lý mã nguồn tập trung trước đây: những người phát triển cùng làm việc tập trung trên một nhánh chính của phần mềm phát triển.

Trong mô hình này, người phát triển sẽ thực hiện clone (tạo một bản sao) nhánh master về máy cục bộ của mình. Sau đó, thực hiện việc lập trình, thay đổi mã nguồn. Cuối cùng sẽ đẩy (push) sự thay đổi ở máy cục bộ lên máy chủ thông qua lệnh git push.



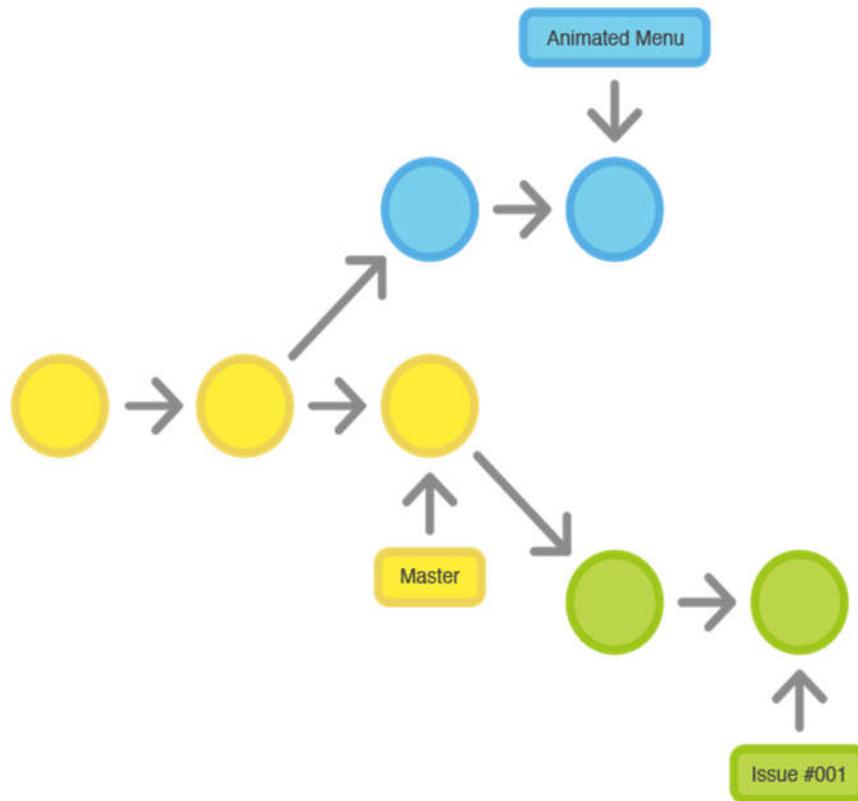
Hình 6.4 Cách làm việc với mô hình sử dụng Git tập trung

Thông thường mô hình này được áp dụng cho những nhóm trước đây đã quá quen thuộc với các hệ thống quản lý mã nguồn tập trung như SVN. Áp dụng mô hình này ở những

bước đầu tiên sẽ giúp nhóm phát triển làm quen dần với git trước khi sử dụng đến những mô hình làm việc khác. Điểm yếu của mô hình này là không tận dụng được nhiều sức mạnh của git. Đặc biệt là ở tính phân tán của nó.

### 6.2.2.2 Mô hình Feature branch

Mô hình này là bước chuyển tiếp sau khi có một nhóm phát triển đã tương đối quen thuộc với Git. Ý tưởng chính của mô hình này là: việc phát triển tính năng nên được diễn ra ở những nhánh riêng dành cho nó, thay vì nhánh master. Điều này làm cho những người phát triển sẽ làm việc trên những nhánh riêng mà không làm cho nhánh chính bị ảnh hưởng. Cũng đồng nghĩa với nó, nhánh master sẽ không bao giờ được chứa broken code – điều cực kì quan trọng khi làm việc với mô hình tích hợp liên tục (Continuous Integration).



Hình 6.5 Cách làm việc với mô hình feature branch

Khi bắt tay vào làm một tính năng mới hay thực hiện việc sửa lỗi trên mã nguồn – ta sẽ tạo ra một nhánh tương ứng cho chức năng làm từ bản sao ra từ nhánh master. Trên thực tế, người ta khuyến khích đặt tên cho feature branch sao cho mang tính mô tả cao và thật rõ ràng. Đây cũng là cách giúp định hình một cách rõ ràng về tính năng/công việc sẽ phát triển trên nhánh.

Sau khi đã hoàn thành công việc của mình trên nhánh feature. Công việc tiếp theo sẽ là hợp nhất mã nguồn (merge code) về nhánh chính. Tuy nhiên, người ta khuyến khích việc tạo một Pull request từ nhánh feature về master thay vì trộn trực tiếp. Pull request (hay ở một số git repository service, nó được gọi là merge request – yêu cầu hợp nhất mã nguồn) là một khái niệm cho phép người phát triển thảo luận, yêu cầu sự giúp đỡ của những người phát triển khác để xem xét những thay đổi của mình trước khi hợp nhất mã nguồn về một nhánh. Đây chính là nơi mà hoạt động xem trước mã nguồn diễn ra một cách hiệu quả nhất.

Khi tất cả thay đổi được xem xét, thì code của nhánh sẽ sẵn sàng để trộn về nhánh chính. Thông thường đối với những nhóm phát triển nhỏ và còn yếu về kĩ thuật, thì người đứng đầu nhóm (leader) nên thực hiện việc hợp nhất các pull request về nhánh chính. Thật ra thì ai thực hiện điều này không quan trọng vì khi tất cả thay đổi đã được xem xét ở pull request, thì thao tác hợp nhất thực sự là đơn giản. Câu hỏi đúng để quyết định tính hiệu quả của mô hình này là: ai sẽ xem xét (review) thay đổi trên Pull request, làm sao để khuyến khích tất cả những người phát triển tham gia xem xét, nên xem xét những gì?

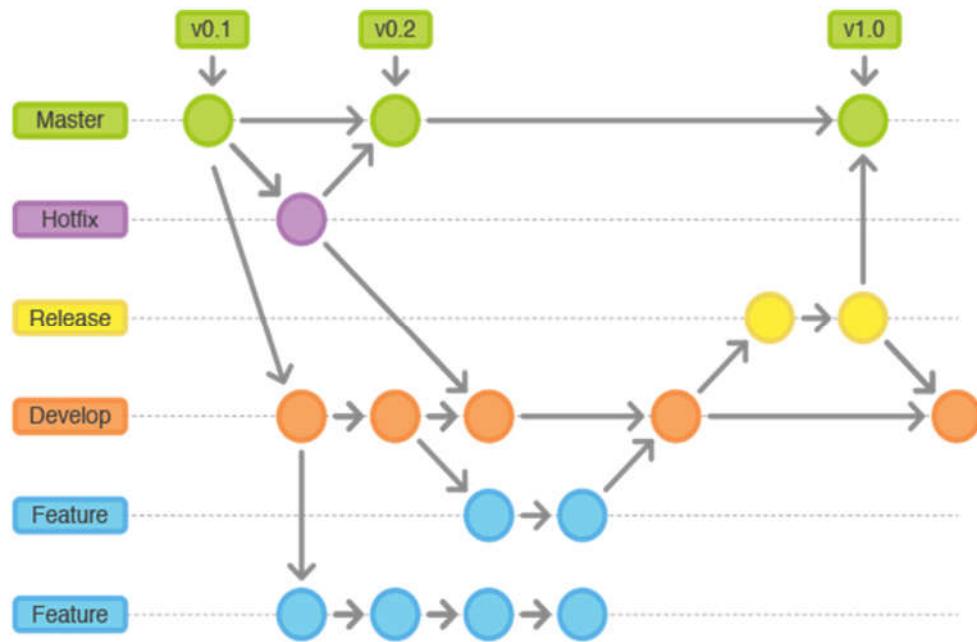
Trước khi tạo pull request, cần đảm bảo đã thực hiện các bước sau:

- Hợp nhất mã nguồn ở nhánh mà ta muốn hợp nhất về phía nhánh feature. Giả sử ta muốn tạo Pull request về nhánh master, ta phải hợp nhất mã nguồn của master về nhánh feature hiện đang làm việc.
- Thực hiện giải quyết tất cả các tranh chấp (conflict - nếu có) trên feature branch. Sau đó push nhánh feature của mình lên Git server (remote branch).
- Tạo pull request từ nhánh feature branch remote về nhánh muốn merge

Tại sao lại phải làm như vậy? Về mặt bản chất – khi tạo ra 1 Pull request, có nghĩa là ta muốn những người phát triển khác xem sự thay đổi của mã nguồn trên nhánh của chính mình so với nhánh muốn hợp nhất lại. Trong quá trình bạn phát triển feature, có thể sẽ có nhiều người phát triển cũng đã merge feature của họ về nhánh chính. Nếu không thực hiện thao tác hợp nhất như mình đề cập ở trên thì sự thay đổi mã nguồn trên nhánh đã quá xa so với nhánh cần hợp nhất. Như vậy khi người khác xem xét sự thay đổi trong Pull request, họ sẽ thấy có rất nhiều thay đổi diễn ra và sẽ không hiệu quả cho việc xem xét.

#### **6.2.2.3 Mô hình Gitflow**

Mô hình gitflow cũng khá giống với feature branch. Tuy nhiên, nó được thiết kế với những ràng buộc chặt chẽ để hỗ trợ cho các phát hành (release) của dự án.



Hình 6.6 Cách làm việc với mô hình gitflow

Một số điểm chính của mô hình gitflow:

- Khi phát triển feature, ta sẽ không tạo nhánh trực tiếp từ master – mà phải tạo nhánh từ một nhánh phát triển (branch develop).
  - Khi phát triển xong feature trên branch, ta sẽ tạo pull request về nhánh phát triển. Như vậy, nhánh phát triển sẽ là nhánh tập trung tất cả những feature chính của đợt release sắp tới.

Ở một thời điểm, nếu thấy rằng mình đã có thể release được – ta sẽ tiến hành bước release bằng các bước dưới đây:

- Tạo một branch release từ nhánh phát triển.
  - Sau đó, làm tất cả các thao tác chuẩn bị hậu cần cho branch release này, như: cập nhật documentation, test etc.
  - Khi nhánh release này đã hoàn thành, ta sẽ hợp nhất nó về nhánh master và đánh tag với một số phiên bản (version number). Khi gửi bản release, ta sẽ gửi thông tin tag này.

Trong mô hình gitflow, nếu cần cố định trước (hotfix) trên đợt release trước – ta sẽ tạo một branch hotfix từ nhánh master. Thực hiện hotfix và hợp nhất ngược lại vào master và cả nhánh phát triển.

Ưu điểm của mô hình này là giúp tách biệt các hoạt động release, phát triển tính năng mới, sửa chữa các lỗi cho bản release cũ mà không ảnh hưởng đến nhau.

Tuy nhiên khi vận dụng mô hình này, một số nhóm phát triển cho rằng nó khá cồng kềnh đối với dự án nhỏ. Các dự án làm sản phẩm – đặc biệt là những sản phẩm nguồn mở thì lại khá thích hợp cho mô hình này.

#### **6.2.2.4 Mô hình forking**

Forking là mô hình sử dụng Git được áp dụng rất nhiều trong các dự án nguồn mở và public cho cộng đồng cùng tham gia phát triển. Trong mô hình này, người sở hữu (owner) của sản phẩm/ dự án sẽ tạo ra một official repository của mình. Ta có thể host repository trên bất kì git repo nào (github, bitbucket, gitlab ...).

Ta sẽ phát triển những tính năng sản phẩm trên repository của mình. Tuy nhiên, nếu những nhà phát triển khác cảm thấy hứng thú với sản phẩm và họ muốn đóng góp – họ sẽ fork repository về repository của họ. Thao tác fork thực ra là tạo một bản sao toàn bộ thông tin của git repository về repository cá nhân của một người. Sau khi phát triển xong tính năng hay sửa xong lỗi mà họ muốn đóng góp, họ vẫn sẽ theo một tiến trình Pull request để người chủ dự án xem xét những thay đổi và quyết định có hợp nhất Pull request hay không.

Điều quan trọng trong mô hình forking là nên chia sẻ về tiến trình sẽ hợp nhất các tính năng đang thực hiện của các người phát triển trong cộng đồng về nhánh nào, nhánh nào là nhánh khuyến khích mọi người clone branch ra để phát triển tiếp feature, cần đảm bảo những ràng buộc gì thì mã nguồn mới được hợp nhất về repository.

#### **6.2.2.5 Mô hình GitHub áp dụng**

Nhóm Github áp dụng một mô hình khá giống với feature branch để đơn giản hoá quy trình làm việc của họ. Tuy nhiên, họ duy trì một số ràng buộc dưới đây:

- Nhánh master luôn luôn là nhánh có thể triển khai được ở bất kì mọi thời điểm.
- Commit lên feature branch local và push lên feature branch remote thường xuyên để đảm bảo code ở máy tính cục bộ không bị mất.
- Chỉ có thể hợp nhất feature branch về master sau Pull request khi đã được xem xét và xác nhận.
- Khi feature branch được hợp nhất về master thì ta nên triển khai luôn sau đó

Trên thực tế, để mô hình này được duy trì tốt khi đã xây dựng được một hệ thống CI và unit test được áp dụng chặt chẽ trong những feature mà ta phát triển.

Việc quyết định sử dụng mô hình nào với git sẽ phụ thuộc:

- Mức độ quen thuộc của các thành viên team với git.
- Tính chất của dự án, sản phẩm đang làm việc.
- Nhu cầu tích hợp liên tục của sản phẩm

Việc sử dụng mô hình nào đôi khi sẽ cần những thử nghiệm cụ thể – hoặc đôi khi bổ sung thêm một số ràng buộc để phù hợp với tính chất của nhóm dự án.

### 6.2.3 Quản lý tiến độ công việc với Kanban

Ngày nay, các dự án phần mềm thường xuyên gặp phải tình trạng quá tải trong công việc, một phần nguyên nhân của việc này là do việc phân bổ, quản lý tiến độ công việc không phù hợp dẫn đến hiệu suất công việc không đạt được tối đa. Vì vậy, đối với các thành viên trong dự án cần phải có phương pháp làm việc hiệu quả để sắp xếp công việc khoa học hơn và nâng cao hiệu suất.



Hình 6.7 Mô hình sử dụng bảng Kanban

Có rất nhiều cách quản lý công việc khác nhau được sử dụng. Trong thực tế mô hình Kanban là lựa chọn của nhiều nhóm phát triển. Về lịch sử, phương pháp Kanban được phát triển ở Nhật Bản sau chiến tranh Thế giới thứ 2 và được ông M.OHNO áp dụng ở tập đoàn Toyota Motor từ những năm 1959 và đến nay vẫn được Thế giới sử dụng rộng rãi trong nhiều lĩnh vực khác nhau. Nguyên tắc của phương pháp này rất đơn giản. Công việc thực hiện sẽ được phân chia và lên kế hoạch được đặt trên 3 cột: To do( kế hoạch), Doing – Work in process (đang thực hiện) và Done (đã hoàn thành). Đầu tiên, lập kế hoạch công việc trong ngày/tuần và đặt trên trạng thái to do. Điều này sẽ giúp ta có được trình tự và thực hiện công việc một cách khoa học. Thông thường, nhiều thành viên trong nhóm phát

triển dự án sẽ bắt tay vào làm ngay việc được giao mà không suy nghĩ, tính toán. Đây không phải là cách hay và tối ưu. Thực tế, các công việc cần làm ta nên sắp xếp độ ưu tiên theo giá trị (cái nào có giá trị thì làm trước), ta có thể mất ít công sức hơn mà làm được nhiều giá trị hơn (theo quy tắc Pareto, 80-20).

Khi quyết định làm việc gì, ta sẽ chuyển công việc sang cột Doing và ghi thời gian lên trên từng công việc. Tránh để các công việc chồng chéo lên nhau sẽ gây mất tập trung và không biết nên làm gì ưu tiên.

Khi làm xong việc gì thì chuyển sang cột Done, lưu ngày hoàn thành trên từng công việc để phục vụ công tác báo cáo và đánh giá về sau. Việc đặt một công việc sang cột Done chứ không vứt đi sẽ giúp ta nhìn thấy được tiến độ công việc, từ đó thực đầy bản thân thực hiện công việc. Nói chung, ta có thể dùng Kanban với 4 nguyên lý sau:

- **Trực quan hóa công việc:** Bảng Kanban là công cụ để trực quan hóa công việc. Bảng Kanban bao gồm các cột tương ứng với trạng thái của công việc. Mỗi công việc khi ở trạng thái nào thì được đặt ở cột tương ứng. Chúng ta có thể dùng một bảng vật lý hoặc các phần mềm hỗ trợ theo nguyên lý của Kanban.
- **Giới hạn công việc đang làm** (Limit WIP – Limit Work In Progress): Số lượng công việc đang được làm đồng thời ở mỗi trạng thái cần được giới hạn. Nguyên lý này giúp giới hạn những việc chưa hoàn thành trong tiến trình, từ đó giảm thời gian mỗi công việc đi qua hệ thống Kanban. Nguyên lý giới hạn WIP còn giúp cho nhóm làm việc tập trung, tránh lãng phí do phải việc chuyển qua lại giữa các công việc khác nhau.
- **Tập trung vào luồng làm việc:** Việc áp dụng nguyên lý giới hạn WIP và phát triển những chính sách hướng theo nhóm giúp nhóm có thể tối ưu hóa hệ thống Kanban để cải tiến luồng làm việc trơn chu.
- **Cải tiến liên tục:** Nhóm đo mức độ hiệu quả bằng cách theo dõi chất lượng, thời gian làm sản phẩm, v.v. để từ đó có những phân tích, thử nghiệm để thay đổi hệ thống nhằm tăng tính hiệu quả của nhóm.

Hiện tại, có nhiều công cụ cho phép sử dụng nguyên lý Kanban chạy trên nhiều nền tảng khác nhau, từ Android, iOS đến Windows hay Ubuntu trên môi trường Web hoặc trên các ứng dụng trong các hệ điều hành di động. Công cụ loại này có thể liệt kê như: Trello, KanbanFlow, Pomodoro Daisuki, LeanKit, Kanbanery, JIRA, KanbanTool, PearlTrees, ... Với những công cụ hỗ trợ đa nền tảng (ví dụ như Trello) bạn có thể tạo lập một bảng công việc lưu trữ “trên mây”, truy xuất ở mọi nơi, mọi lúc.



Hình 6.8 Mô hình sử dụng Kanban trên nhiều nền tảng

Việc tạo lập bảng Kanban trên các dịch vụ này khá dễ dàng đa phần là miễn phí cho các nhóm nhỏ từ 3 đến 5 người. Truy cập vào trang web của nhà cung cấp và đăng ký một tài khoản, ta có thể sử dụng một bảng kanban tiện dụng cho mục đích công việc hằng ngày.

Hình 6.9 Ví dụ minh họa sử dụng Kanban với Trello ([trello.com](http://trello.com))

### 6.3 Một số nguyên tắc cơ bản trong xây dựng phần mềm

#### 6.3.1 Chuẩn hóa mã hóa

Việc mã hóa là công việc đòi hỏi tính logic cao và phức tạp. Để có một chương trình tốt trước hết phải bắt đầu từ những đoạn mã tốt, việc này đòi hỏi người lập trình mất nhiều thời gian và công sức. Hầu hết những người mới bước vào lĩnh vực xây dựng phần mềm

thường viết mã không theo một khuôn mẫu nào và theo kiểu miễn sao chương trình chạy là được do đó họ thường rất ít quan tâm đến phong cách viết mã và cách viết chương trình tốt. Tuy nhiên đây là một thói quen xấu mà nếu không thiết lập ngay từ đầu sẽ rất khó thay đổi về sau. Hậu quả thường thấy từ những việc viết mã tự do như vậy là sự lộn xộn của mã trong ứng dụng và nó có thể gây nhiều phần toái sau này ví dụ như khi muốn phải thay đổi và chỉnh sửa mã.

Thông thường mỗi lập trình viên đều có phong cách mã (Coding Style<sup>1</sup>) khác nhau, phong cách đó sẽ được hình thành dựa trên những nguyên tắc chung trong lập trình, chưa quan tâm đến là người đó sử dụng ngôn ngữ lập trình nào. Để tạo được một phong cách viết mã hiệu quả, phải biết được mã xấu “Code smells” và biết cách tùy chỉnh nó sao cho tối ưu nhất.

Việc phải có những quy tắc chung là bắt buộc trong ngành công nghệ phần mềm khi mà việc xây dựng các phần mềm luôn được thực hiện bởi nhóm nhiều người. Do vậy việc xây dựng phần mềm trong các nhóm phát triển luôn có những quy ước chung cho các thành viên trong nhóm để có thể dễ dàng xây dựng các chức năng của phần mềm. Mỗi ngôn ngữ lập trình có những quy tắc mã hóa khác nhau. Trong phần này, sẽ lấy ngôn ngữ C# làm ngôn ngữ chủ đạo để thể hiện được quy tắc và phong cách viết mã trong xây dựng phần mềm.

### 6.3.1.1 Quy tắc đặt tên

Các kiểu đặt tên

Kiểu	Ví dụ	Mô tả
Pascal	SinhVien	Chữ cái đầu tiên trong từ định danh và chữ các đầu tiên của mỗi từ theo sau phải được viết hoa. Sử dụng Pascal Case để đặt tên cho một tên có từ 3 ký tự trở lên
Camel	sinhVien	Chữ cái đầu tiên trong từ định danh là chữ thường và chữ cái đầu tiên của mỗi từ nối theo sau phải được viết hoa
Uppercase	SINHVIEN	Tất cả các ký tự trong từ định danh phải được viết hoa. Sử dụng quy tắc này đối với tên định danh có 2 ký tự trở xuống

Bảng 6.4 Phân biệt các kiểu đặt tên

Trường hợp sử dụng

Loại	Kiểu đặt tên	Ví dụ	Ghi chú

<sup>1</sup> Coding Style là các quy tắc trong quá trình viết mã bao gồm các quy tắc về ngữ pháp và ngữ nghĩa như quy tắc đặt tên hàm, biến, cách comment v.v..

Tên dự án (Project file)	Pascal	QuanLySinhVien	
Tên file mã nguồn (Source File)	Pascal	SinhVien.cs	
Tên biến (Variable)	Camel	hoTen	
Hằng số (Constant)	Uppercase	SO_PI	Có gạch chân giữa các từ
Tên class, enum, Delegate	Pascal	SinhVien, XepLoai	
Tham số (Parameter)	Camel	tenKhachHang	
Thuộc tính (Property)	Pascal	NgaySinh	
Phương thức (method)	Pascal	TinhDiemTrungBinh()	
Sự kiện (event)	Pascal	ClickEventHandler	Thường có hậu tố EventHandler
Giao diện (Interface)	Pascal	IHinhHoc	Thường có tiền tố I

Bảng 6.5 Phân biệt các trường hợp sử dụng

Tiền tố một số điều khiển

STT	Tên điều khiển	Tiền tố	Ví dụ
1	Panel	pnl	pnlGroup
2	CheckBox	chk	chkReadOnly
3	ComboBox, Drop-DownList Box	cbo	cboCountry
4	Command Button	btn	btnExit
5	Common dialog	dlg	dlgFileOpen
6	Data	dat	datBiblio
7	Data-Bound Combo Box	cbo	cboLanguage
8	Data-Bound grid	grd	grdQueryResult
9	Data-Bound List Box	lst	lstJobType
10	Repeater	rpt	drpLocation
11	DateTimePicker	dtp	dtpPublished
12	Form	frm	frmEntry
13	Frame	fra	fraLanguage
14	DataGridView	dgv	dgvPrices
15	GridView	grd	grdProduct
16	DownList	dtl	dtlOrders
17	Horizontal scroll bar	hsb	hsbVolume
18	Image	img	imgIcon
19	ImageList	ils	ilsAllIcons

20	ImageButton	ibt	ibtNext
21	HyperLink	hpl	hplHome
22	LinkButton	lbt	lbtClick
23	Label	lbl	lblHelpMessage
24	List box	lst	lstPolicyCodes
25	ListView	lvw	lvwHeadings
26	Menu	mnu	mnuFileOpen
27	Option button	opt	optGender
28	Picture box	pic	picVGA
29	Picture clip	clp	clpToolbar
30	ProgressBar	prg	prgLoadFile
31	RichTextBox	rtf	rtfReport
32	Slider	sld	sldScale
33	Spin	spn	spnPages
34	StatusBar	sta	staDateTime
35	TextBox	txt	txtLastName
36	Timer	tmr	tmrAlarm
37	Toolbar	tlb	tlbActions
38	TreeView	tre	treOrganization
39	UpDown	upd	updDirection
40	Vertical scroll bar	vsb	vsbRate
41	SqlDataSource	sql	sqlAccounts
42	LinqDataSource	linq	linqCategories

Bảng 6.6 Tiền tố điều khiển

### 6.3.1.2 Quy tắc viết mã

**Quy tắc viết comment:** Sử dụng ngôn ngữ quy ước (tiếng việt hoặc tiếng anh) phụ thuộc vào quy định của nhóm hoặc công ty về ngôn ngữ sử dụng.

- Comment cho module, class: mỗi module, class cần có mô tả ngắn về mục đích của module hay class đó. Nội dung gồm:
  - Mục đích: module hay class thực hiện những công việc gì
  - Người lập: người tạo module hay class
  - Những biến/hàm quan trọng (không bắt buộc): liệt kê tên các biến và hàm quan trọng trong module/class.

- Comment cho method hoặc event: tất cả các method và event phải có comment.  
Comment cho method/event gồm hai phần:
  - Phần 1 (không bắt buộc): mô tả mục đích và diễn giải ngắn gọn ý nghĩa các tham số đầu vào, đầu ra. Lưu ý: mô tả method đó làm gì (What), không mô tả method đó thực hiện thế nào (how). Người lập trình có thể không cần viết phần mô tả mục đích này với các method/event đơn giản.
  - Phần 2 (bắt buộc): ghi thông tin về lịch sử tạo và sử method/event đó (người tạo/ngày tạo, người sửa/ngày sửa). Thông tin này bắt buộc phải có với mọi method/event.
  - Ví dụ comment cho method/event đơn giản:

**Ví dụ:** comment cho method/event đơn giản

```
//Created by HoaiKhong - 31/05/2016:  
//Lấy danh sách sinh viên theo lớp  
//Modified by VoDanh - 01/06/2016:  
//Sửa tham số truy vào tên lớp thay vì mã lớp  
protected void DanhSachSinhVienTheoLop(string tenLop)  
{  
  
}
```

Hình 6.10 Comment cho các phương thức và event đơn giản

**Ví dụ:** comment cho method/event phức tạp

```
/// <summary>  
/// Lấy bảng điểm sinh viên theo học kỳ  
/// </summary>  
/// <param name="maSinhVien">Mã sinh viên</param>  
/// <param name="hocKy">Mã học kỳ</param>  
/// <remarks>Nhận xét - nếu có</remarks>  
/// Created by HoaiKhong - 31/05/2016:  
/// Lấy bảng điểm sinh viên theo học kỳ  
/// Modified by VoDanh - 01/06/2016:  
/// Sửa tham số truy vấn SQL  
protected void XemBangDiem(string maSinhVien, string hocKy)  
{  
  
}
```

Hình 6.11 Comment cho các phương thức và event phức tạp

- Comment cho đoạn code: những đoạn mã phức tạp cần có comment gắn liền bên trên để chú thích. Những đoạn mã sửa đổi (modified), bổ sung (added) hoặc xóa bỏ (removed) bởi người không phải là người tạo ra cần có comment rõ ngay tại nơi sửa đổi, bổ sung: người sửa, ngày sửa, mục đích

### Quy tắc phân nhóm

Phải sử dụng region phân nhóm mã để thuận tiện cho việc sửa đổi, bảo trì. Phân nhóm mã theo cấu trúc như sau (theo thứ tự bắt buộc, nhưng không bắt buộc có đủ tất cả các region): Declaration, Constructor, Property, Method/Function, Event. Tùy theo yêu cầu của các form, class và module, người lập trình có thể chia nhỏ các region chính thành các sub-region. Ví dụ như region của Method/Function có thể chứa các region con sau: Method/Function Public, Overridable (trường hợp là base form/class), Override (trường hợp là derive form/class), Private, Other.

Trường hợp form hoặc class có sử dụng các component độc lập (Security, Document, MassEmail,...) thì phải tạo các region riêng cho từng component, chứa toàn bộ mã liên quan đến việc tương tác với các component đó.

### Quy tắc bẫy lỗi

Bắt buộc bẫy lỗi bằng cách sử dụng khối lệnh try ... catch hoặc using trong tất cả các xử lý sự kiện của form và control trên form, trong các xử lý kết nối với cơ sở dữ liệu, trong việc đọc ghi thông tin vào các tập tin liên quan đến các đối tượng nhập xuất dữ liệu .vv. .. Một lưu ý là không sử dụng khối lệnh lệnh try ... catch để che dấu lỗi tức là không xử lý gì sau từ khóa catch.

#### 6.3.2 Sử dụng mẫu thiết kế

Bên cạnh phong cách mã của từng cá nhân trong xây dựng phần mềm thì các mẫu các mô hình thiết kế cũng là một phần quan trọng được áp dụng để xây dựng một chương trình tốt. Cuối cùng trong công nghệ phần mềm, một mẫu thiết kế (design pattern) là một giải pháp tổng thể cho các vấn đề chung trong thiết kế phần mềm. Một mẫu thiết kế không phải là một thiết kế hoàn thiện để có thể được chuyển đổi trực tiếp thành mã, mà nó chỉ là một mô tả hay là mẫu (template) mô tả cách giải quyết một vấn đề mà có thể được dùng trong nhiều tình huống khác nhau. Các mẫu thiết kế hướng đối tượng thường cho thấy mối quan hệ và sự tương tác giữa các lớp hay các đối tượng, mà không cần chỉ rõ lớp hay đối tượng của từng ứng dụng cụ thể. Các giải thuật toán không được xem là các mẫu thiết kế vì chúng giải quyết các vấn đề về tính toán hơn là các vấn đề về thiết kế.

Các mẫu thiết kế có thể giúp tăng tốc quá trình phát triển phần mềm bằng cách cung cấp các mô hình phát triển đã được chứng thực và kiểm chứng. Để thiết kế phần mềm hiệu quả đòi hỏi phải xem xét các yếu tố mà chỉ trở nên rõ ràng sau khi đã thành hiện thực. Xác định được chúng, thông qua các mẫu thiết kế, chúng ta sẽ thoát khỏi chúng vì chúng có thể dẫn đến những rắc rối và cải tiến khả năng dễ đọc của mã cho người viết mã và các nhà kiến trúc sẽ cảm thấy quen thuộc với các mẫu.

Thông thường người lập trình chỉ biết cách áp dụng một số kỹ thuật thiết kế phần mềm nào đó vào một vài vấn đề cụ thể nào đó. Những kỹ thuật này khó áp dụng mở rộng cho các vấn đề khác. Các mẫu thiết kế cung cấp các giải pháp chung, được viết tài liệu dưới một định dạng mà không gắn liền với một vấn đề cụ thể nào. Việc vận dụng các mẫu thiết kế vào chức năng cụ thể của phần mềm sẽ phụ thuộc vào cách nhìn nhận và kinh nghiệm của người lập trình.

Các mẫu thiết kế cho phép các người phát triển giao tiếp với nhau dùng các tên dễ hiểu, được dùng rộng rãi để đặt cho các tương tác của phần mềm. Các mẫu thiết kế chung có thể được cải tiến qua thời gian, để trở nên ổn định hơn là thiết kế tùy biến.

Các mẫu thiết kế có thể được phân loại dựa vào nhiều tiêu chí, chung nhất là dựa vào vấn đề cơ bản mà chúng giải quyết. Theo tiêu chuẩn này, các mẫu thiết kế có thể được phân loại thành nhiều nhóm khác nhau:

- Các mẫu cơ sở (Fundamental pattern): Interface, Abstract Parent Class, Private Method, Accessor Methods, Monitor
- Các mẫu tạo lập (Creational pattern): Factory, Singleton, Abstract, Prototype, Builder
- Các mẫu cấu trúc (Structural pattern): Decorator, Adapter, Chain of Responsibility, Façade, Proxy, Bridge, Virtual Proxy, Counting Proxy
- Các mẫu ứng xử (Behavioral pattern): Command, Mediator, Memento, Observer, Interpreter, State, Strategy, Null Object, Template Method,
- Các mẫu đồng thời (Concurrency pattern): Critical Section, Consistent Lock, Guard Suspension, Read-Write lock
- Các mẫu tập hợp (Collectional Patterns): Composite, Iterator, Flyweight

### Ví dụ mẫu thiết kế Singleton

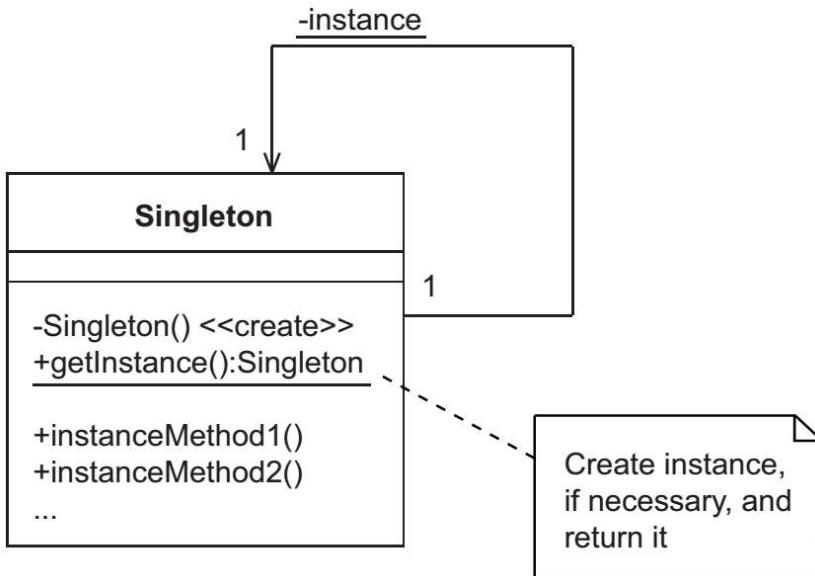
Để nắm vững những nguyên lý bên trong của mỗi mẫu thiết kế, người lập trình cần hiểu và áp dụng vào một vấn đề cụ thể. Ví dụ khi một lớp được xây dựng và có yêu cầu chỉ có duy nhất một thê hiện (instance) được sử dụng xuyên suốt trong chương trình khi thực thi

thì mẫu thiết kế Singleton có thể được lựa chọn để áp dụng. Theo thời gian sử dụng phần mềm, việc tạo một thể hiện duy nhất cho một lớp đối tượng là hữu ích trong việc tiết kiệm không gian bộ nhớ sự dụng của phần cứng cũng như là tối ưu hóa thời gian thực thi bằng việc hạn chế việc khởi tạo và hủy một đối tượng. Ví dụ chúng ta chỉ cần sử dụng một đối tượng Calendar duy nhất để trả lời cho tất cả những câu hỏi “Có bao nhiêu ngày trong tháng 2” thay cho việc tạo ra nhiều đối tượng Calendar khác nhau. Để sử dụng mẫu thiết kế Singleton một cách hữu ích, những tiêu chí được ra khi xem xét sử dụng mẫu thiết kế này:

- Nó phải dễ dàng để tìm kiếm
- Đảm bảo là một thể hiện duy nhất
- Không được tạo ra cho đến khi nó cần thiết

Để làm cho mẫu Singleton dễ dàng tìm thấy, chúng ta có thể lưu trữ nó trong một thuộc tính của lớp và được truy cập như một thông điệp của lớp. Điều này là dễ dàng trong ngôn ngữ lập trình hướng đối tượng, các lớp đã được sử dụng từ lâu như là một điểm khởi đầu cho việc chia sẻ dữ liệu và dịch vụ. Để đảm bảo chỉ có một thể hiện được tạo, chúng ta thiết lập thuộc tính dùng cho việc tạo thể hiện này là không thể truy cập bằng việc đặt chế độ private.

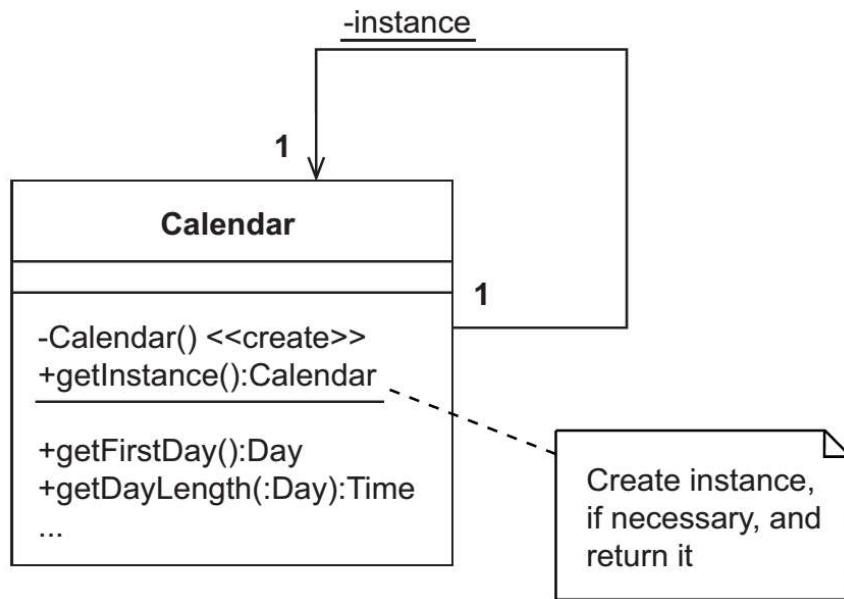
Lớp sử dụng Singleton chỉ được khởi tạo thực sự khi cần thiết. Điều này nhằm mục đích hạn chế việc khởi tạo một đối tượng mà thực tế không bao giờ sử dụng. Chúng ta có thể lựa chọn việc khởi tạo đối tượng khi hệ thống bắt đầu hoặc khi nhận được yêu cầu. Nếu chọn lựa cách thứ hai thì ta có thể sử dụng kỹ thuật *lazy initialization* ở đây điều kiện logic được sử dụng để kiểm tra để xác định thể hiện của lớp đã được khởi tạo hay chưa.



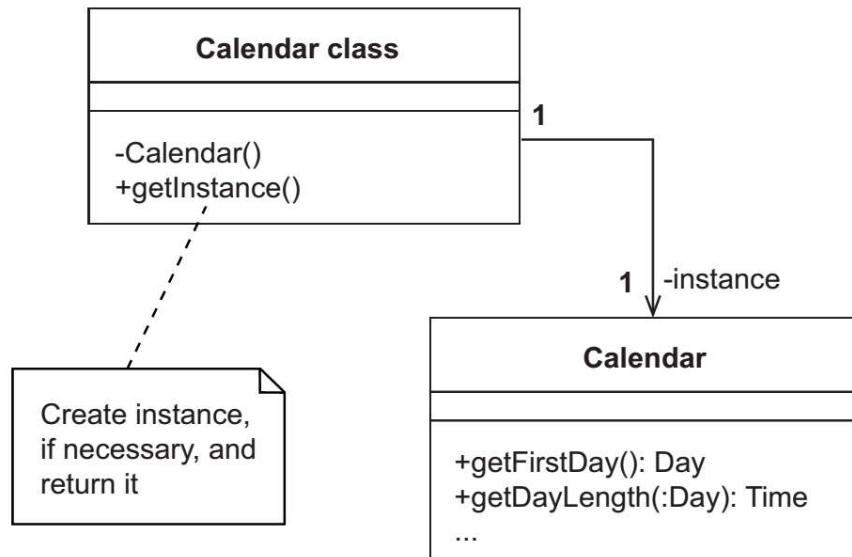
Hình 6.12 Sơ đồ lớp sử dụng Singleton

Sơ đồ trên thể hiện một thực thi của mẫu thiết kế Singleton được trình bày trong sơ đồ lớp. Ở đây, chúng ta có một biến *instance* của lớp Singleton được đặt chế độ private, đối tượng này được tạo theo yêu cầu bởi phương thức *getInstance()*. Chúng ta có một phương thức khởi tạo *Singleton()* được đặt ở chế độ private, nó không thể được sử dụng bên ngoài lớp này. Ngoài ra, lớp Singleton còn có định nghĩa của các phương thức thể hiện khác *instanceMethod1*, *instanceMethod2*.

Trên đây là sơ đồ chung của một lớp sử dụng mẫu thiết kế Singleton, Hình dưới cho ta một ví dụ cụ thể của một lớp sử dụng Singleton. Ở đây, chúng ta xem xét ví dụ về lớp *Calendar* bao gồm các phương thức khởi tạo đặt ở chế độ private *Calendar()*. Ta có thể khởi tạo đối tượng của lớp *Calendar* bằng cách gọi phương thức *getInstance()*. Việc khởi tạo và truy cập được quản lý trực tiếp bởi chính lớp đó. Và ta có thể sử dụng thông thường như các đối tượng khác như cách khởi tạo khác.



Hình 6.13 Sơ đồ lớp Calendar



Hình 6.14 Sơ đồ lớp Calendar được sử dụng như một đối tượng

Việc thực thi mẫu thiết kế Singleton thực sự dễ dàng bằng cách định nghĩa instance và phương thức khởi tạo getInstance() với sơ đồ lớp Calendar trên như sau:

```
public class Calendar
{
    private static Calendar instance; // static means "class field"
    private Calendar()
    { // Must declare a private constructor
        // Initialize any fields here
    }
    public static Calendar getInstance()
    {
        if (instance == null)
        {
            instance = new Calendar();
        }
        return instance;
    }
    // Declare any instance fields here...
    // And now for the instance methods:
    public DateTime getFirstDay() { return firstday; }
    public int getDayLength(DateTime aDay) { return aDay.Day; }

    DateTime firstday;
    int dayLength;
}
```

Hình 6.15 Cấu trúc của lớp Calendar

Khởi tạo và sử dụng lớp Calendar như sau:

```
DateTime d = Calendar.getInstance().getFirstDay();
```

Trên thực tế có nhiều phương án khác nhau để sử dụng các mẫu thiết kế. Điều này phụ thuộc vào mục đích sử dụng cũng như yêu cầu thực tế. Trong phạm vi giáo trình này chỉ nêu một số khái niệm cơ bản về mẫu thiết kế, do việc làm chủ được các mẫu thiết kế đòi hỏi thời gian hiểu và thực hành trên các vấn đề cụ thể. Sinh viên có thể tham khảo thêm trong chương 11 của tài liệu tham khảo chính.

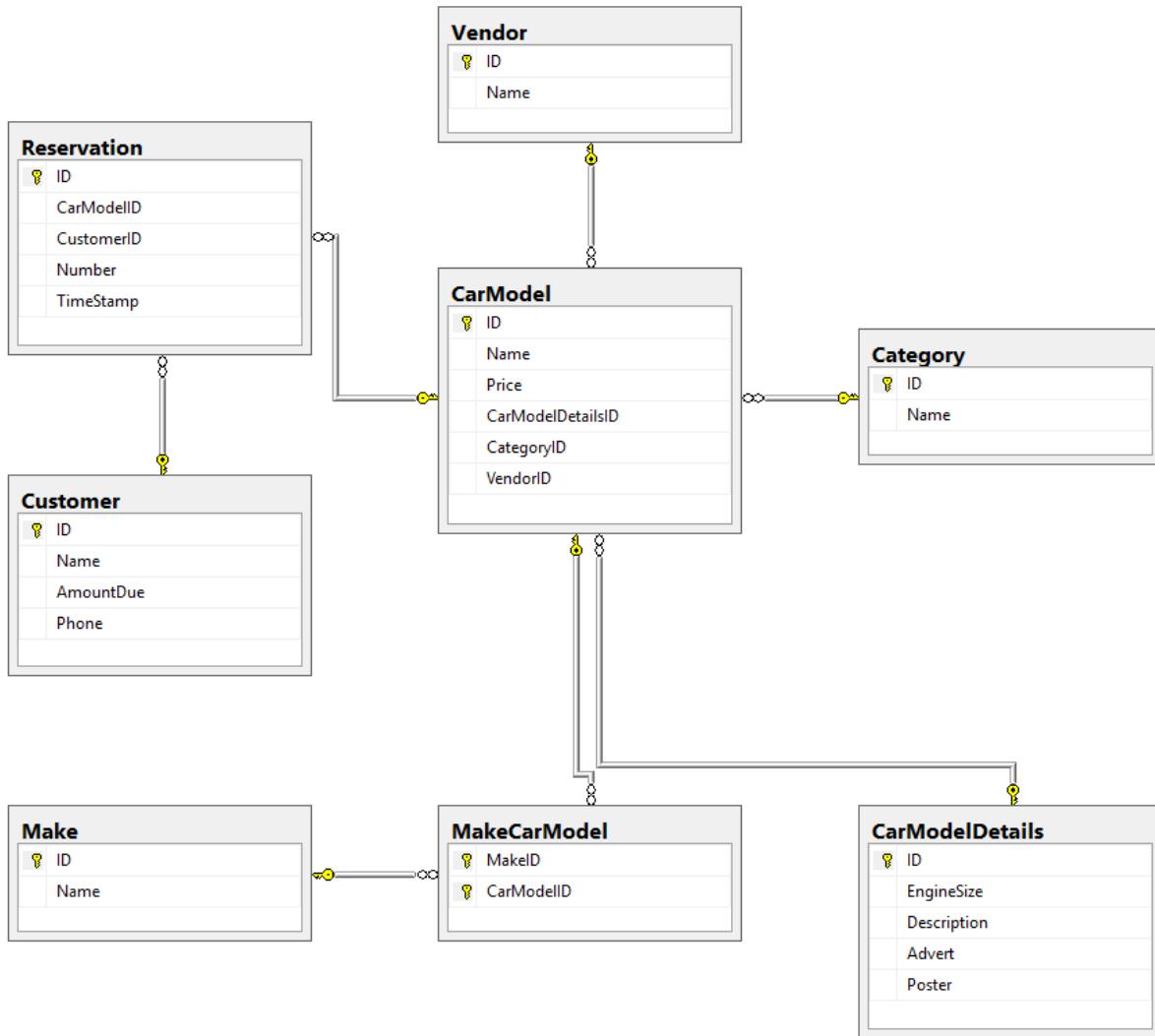
## 6.4 Ví dụ minh họa

### 6.4.1 Mô hình kiến trúc chung

Từ những ví dụ được phân tích và mô hình hóa trong các chương trước, phần này sẽ thực hiện minh họa quy trình xây dựng một chức năng của phần mềm. Ở đây ta giả định việc mọi phân tích đã hoàn thành tức là đã có sơ đồ use case, sơ đồ hoạt động, sơ đồ lớp, sơ đồ tuần tự cũng như ta đã hình thành được mô hình cơ sở dữ liệu của phần mềm (Hình

6.16). Người phát triển sẽ tập trung vào mã hóa các module của phần mềm dựa trên nền các phân tích này. Trong phần này chúng ta sẽ xem xét các Use case từ 2 và 3.

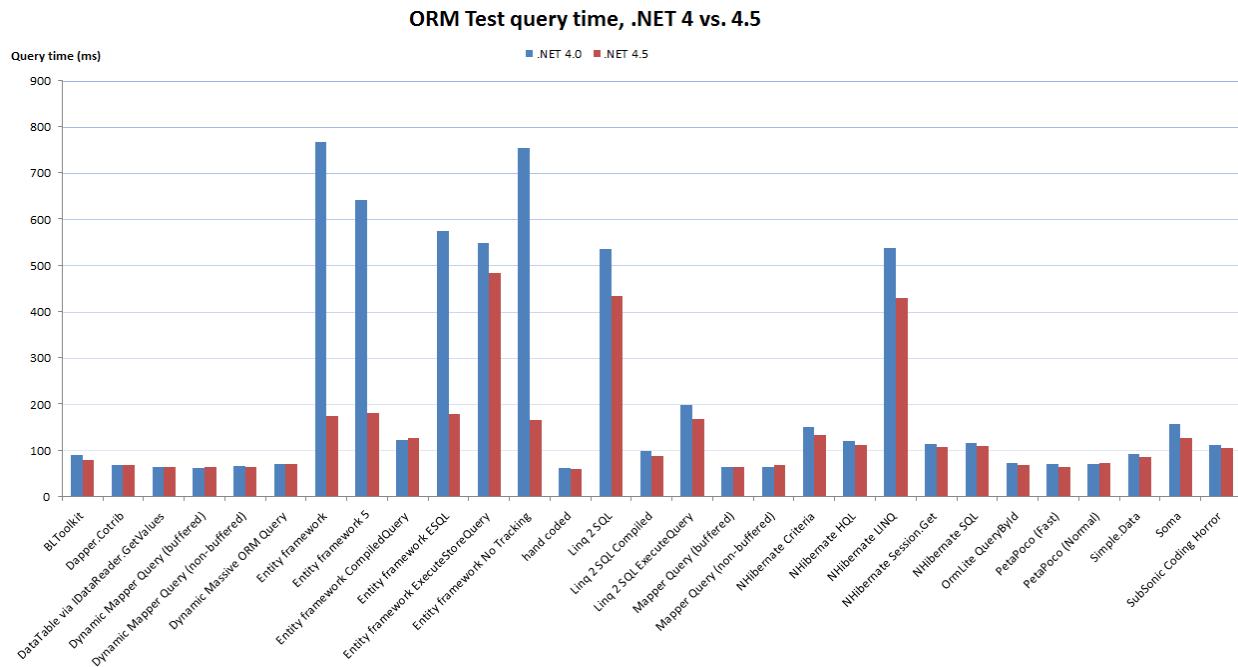
- U2: Hiển thị danh sách các mẫu xe
- U3: Hiển thị chi tiết thông tin của các mẫu xe được chọn



Hình 6.16 Một phần mô hình cơ sở dữ liệu của ví dụ minh họa

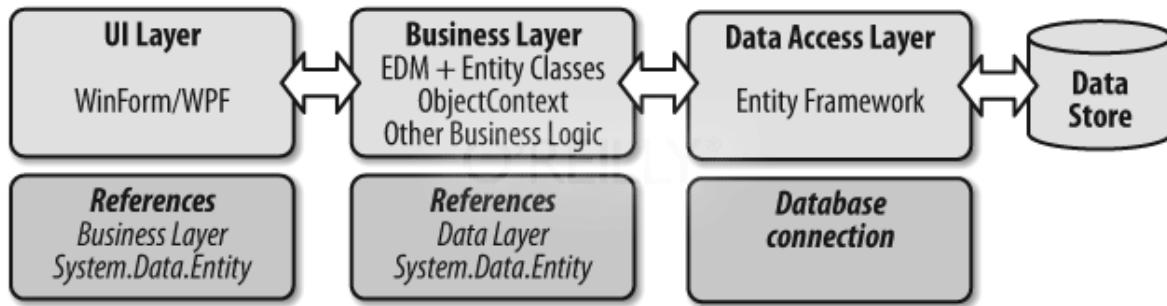
Việc xây dựng phần mềm sẽ dựa trên kiến trúc N-Tier như đã giới thiệu ở phần trước. Trước đây thông thường người phát triển sẽ xây dựng mô hình các tầng dựa trên nền tảng ADO.NET của Microsoft. Với nền tảng này, chúng ta có thể truy cập, thao tác với dữ liệu trong các hệ quản trị cơ sở dữ liệu dựa trên các đối tượng được người phát triển xây dựng trực tiếp, các tầng được phân chia một cách rạch ròi (ví dụ minh họa chi tiết về sử dụng ADO.NET trong tài liệu “Xây dựng thư viện Data Access” tác giả Đỗ Ngọc Cường). Tuy nhiên, việc này đòi hỏi mất nhiều thời gian cho việc mã hóa và thường sẽ xuất hiện nhiều

vấn đề về lỗi và bảo mật mà người phát triển không lường trước. Một xu hướng hiện nay được sử dụng khá nhiều trong các dự án là sử dụng các ORM (Object Relational Mapper) để đẩy nhanh tốc độ phát triển phần mềm. Về cơ bản tất cả các ORM đều được xây dựng dựa trên nền tảng công nghệ ADO.NET cho việc thao tác với cơ sở dữ liệu. Người phát triển có thể tận dụng tính năng tự động ánh xạ các bảng dữ liệu thành và quan hệ giữa các bảng này bằng các đối tượng tương ứng trên ứng dụng cũng như hỗ trợ mạnh mẽ các thao tác CRUD (tạo, đọc, cập nhật và xóa) trên dữ liệu. Đồng thời, người phát triển có thể truy vấn trực tiếp dữ liệu thay vì viết những hàm, thủ tục truy vấn trên các hệ quản trị bảng cách sử dụng ngôn ngữ được hỗ trợ mạnh mẽ hiện nay như LINQ. Một số ORM thông dụng hiện nay: Entity Framework, NHibernate, BLToolkit, Subsonic, etc



Hình 6.17 So sánh thời gian truy vấn các ORM

Trong ví dụ minh họa này, ORM Entity Framework được sử dụng cho việc thiết lập việc ánh xạ và thao tác dữ liệu. Đây là một nền tảng được phát triển bởi Microsoft giúp cho người phát triển làm việc các cơ sở dữ liệu như một mô hình ở tầng Business, có thể hiểu ở đây tầng truy cập dữ liệu bằng ADO.NET sẽ được tự động sinh ra khi ta sử dụng ORM này. Ưu điểm nổi trội là sẽ giảm thiểu khôi lượng mã hóa mà người phát triển phải thực hiện khi xây dựng tầng truy cập dữ liệu. Phiên bản đầu tiên của Entity Framework ra đời vào năm 2008 dưới nền tảng .NET Framework 3.5 cho đến hiện nay là phiên bản 6.0.



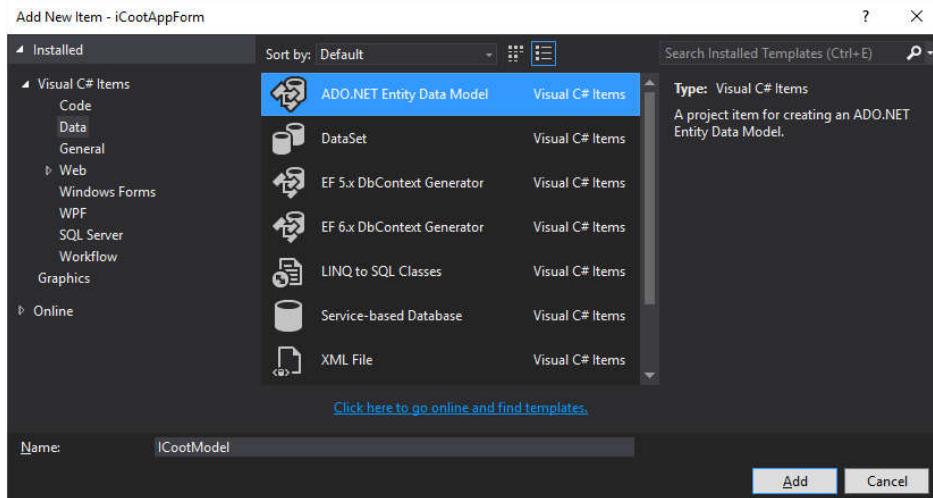
Hình 6.18 Kiến trúc ứng dụng khi sử dụng Entity Framework

Với Entity Framework, có 3 cách tiếp cận chính cho việc phát triển:

- Database First: Trong cách tiếp cận này cơ sở dữ liệu của phần mềm đã được phân tích và xây dựng trong các hệ quản trị cơ sở dữ liệu. Mọi việc chỉ bắt đầu từ việc chọn kết nối và sinh ra tự động tầng truy cập dữ liệu và một phần tầng Business. Trong giáo trình này sẽ minh họa bằng cách tiếp cận này.
- Model First: Cách tiếp cận này tạo ra các mô hình cơ sở dữ liệu dựa trên các công cụ trực quan được cung cấp sẵn trong bộ Microsoft Visual Studio. Sau khi chọn kết nối và biên dịch, cơ sở dữ liệu sẽ được sinh trong hệ quản trị cơ sở dữ liệu.
- Code First: Cách tiếp cận này thường dùng cho các mô hình cơ sở dữ liệu đơn giản. Hai cách tiếp cận trên, cơ sở dữ liệu là điểm đầu tiên được quan tâm xây dựng. Trong mô hình Code first thì các đối tượng trong tầng Business sẽ được tạo ra đầu tiên sau đó ánh xa xuống dưới để tạo ra mô hình cơ sở dữ liệu.

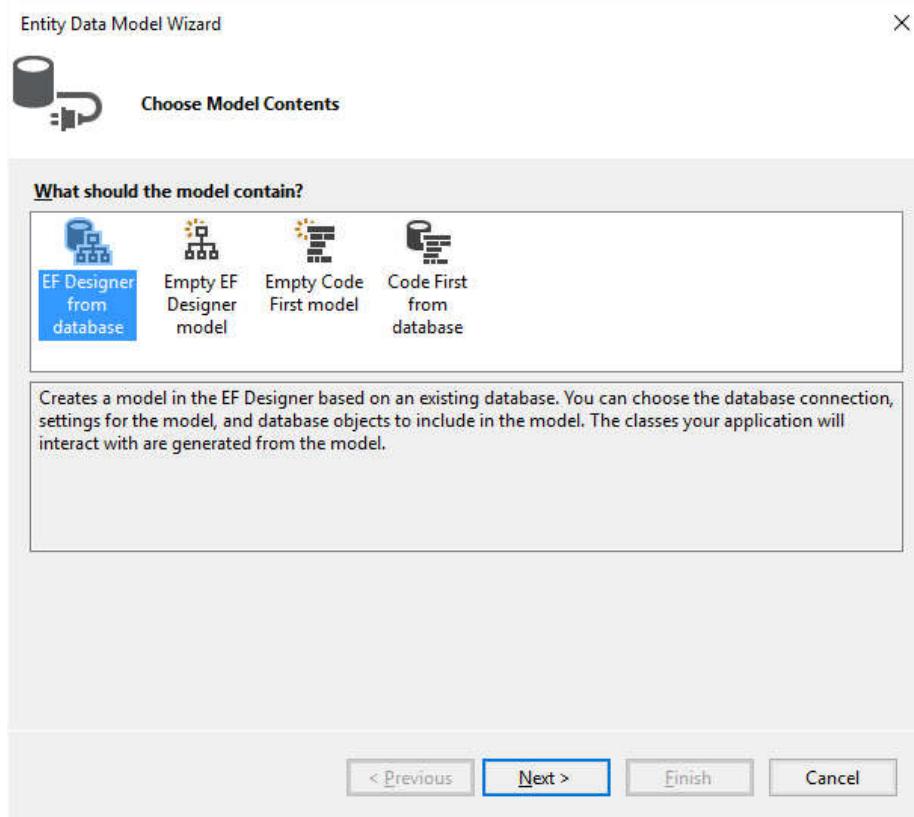
#### 6.4.2 Kết nối và thao tác dữ liệu

Dựa trên cơ sở dữ liệu được tạo ra, chúng ta sẽ kết nối trực tiếp trên bộ công cụ phát triển Visual Studio, để tạo mô hình Entity Framework mới ta sẽ thêm mới một item (Add-> New Item) chọn ADO.NET Entity Data Model trong tab Data trong màn hình Wizard hỗ trợ như hình dưới:



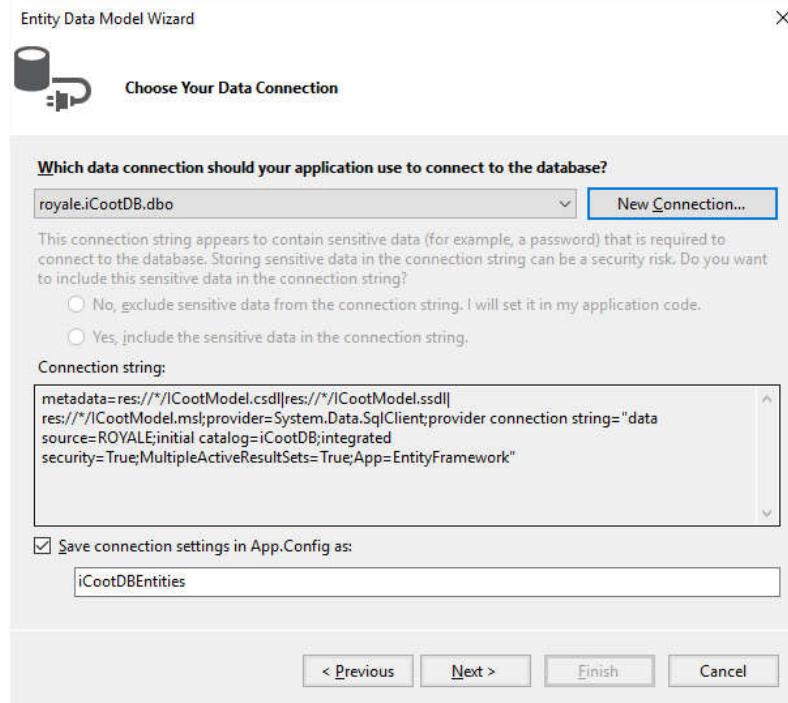
Hình 6.19 Chọn Database First để kết nối tới cơ sở dữ liệu đã có

Đến giai đoạn này, ta chọn EF Designer from database để kết nối tới cơ sở dữ liệu sẵn có:



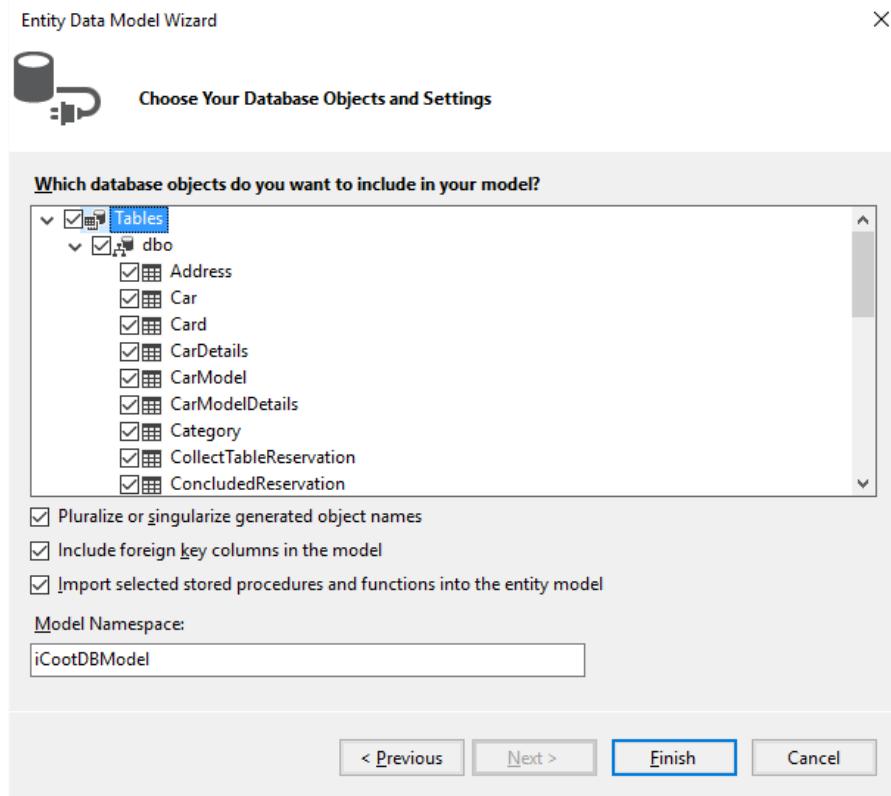
Hình 6.20 Chọn để kết nối tới cơ sở dữ liệu

Sau khi chọn cơ sở dữ liệu và kết nối thành công, ta có màn hình Wizard hiển thị các thông số kết nối được thêm vào file cấu hình app.config:



Hình 6.21 Thông tin kết nối

Màn hình tiếp theo, sau khi kết nối thành công, ta sẽ chọn các bảng dữ liệu cần ánh xạ vào mô hình trong tầng Business. Ở đây ta có thể chọn bảng, View và các hàm Store procedure:



Hình 6.22 Chọn các bảng dữ liệu kết nối

Đến thời điểm này trong dự án sẽ xuất hiện tập tin với đuôi .EDMX chứa mô phỏng trực quan mỗi quan hệ giữa các bảng được ánh xạ thành các đối tượng. Tuy nhiên ta có thể chỉnh sửa các thuộc tính của các đối tượng hoặc thêm vào định nghĩa các thuộc tính mới. Xem xét kỹ các thành phần sinh ra cùng với .EDMX ta sẽ thấy một loạt các tập tin được sinh ra cùng. Trong đó, tập tin ICootModel.Context.cs được xem là nơi đặt các khai báo các đối tượng được ánh xạ đến các bảng dữ liệu thực tế. Các đối tượng dữ liệu được định nghĩa thành các tập tin với tên tương ứng với tên bảng dữ liệu trong cơ sở dữ liệu. Các thuộc tính của đối tượng chính là các cột của bảng dữ liệu. Đối với các mối quan hệ khóa ngoại một-nhiều, một-một, nhiều-nhiều cũng sẽ được định nghĩa trong các đối tượng này. Trong hình dưới ta có thể thấy các quan hệ khóa ngoại được định nghĩa tương ứng với các khai báo *virtual*.



Hình 6.23 Các tập tin được sinh cùng tập tin .EDMX

```
public partial class CarModel
{
    public CarModel()
    {
        this.Reservations = new HashSet<Reservation>();
        this.Makes = new HashSet<Make>();
    }

    public int ID { get; set; }
    public string Name { get; set; }
    public Nullable<decimal> Price { get; set; }
    public Nullable<int> CarModelDetailsID { get; set; }
    public Nullable<int> CategoryID { get; set; }
    public Nullable<int> VendorID { get; set; }

    public virtual CarModelDetail CarModelDetail { get; set; }
    public virtual Category Category { get; set; }
    public virtual Vendor Vendor { get; set; }
    public virtual ICollection<Reservation> Reservations { get; set; }
    public virtual ICollection<Make> Makes { get; set; }
}
```

Hình 6.24 Một lớp được sinh ra tự động dựa trên bảng dữ liệu thực tế

```
public partial class iCootDBEntities : DbContext
{
    public iCootDBEntities()
        : base("name=iCootDBEntities")
    {
    }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        throw new UnintentionalCodeFirstException();
    }

    public DbSet<Address> Addresses { get; set; }
    public DbSet<Car> Cars { get; set; }
    public DbSet<Card> Cards { get; set; }
    public DbSet<CarDetail> CarDetails { get; set; }
    public DbSet<CarModel> CarModels { get; set; }
    public DbSet<CarModelDetail> CarModelDetails { get; set; }
    public DbSet<Category> Categories { get; set; }
    public DbSet<CollectTableReservation> CollectTableReservations { get; set; }
    public DbSet<ConcludedReservation> ConcludedReservations { get; set; }
    public DbSet<Customer> Customers { get; set; }
    public DbSet<DisplayableReservation> DisplayableReservations { get; set; }
    public DbSet<InternetAccount> InternetAccounts { get; set; }
    public DbSet<Make> Makes { get; set; }
    public DbSet<Member> Members { get; set; }
    public DbSet<NeedingRewealReservation> NeedingRewealReservations { get; set; }
    public DbSet<NonMember> NonMembers { get; set; }
    public DbSet<NotifiableReservation> NotifiableReservations { get; set; }
    public DbSet<Rental> Rentals { get; set; }
    public DbSet<RentalCar> RentalCars { get; set; }
    public DbSet<Reservation> Reservations { get; set; }
    public DbSet<sysdiagram> sysdiagrams { get; set; }
    public DbSet<Vendor> Vendors { get; set; }
    public DbSet<WaitingReservation> WaitingReservations { get; set; }
}
```

Hình 6.25 Lớp định nghĩa các ánh xạ với cơ sở dữ liệu

Đến bước này, ta xem như tầng truy xuất dữ liệu đã được thực hiện hoàn chỉnh với sự trợ giúp của Entity Framework. Tiếp theo ta sẽ thực hiện các thao tác dữ liệu có thể được thực hiện trong tầng Business như lấy tất cả dữ liệu, lấy theo khóa, thêm, xóa sửa và tìm kiếm etc.

Cụ thể ta sẽ tạo một lớp CarModel mới theo dạng tích hợp tức là partial để thuận tiện cho các thao tác dữ liệu trên cùng đối tượng khai báo của dữ liệu. Đây là trường hợp đơn giản nên ta dùng luôn dạng partial, tuy nhiên trường hợp kết hợp nhiều đối tượng khác nhau ta phải viết ở một lớp khai báo riêng biệt không viết theo dạng partial này.

```
public CarModel Single(int id)
{
    using (iCootDBEntities db = new iCootDBEntities())
    {
        CarModel carModel = db.CarModels.Find(id);
        return carModel;
    }
}
```

Hình 6.26 Lấy dữ liệu theo khóa

```
public List<CarModel> All()
{
    using (iCootDBEntities db = new iCootDBEntities())
    {
        List<CarModel> carModels = db.CarModels.ToList();
        return carModels;
    }
}
```

Hình 6.27 Lấy tất cả dữ liệu trong bảng

```
public int Add(CarModel carModel)
{
    using (iCootDBEntities db = new iCootDBEntities())
    {
        db.CarModels.Add(carModel);
        return db.SaveChanges();
    }
}
```

Hình 6.28 : Thêm một dòng dữ liệu mới vào cơ sở dữ liệu

```
public int Update(CarModel updateCarModel, CarModel oldCarModel)
{
    using (iCootDBEntities db = new iCootDBEntities())
    {
        oldCarModel = db.CarModels.Find(oldCarModel.ID);
        oldCarModel.Name = updateCarModel.Name;
        return db.SaveChanges();
    }
}
```

Hình 6.29 Cập nhật dữ liệu

```
public int Delete(int id)
{
    using (iCootDBEntities db = new iCootDBEntities())
    {
        CarModel carModel = this.Single(id);
        db.CarModels.Remove(carModel);
        return db.SaveChanges();
    }
}
```

Hình 6.30 Xóa một dòng dữ liệu

### 6.4.3 Thực thi một số Use case

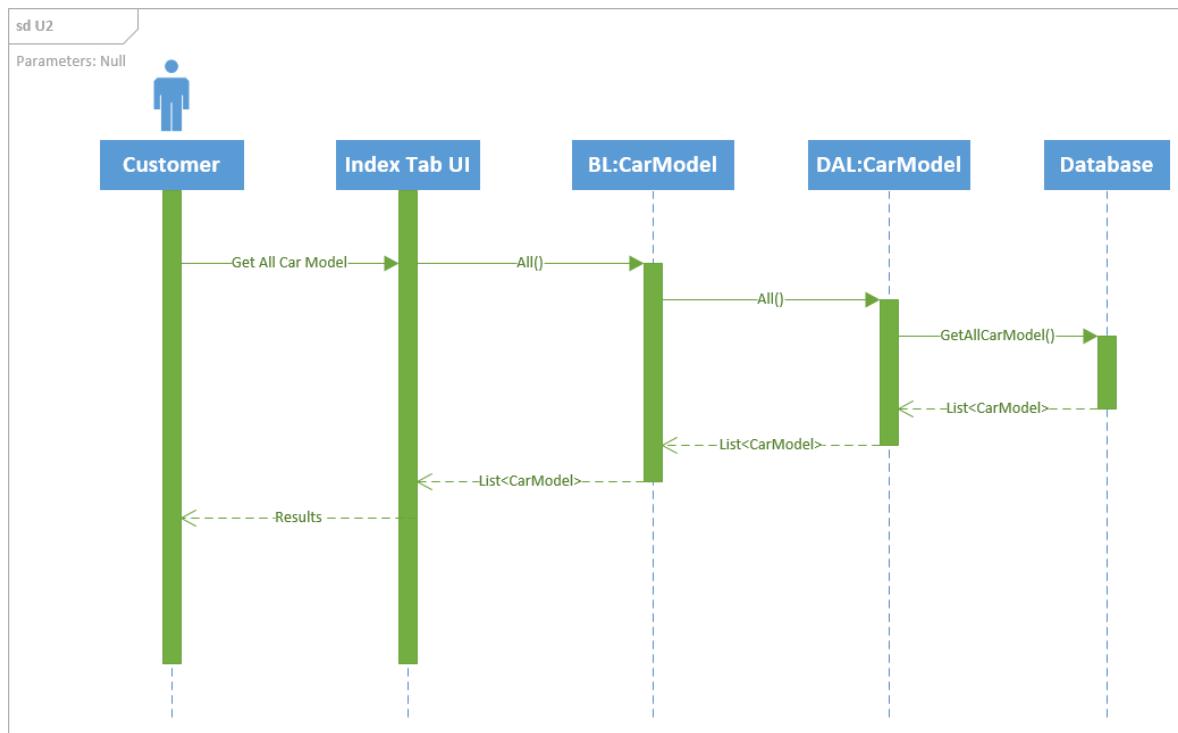
#### 6.4.3.1 Use case 02: Xem danh sách mẫu xe

Điều kiện tiên quyết: Không

1. Hệ thống trình bày cho khách hàng với các mẫu xe
2. Mở rộng sang use case 03

Điều kiện đầu ra: Không

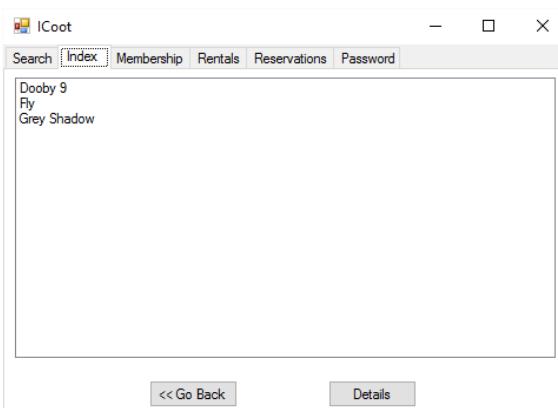
Trong use case này, danh sách mẫu xe sẽ được tải lên khi người dùng chọn vào Tab Index trên giao diện chính. Dữ liệu đơn thuần được lưu trữ trên bảng CarModel. Do vậy ta cần truy cập qua đối tượng CarModel ở các tầng Business và tầng truy cập dữ liệu để lấy dữ liệu.



Hình 6.31 Sơ đồ tuần tự Use Case 02

```
private void LoadData()
{
    CarModel carModel = new CarModel();
    List<CarModel> all = carModel.All();
    foreach (CarModel i in all)
        lbIndex.Items.Add(i.Name);
}
```

Hình 6.32 Mã thực thi ở tầng giao diện



Hình 6.33 Giao diện chính khi load các mẫu xe

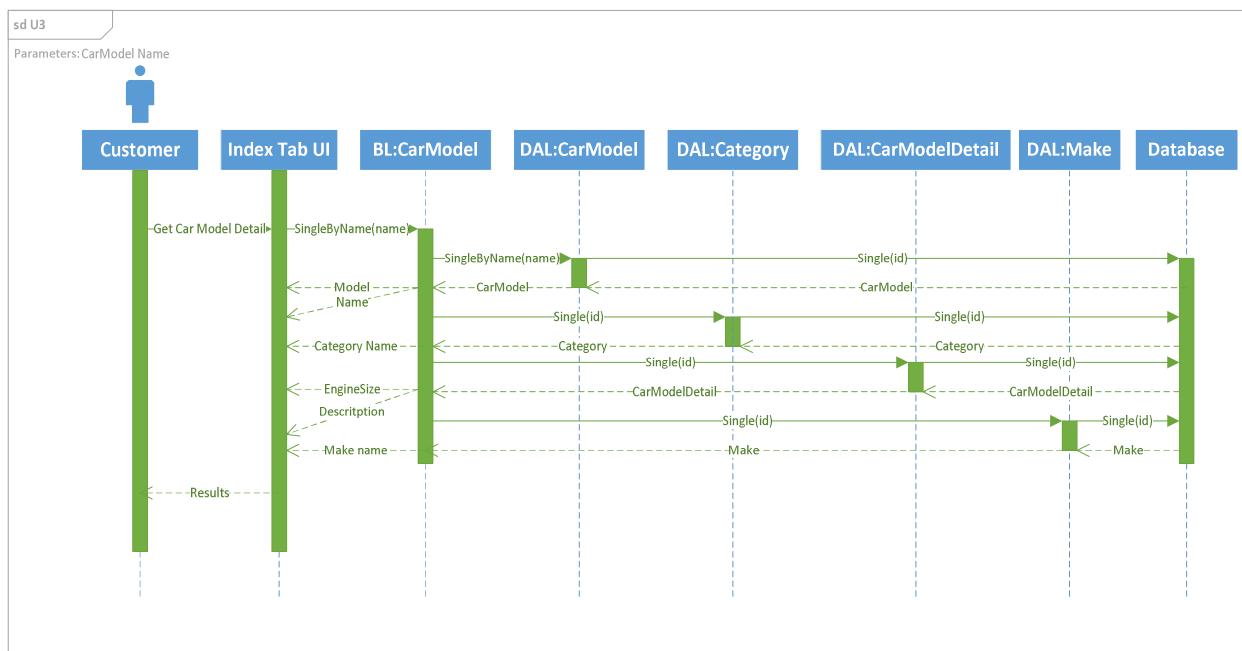
### 6.4.3.2 Use case 03: Xem chi tiết mẫu xe

Điều kiện tiên quyết: không

1. Khách hàng chọn một trong các mẫu xe
2. Khách hàng yêu cầu xem chi tiết
3. Hệ thống hiện thị thông tin chi tiết

Điều kiện đầu ra: thông tin chi tiết của mẫu xe

Trong use case này, thông tin chi tiết của mẫu xe sẽ được lấy từ nhiều bảng dữ liệu khác nhau. Ở đây thông tin về tên danh mục lấy từ bảng danh mục, thông tin về kích thước máy và mô tả được lấy từ bảng chi tiết mẫu xe và thông tin về nhà sản xuất được lấy từ bảng Make.



Hình 6.34 Sơ đồ tuần tự Use Case 03

```
public CarModel SingleByName(string name)
{
    using (iCootDBEntities db = new iCootDBEntities())
    {
        CarModel carModel = db.CarModels.Where(p=>p.Name== name).FirstOrDefault();
        Category category = new Category();
        category = category.Single(carModel.CategoryID.Value);
        carModel.Category = category;

        CarModelDetail carModelDetail = new CarModelDetail();
        carModelDetail = carModelDetail.Single(carModel.CarModelDetailsID.Value);
        carModel.CarModelDetail = carModelDetail;

        Make make = new Make();
        make = make.Find(carModel.ID);
        carModel.Makes.Add(make);
        return carModel;
    }
}
```

Hình 6.35 Mã hóa ở tầng Business trong lớp CarModel

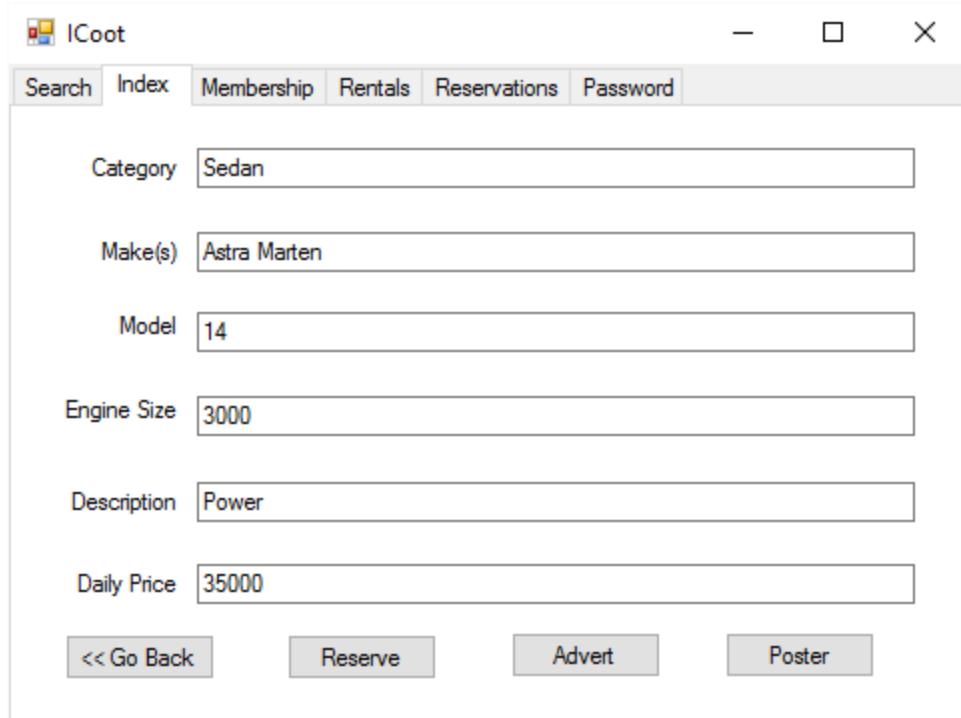
```
private void btDetails_Click(object sender, EventArgs e)
{
    if (lbIndex.SelectedItem != null)
    {
        String name = lbIndex.SelectedItem.ToString();
        CarModel carModel = new CarModel();

        carModel = carModel.SingleByName(name);

        tbDailyPrice.Text = carModel.Price.ToString();
        tbModel.Text = carModel.Name;
        tbCategory.Text = carModel.Category.Name;
        tbEngineSize.Text = carModel.CarModelDetail.EngineSize.ToString();
        tbDescription.Text = carModel.CarModelDetail.Description;
        tbModel.Text = carModel.ID.ToString() ;
        tbMakes.Text = carModel.Makes.FirstOrDefault().Name;

        pnDetails.Show();
        pnList.Hide();
        pnDetails.Refresh();
    }
}
```

Hình 6.36 Mã hóa ở tầng ứng dụng



Hình 6.37 Giao diện chi tiết thông tin của mẫu xe

## 6.5 Kết chương

### Tài liệu tham khảo

1. Microsoft, “Microsoft Application Architecture Guide 2<sup>nd</sup> Edition”, 2009, Microsoft Corporation
2. Ian Sommerville, “Software Engineering 9<sup>th</sup> edition”, 2011, Addison-Wesley
3. Scott Chacon, Ben Straub, “Pro Git”, 2014, Press
4. Lance Hunt , “C# Coding Standards for .NET”, 2007
5. Judith Bishop, “C# 3.0 Design Patterns”, 2007, O'Reilly

6. Mike O'Docherty, “*Object-Oriented Analysis & Design: Understanding System Development with UML 2.0*”, 2005, John Wiley & Sons
7. Rahul Rajat Singh, “*Mastering Entity Framework*”, PACKT Publishing, 2015

## CHƯƠNG 7. KIỂM THỬ PHẦN MỀM

### 7.1 Giới thiệu

Phát triển phần mềm là một công việc phức tạp. Mặc dù người lập trình đã rất cố gắng nhưng cũng không thể loại bỏ tất cả các lỗi từ các công đoạn xác định yêu cầu, phân tích, thiết kế và cài đặt. Tuy nhiên, với các trải nghiệm trong quá trình phát triển và thông qua giai đoạn kiểm thử riêng, ta có thể loại trừ tất cả các lỗi trước khi đưa ứng dụng vào thực tế.

Ngày nay, việc kiểm thử phần mềm được thực hiện bởi nhiều người khác nhau trong các giai đoạn phát triển phần mềm: các phát triển viên, các đồng nghiệp (không liên quan trực tiếp đến dự án), khách hàng, quản lý dự án, các kiểm thử viên (chịu trách nhiệm chính trong giai đoạn kiểm thử). Có 3 giai đoạn trong kiểm thử: trong quá trình phát triển (được thực hiện bởi các phát triển viên), trong quá trình kiểm thử (được thực hiện bởi các kiểm thử viên) và giai đoạn sau khi đưa vào ứng dụng, khi các nhà phát triển tập hợp các phản hồi từ người sử dụng và đưa ra các bản vá lỗi khi phần mềm đang hoạt động.

Chương này hướng đến việc loại bỏ các lỗi trong giai đoạn phát triển (được thực hiện bởi các phát triển viên), nó có tên là test-driven development (TDD), là một phương thức làm việc, hay một quy trình viết mã hiện đại. Đó là dạng kiểm thử liên tục, nơi các nhà phát triển thường xuyên kiểm tra chương trình của họ. Những ưu điểm của TDD là:

- Cải thiện chất lượng của phần mềm.
- Giảm chi phí của giai đoạn kiểm thử.
- Giảm số lượng các lỗi do việc liên kết giữa các lập trình viên với các kiểm thử viên.
- Giúp các lập trình viên cấu trúc (tổ chức) lại các đoạn mã của họ.
- Cho thấy các lập trình viên đang hướng vào thực tế của ứng dụng (thay vì chỉ sản xuất dòng mã).

#### 7.1.1 Thuật ngữ “Kiểm thử”

“Kiểm thử” (testing) là kiểm tra một số khía cạnh của phần mềm xem đã chính xác chưa? chẳng hạn như “kiểm tra khi nào người sử dụng có thể đăng nhập hệ thống” hoặc “kiểm tra dữ liệu được tải lên (upload) / lấy về (download) có vượt quá 500 MB tối đa cho phép không?”.

“Lỗi” (error) là những vấn đề mà lập trình viên mắc phải trong quá trình phát triển phần mềm. Chẳng hạn, một lỗi về xác định yêu cầu có thể dẫn đến sai lầm về thiết kế và càng sai khi lập trình theo thiết kế này. Lỗi là nguyên nhân dẫn đến sai.

“Sai” (fault) là một đoạn mã được thiết kế bị sai do lỗi. Sai có thể khó bị phát hiện. Khi nhà thiết kế mắc lỗi bỏ sót trong quá trình thiết kế, sai kết quả từ lỗi là do thiếu mất cái gì đó mà lẽ ra cần phải có. Sai về nhiệm vụ xuất hiện khi vào sai thông tin, còn sai về bỏ quên xuất hiện khi không vào đủ thông tin. Loại sai thứ hai khó phát hiện và khó sửa hơn loại sai thứ nhất.

“Thất bại” (failure) xuất hiện khi một lỗi được thực thi gây trực tiếp của hệ thống, thường được gây ra bởi một hoặc nhiều “sai”.

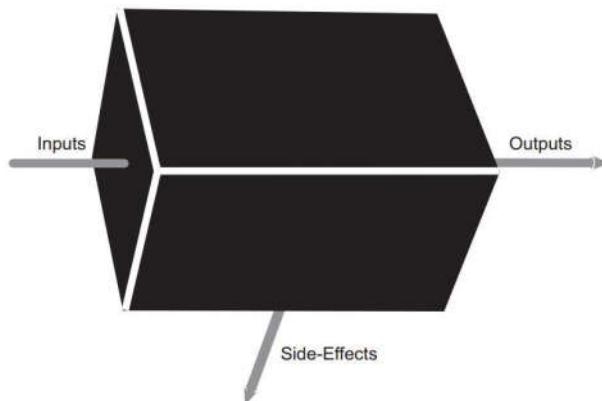
“Sửa lỗi” (fix) là khi có một lỗi được sửa chữa.

“Xác minh” (validation) là kiểm tra tính đáp ứng yêu cầu của phần mềm, nghĩa là kiểm tra việc thực hiện các chức năng mà khách hàng cần.

“Đặc tả” (specification) là một mô tả giao diện của một đoạn mã, trong đó sẽ mô tả dữ liệu vào và dữ liệu xuất. Do đó, kỹ thuật kiểm thử “đặc tả” là một thuật ngữ chung cho black-box testing và use case testing.

### 7.1.2 Black-Box Testing (kiểm thử hộp đen)

Với black-box testing, bất cứ điều gì đang được thử nghiệm (hệ thống, hệ thống con, lớp học, hay phương thức) được coi là một đối tượng bất khả xâm phạm (xem hình 7.1). Đầu vào của hộp đen là dữ liệu vào như các thông điệp, thông số, .... Vì vậy, tất cả các thử nghiệm có thể làm để đánh giá tính đúng đắn của hộp đen là nắm bắt và phân tích các dữ liệu xuất cũng như kiểm tra tác dụng phụ bên ngoài (như việc tạo ra các mục trong một cơ sở dữ liệu) và kiểm tra thời gian thực hiện để trả lời các yêu cầu hoặc thực hiện các lệnh.



Hình 7.1 Black-box testing

Black-box testing xuất phát từ triết lý của “không quan tâm làm thế nào đoạn mã đạt được mục đích của nó, miễn là nó đạt được”, điều này phù hợp với ý tưởng “các yêu cầu của hệ thống được viết ra trước khi phần mềm được tạo ra” là các yếu tố quan trọng của phát triển phần mềm: trong giai đoạn xác định yêu cầu, các nhà phân tích, nhà thiết kế và lập trình viên tự do trong việc khóa lắp đầy đủ các yêu cầu theo cách mà họ thấy phù hợp. Với black-box testing, các kiểm thử viên sẽ không quan tâm đến sự phức tạp bên trong của phần mềm.

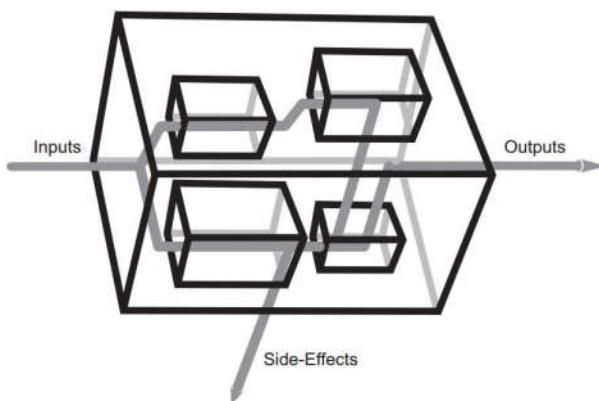
Một bất lợi của black-box testing là do cấu trúc bên trong và các đoạn mã của phần mềm được bỏ qua nên các cải tiến tiềm năng có thể bị bỏ qua – Do vậy, chương trình được tạo ra chỉ đạt về độ chính xác và đầy đủ mà có thể là chưa tối ưu nhất. Một nhược điểm nữa là các “thiếu sót” không được xác định khi kiểm tra sẽ tiếp tục chưa được phát hiện cho đến mãi sau.

Các thuật ngữ khác của black-box testing là behavioral testing (kiểm thử hành vi) hoặc functional testing (kiểm thử chức năng), chỉ kiểm tra kết quả của hộp đen mà không quan tâm đến làm thế nào để đạt được kết quả.

### 7.1.3 White-Box Testing (kiểm thử hộp trắng)

Với white-box testing, các kiểm thử viên được phép nhìn vào chiếc hộp, kiểm tra cấu trúc bên trong và các đoạn mã của phần mềm (hình 7.2).

Những thuận lợi của white-box testing: các đồng nghiệp (nhà phát triển phần mềm khác) có thể đề xuất các cải tiến, chẳng hạn như tái cấu trúc để nâng hiệu suất tốt hơn, dễ bảo trì hơn, nhiều cơ hội để tái sử dụng. Một lợi ích khác là các “thiếu sót” có thể được phát hiện sớm như lỗi lập trình, chưa đáp ứng đầy đủ hoặc không chính xác các yêu cầu.



Hình 7.2 White-box testing

Trong quá trình phát triển, cấu trúc và nội dung của các đoạn mã sẽ được thay đổi, do vậy, white-box testing thường được sử dụng trong giai đoạn kiểm thử. Các thuật ngữ khác

của white-box testing là structural testing (kiểm thử cấu trúc) hoặc glass-box testing (kiểm thử phản chiếu), phản ánh các cấu trúc bên trong của chiếc hộp cũng sẽ được quan tâm như kiểm tra kết quả đầu ra.

## 7.2 Các mức kiểm thử phần mềm

Hầu hết các chuyên gia kiểm thử đều chia quá trình kiểm thử nghiệm các mức sau:

- Unit testing (kiểm thử hàm) là mức thấp nhất. Một đơn vị là một đoạn mã hoặc 1 hàm độc lập.
- Integration testing (kiểm thử tích hợp) là mức kiểm thử tính tích hợp của các đoạn mã hay các hàm độc lập.
- System testing (kiểm thử hệ thống) là mức kiểm tra hoạt động và sự tương tác của tất cả các hệ thống con khi làm việc cùng nhau trong hệ thống lớn sau khi được tích hợp.
- Use case testing (kiểm thử ca sử dụng) là một dạng đặc biệt của kiểm thử hệ thống bởi vì hệ thống dùng các ca sử dụng (use case) để mô tả hoàn toàn những gì hệ thống có khả năng thực hiện.

### 7.2.1 Unit testing (kiểm thử hàm)

Đối với mỗi lớp, các bài kiểm tra sẽ được thiết kế để kiểm tra tính chính xác, hiệu quả và dễ sử dụng. Trong thiết kế các bài kiểm tra, các dữ kiện như tên, kiểu dữ liệu, loại thông số và tên thông số sẽ được quan tâm cũng như kiểm tra tính đúng của kết quả với dữ liệu đầu vào. Đối với black-box testing, các trạng thái sẽ được xác minh bằng các thông điệp (dữ liệu đầu vào và dữ liệu xuất); với white-box testing, các trường sẽ được kiểm tra về kiểu dữ liệu, cách sử dụng để xác định các sai sót.

Với unit testing, sẽ có tình huống khó xử: làm thế nào kiểm tra các đơn vị (đoạn mã, hàm) độc lập mà không kiểm tra các đơn vị (đoạn mã, hàm) liên quan đến nó? Trong hướng đối tượng, các đối tượng được đóng gói đơn giản chỉ là một phần của các đơn vị và do đó không cần có các bước đặc biệt để kiểm tra hộp đen, đối với white-box testing các đơn vị (đoạn mã, hàm) liên quan đến nó có thể được bỏ qua, các đơn vị (đoạn mã, hàm) liên quan đến nó có thể được thay thế bằng các mô phỏng.

### 7.2.2 Integration Testing (Kiểm thử tích hợp)

Khi một hệ thống được hoàn chỉnh, mỗi đối tượng sẽ kết hợp với các đối tượng khác, vì vậy integration testing là một bước cần thiết sau khi thực hiện unit testing (đặc biệt là khi các lớp khác nhau thường được viết bởi các nhà phát triển khác nhau). Kiểm thử tích

hợp nhằm đảm bảo hệ thống làm việc ổn định trong môi trường thí nghiệm để sẵn sàng cho việc đưa vào môi trường thực sự bằng cách đặt các đơn vị với nhau theo phương pháp tăng dần.

Integration testing thường diễn ra ở một hoặc nhiều mức sau:

- Các lớp kết hợp nhau tạo thành một đơn vị logic.
- Lớp (layer)
- Gói (package)
- Thư viện (library)
- Khung chương trình (framework)
- Hệ thống hoặc hệ thống con (system or subsystem)

Do vậy, System hoặc subsystem testing là một dạng integration testing mà ở đó việc tích hợp các lớp sẽ tạo nên một hệ thống con.

### 7.2.3 Use Case Testing (Kiểm thử ca sử dụng)

Use case testing là dạng kiểm thử hệ thống black-box. Với Use case testing, từng ca sử dụng lần lượt sẽ được kiểm thử, các bài kiểm thử được thiết kế để xác nhận rằng hệ thống đáp ứng được các trường hợp sử dụng. Các kịch bản cho mỗi ca sử dụng sẽ được thiết kế, mỗi kịch bản là một chuỗi các sự kiện được kết hợp với nhau.

## 7.3 Một số nguyên tắc trong kiểm thử

Mặc dù một phần mềm được thiết kế, cài đặt và kiểm tra lỗi trong giai đoạn phát triển nhưng vẫn phải cần giai đoạn kiểm thử vì:

- Chưa sử dụng các kỹ thuật như use case hoặc lược đồ UML.
- “Sai” (fault) khó được phát hiện và tăng chi phí để sửa nó về sau.
- Mọi lập trình viên đều muốn thử đoạn chương trình mà họ viết trong thời gian sớm nhất, điều này có thể làm ảnh hưởng đến hiệu quả của toàn hệ thống do chủ quan và tính đơn lẻ.

Do vậy, cần phải có một chiến lược kiểm thử liên quan đến tất cả các loại kiểm thử như: kiểm thử bằng tay, kiểm thử bằng đoạn chương trình mẫu, kiểm thử tự động,... Khi chưa dùng các kỹ thuật như use case hoặc lược đồ UML, kiểm thử thủ công sẽ được sử dụng thông qua:

- Đội ngũ phát triển, vì họ đang chuyên gia có liên quan đến sự thành công của hệ thống.
- Khách hàng, vì khách hàng là những chuyên gia cụ thể liên quan đến tính chính xác và hữu ích của hệ thống.
- Đồng nghiệp, họ là những người phát triển nhưng không tham gia trực tiếp vào dự án hiện tại, sẽ kiểm thử hệ thống một cách khách quan.

Khi đã dùng các kỹ thuật mới như use case hoặc lược đồ UML, kiểm thử được sử dụng thông qua:

- Đội ngũ phát triển, để họ có thể tiến bộ hơn (sẽ sản xuất một sản phẩm có chất lượng tốt hơn).
- Đồng nghiệp, người tham gia đánh giá chương trình, giúp phát hiện lỗi (fault) và đưa ra các ý kiến để tái cấu trúc hệ thống.
- Đội ngũ thử nghiệm, những người xác minh tính đúng đắn và xác nhận tính hiệu quả của hệ thống, để đưa ra hệ thống tốt nhất.

### 7.3.1 Kiểm thử trong quá trình phát triển

Trong thời gian phát triển, các kết quả của chương trình sẽ được kiểm tra thủ công, nếu dùng công cụ thì việc kiểm thử đã hướng vào một phương pháp cụ thể của công cụ đó, từ đó làm hạn chế việc kiểm tra tính khách quan và nhất quán của các thành phần trong hệ thống.

Đối với kiểm thử các đoạn mã chương trình, cần khuyến khích các lập trình viên sử dụng các bài kiểm thử song song với các hoạt động viết chương trình. Một framework như Junit, một framework đơn giản dùng cho việc tạo các unit testing tự động và chạy các test có thể lặp đi lặp lại, có thể hỗ trợ: trước khi viết một đoạn chương trình hệ thống, các lập trình viên viết một số mẫu để kiểm thử nó. Vì vậy, khi một thành phần mới của chương trình được hoàn tất, các lập trình viên có thể xác định ngay rằng họ đã thành công hay chưa?

Đánh giá của đồng nghiệp cũng cần được sử dụng thường xuyên vào cuối mỗi thành phần của hệ thống. Tất cả các thành phần của hệ thống cần được kiểm thử sẽ được gửi đến các thành viên của nhóm phát triển và các đồng nghiệp khác và sau đó một cuộc họp sẽ được sắp xếp để kiểm thử chúng một cách chi tiết, tìm kiếm lỗi và những bất cập. Mục đích là để xác nhận các thành phần của hệ thống đến giai đoạn là đúng trước khi tiếp tục phát triển các thành phần tiếp theo. Việc kiểm thử này phụ thuộc vào: chính sách của công ty, chính sách đối với khách hàng, chính sách của dự án,...

Để có đánh giá của khách hàng, cần có một chiến lược để yêu cầu các khách hàng kiểm tra tính đúng đắn và xác nhận đáp ứng yêu cầu. Khách hàng không nên can thiệp quá sâu vào chương trình như sơ đồ trình tự hoặc đoạn mã.

Đánh giá của đồng nghiệp và của khách hàng không nên được kết hợp, nếu không các đồng nghiệp sẽ bị ức chế bởi các nhận xét quá sâu vào chi tiết của khách hàng hoặc khách hàng sẽ phải tham gia các cuộc thảo luận kỹ thuật chi tiết.

Sau khi kiểm từ đầu tiên, cần phải có kế hoạch kiểm thử thường xuyên trong giai đoạn phát triển.

### 7.3.2 Kiểm thử trong quá trình kiểm thử

Trong giai đoạn này, trước khi đưa hệ thống vào sử dụng, đội ngũ kiểm thử sẽ đảm nhận việc đánh giá tính đúng đắn và đáp ứng yêu cầu của hệ thống, gồm:

- Chạy lại các bài kiểm thử dùng trong giai đoạn phát triển.
- Kiểm tra các hệ thống con (các lớp) dựa vào giai đoạn thiết kế và đặc tả.
- Kiểm tra cả hệ thống chủ yếu dựa vào các ca sử dụng (use case) được ghi nhận vào giai đoạn xác định yêu cầu.
- Kiểm thử tính chấp nhận của hệ thống, việc này sẽ thực hiện cùng với người sử dụng và quản trị hệ thống.
- Kiểm thử cài đặt: cài đặt trên các hệ thống khác nhau để kiểm ra tính tương thích.
- Kiểm thử tài liệu: kiểm tra các tài liệu hướng dẫn và tập huấn.
- Kiểm thử Beta.
- Thu thập số liệu

Tất cả các việc của giai đoạn này đều được thực hiện bằng kiểm thử hộp đen và kiểm thử hộp trắng.

### 7.3.3 Kiểm thử sau khi phát hành

Sau khi phát hành, việc kiểm thử hệ thống vẫn được tiếp tục. Mỗi khi người sử dụng hoặc người quản trị sử dụng các thành phần của hệ thống sẽ kiểm tra chúng một lần nữa, các lỗi xảy ra, các đề nghị cải tiến hoặc muốn có tính năng khác trong quá trình sử dụng cần được ghi nhận lại và báo cáo lại đội ngũ của dự án.

Đội ngũ của dự án sẽ phải vá các lỗi khi bị phát hiện và họ sẽ quyết định khi nào sẽ đưa bản vá lỗi vào sử dụng, các cải tiến và tính năng mới cũng sẽ được đưa vào trong giai đoạn này.

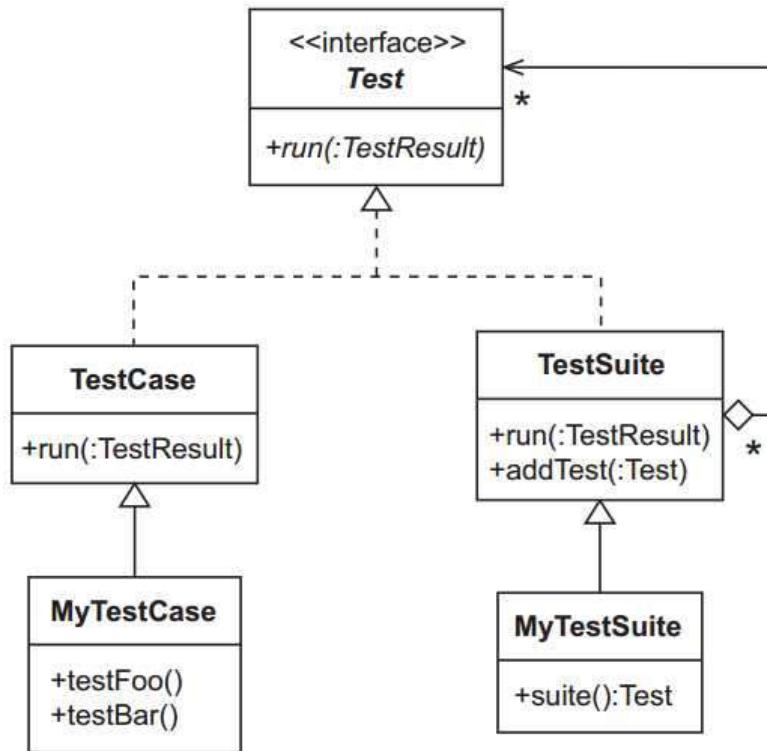
Bất kỳ sửa chữa nào được áp dụng cho hệ thống cũng nên dùng thử nghiệm hồi quy giữa người sử dụng hoặc người quản trị hệ thống và đội ngũ của dự án.

## **7.4 TEST-DRIVEN DEVELOPMENT**

### **7.4.1 Tổng quan về TDD**

TDD là một mô hình phát triển hướng về việc kiểm thử. Vào giữa những năm 1990, Kent Beck và Erich Gamma đã thiết kế một framework cho TDD có tên là Smalltalk. Được mệnh danh là SUnit, đã trở nên phổ biến đối với các ngôn ngữ lập trình: Java, C++, C #, VisualBasic, Eiffel,... từ khóa xUnit được sử dụng với x là các ký tự đại diện cho ngôn ngữ lập trình (với Java được gọi là JUnit, với C++ được gọi là CPPUNIT,...). Từ các tên đó cho biết đây là các framework kiểm thử hàm, trong đó mỗi hàm/lớp (unit) có thể là một hàm/lớp, hoặc kết hợp của các hàm/lớp.

Lõi của framework được minh họa trong hình 7.3, các thành phần cho phép chúng ta xây dựng nên một hệ thống các bài kiểm thử có thứ bậc, với các đối tượng TestSuite tại các nút và các đối tượng TestCase ở lá.



Hình 7.3 Lõi của framework

TDD được xây dựng theo hai tiêu chí: Test-First (kiểm thử trước) và Refactoring (điều chỉnh mã nguồn). Trong đó, khi một yêu cầu phần mềm (requirement) được đặt ra:

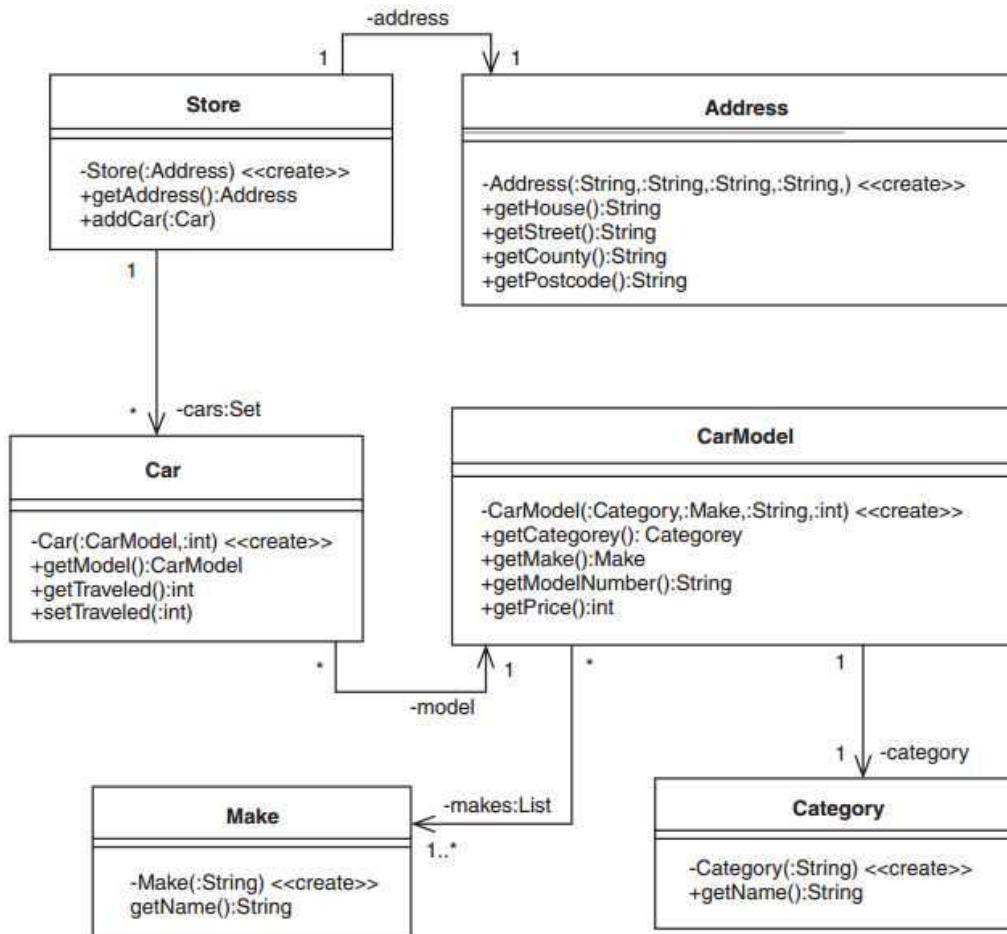
- Người developer soạn thảo kịch bản kiểm thử (test case) cho yêu cầu đó trước tiên và chạy thử kịch bản đó lần đầu tiên. Hiển nhiên, việc chạy thử sẽ đưa ra 1 kết quả thất bại vì hiện tại chức năng đó chưa được xây dựng (và thông qua kết quả đó, ta cũng kiểm tra được là kịch bản kiểm thử đó được viết đúng).
- Theo đó, dựa vào mong muốn (expectation) của kịch bản kia, người developer sẽ xây dựng một lượng mã nguồn (source code) vừa đủ để lần chạy thứ 2 của kịch bản đó thành công.
- Nếu trong lần chạy thứ 2 vẫn đưa ra 1 kết quả thất bại, điều đó có nghĩa là thiết kế chưa ổn và người developer lại chỉnh sửa mã nguồn và chạy lại kịch bản đến khi thành công.
- Khi kịch bản kiểm thử được chạy thành công, người developer tiến hành chuẩn hóa đoạn mã nguồn (base-line code) và tiếp tục hồi quy với kịch bản kiểm thử tiếp theo. Việc chuẩn hóa bao gồm thêm các chú thích, loại bỏ các dư thừa, tối ưu các biến...

### **7.4.2 Ví dụ về TDD sử dụng JUnit**

Sơ đồ các lớp được xây dựng ở giai đoạn thiết kế được thể hiện trong hình 7.4. Trong đó:

- Lớp Store chứa địa chỉ của điểm cho thuê xe và những chiếc xe có sẵn cho thuê.
- Lớp Address là một lớp địa chỉ để lưu vị trí của điểm cho thuê xe.
- Lớp Car chứa các thông tin về xe cho thuê.
- Lớp CarModel là sự bổ sung cho lớp Car, nó lưu thông tin về chủng loại cụ thể của chiếc xe mà có sẵn cho thuê.
- Lớp Category cho biết thể loại cho thấy của xe (thể thao, gia đình, sang trọng,...).
- Lớp Make cho biết tên hãng sản xuất xe.

Với mỗi lớp, các test-case tương ứng sẽ được viết để kiểm tra các lớp đó: AddressTestCase, CarModelTestCase, CategoryTestCase and MakeTestCase.



Hình 7.4 Sơ đồ lớp iCoot

#### 7.4.2.1 Kiểm tra lớp Car

Trước khi đến với công cụ mã hóa cần bắt đầu với lớp Car (xe ôtô), với kỹ thuật TDD thì các bài kiểm tra được thiết kế trước khi viết mã. Vì vậy, cần thực hiện CarTestCase đầu tiên. Viết một trường hợp thử nghiệm với JUnit sẽ đơn giản vì: các lớp mới phải kế thừa từ lớp JUnit TestCase và nó phải chứa một hoặc nhiều phương thức bắt đầu bằng các test - kiểm tra tự động tìm kiếm các phương thức tương tự và thực thi chúng như một phần của test case.

Luôn nhớ rằng mỗi phương pháp thử nghiệm nên kiểm tra một vấn đề nào đó thật kỹ, bài test đầu tiên sẽ kiểm tra việc tạo ra một loại xe ôtô. Vì vậy, CarTestCase ban đầu sẽ như sau:

```

public class CarTestCase extends TestCase {
    public void testCreate() {
    }
}
  
```

Trong phương thức testCreate, một loại xe ôtô sẽ được tạo ra, sử dụng phương thức tạo lập CarModel và số km đã đi trên đồng hồ. Việc tạo CarModel đòi hỏi phải tạo Category (hạng xe) cùng với tên của hạng xe, và một danh sách các thuộc tính Make (hãng sản xuất). Để thuận tiện, chỉ thêm một Make vào CarModel.

Sau khi viết mã cho việc tạo các đối tượng, phương thức testCreate sẽ là:

```
public void testCreate() {  
    Category category = new Category("Saloon");  
    Make make = new Make("Ford");  
    List<Make> makes = new LinkedList<Make>();  
    makes.add(make);  
    String modelNumber = "Blur 1.6";  
    int price = 30;  
    CarModel carModel = new CarModel(category, make, modelNumber, price);  
    int traveled = 234243;  
    Car car = new Car(carModel, 33445);  
}
```

Để tạo ra một loại ôtô, công việc của bài test sẽ là việc tạo ra loại xe ôtô đó có đúng không? Để làm được điều này, các thuộc tính của loại xe ôtô mới sẽ được so sánh với các thuộc tính của loại xe đã tạo. Test case đã kế thừa một số phương thức assertX từ lớp TestCase. Các phương thức được sử dụng đó là assertEquals, sẽ lấy 2 đối tượng là tham số của phương thức và kiểm tra xem chúng có giống nhau không, và assertEquals, sẽ lấy 2 thông số của đối tượng và kiểm tra xem chúng có bằng nhau không? Ví dụ như:

- assertEquals(aCar,aCar) will pass.
- assertEquals(new Object(),new Object()) will fail.
- assertEquals(new Category("Sports"),new Category("Sports")) will pass.
- assertEquals(10,11) will fail.

Do vậy, phải thêm 2 dòng lệnh sau vào cuối của testCreate:

```
assertEquals(car.getModel(), carModel);  
assertEquals(car.getTraveled(), traveled);  
}
```

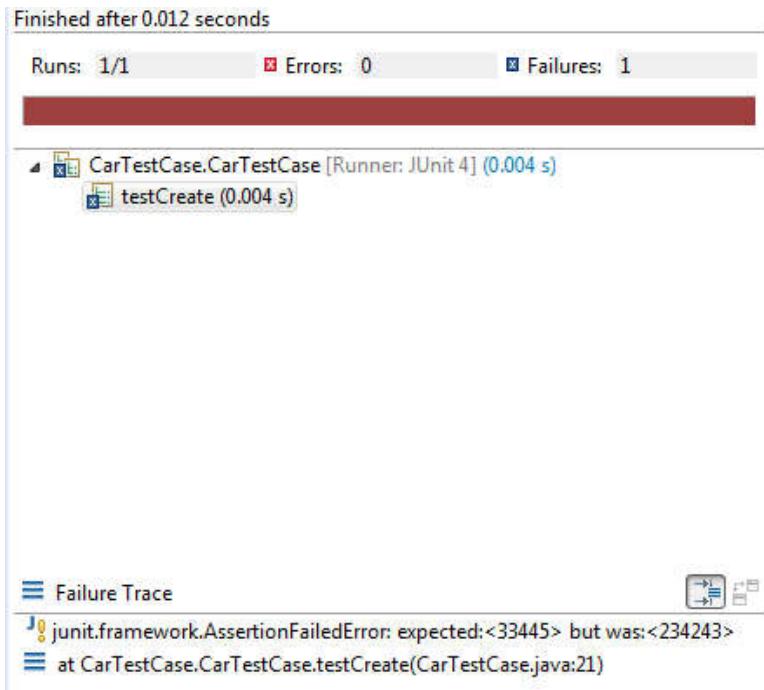
#### 7.4.2.2 Cài đặt lớp Car

Sau khi đã thực thi các thử nghiệm cho lớp Car, ta sẽ xây dựng nội dung bên trong của lớp Car. Lớp Car bao gồm: các thuộc tính mẫu xe (thuộc lớp CarModel) và quãng đường đi được (giá trị kiểu int); một phương thức tạo lập để khởi tạo các giá trị ban đầu; các phương thức getter và setter để gán (set) và thu nhận (get) các giá trị.

Lớp Car được triển khai như sau:

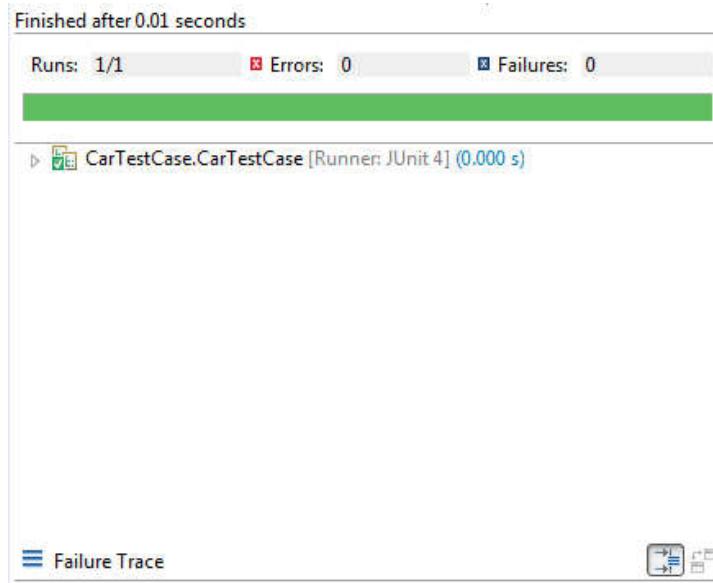
```
public class Car {  
  
    private CarModel model;  
    private int traveled;  
  
    Car(CarModel m, int t) {  
        model = m;  
        traveled = t;  
    }  
  
    public CarModel getModel() {  
        return model;  
    }  
  
    public int getTraveled() {  
        return traveled;  
    }  
  
    public void setTraveled(int t) {  
        traveled = t;  
    }  
}
```

Để chạy được kết quả thử nghiệm, click phải lên lớp *CarTestCase*, chọn Run As, chọn JUnit Test. Kết quả chạy cho thấy sự thất bại của kết quả (được thể hiện bằng màu đỏ), như hình sau:



Nhìn kỹ vào thông tin được đưa ra trong phần Failure Trace, có thể thấy lỗi xảy ra khi so sánh 2 giá trị là 33445 và 234243. Nhìn vào phần mã nguồn của phương thức *testCreate* trong lớp *CarTestCase* có thể thấy lỗi xảy ra khi có khai báo một biến cục bộ là *traveled*

nhưng không sử dụng khi khởi tạo đối tượng *car*, chính vì vậy mà phép so sánh *assertEquals* thất bại. Khi thay thế giá trị của *traveled* bởi 33445, biên dịch lại và chạy lại bộ kiểm thử, sẽ cho kết quả thành công (thanh màu xanh) như hình sau:



#### 7.4.3 Tô chức lại mã nguồn các bộ Kiểm thử

Một chu kỳ kiểm thử bao gồm: viết một kiểm thử, viết code, chạy các kiểm thử, sửa chữa các sai sót và chạy thử nghiệm. Để hoàn thành các kiểm thử cho lớp *Car*, giả sử cần thêm phương thức *testSetTraveled* vào lớp *CarTestCase* như là một bài tập. Giống như đã thực hiện trước đó với phương thức *testCreate*, phương thức *testSetTraveled* được triển khai như sau:

```
public void testSetTraveled() {
    Category category = new Category("Luxury");
    Make make = new Make("Plexus");
    List<Make> makes = new LinkedList<Make>();
    makes.add(make);
    String modelNumber = "Neo STS 3.0";
    int price = 109;
    CarModel carModel = new CarModel(category, make, modelNumber, price);
    int traveled = 32432;
    Car car = new Car(carModel, traveled);
    int newDistance = 534;
    car.setTraveled(newDistance);
    assertEquals(car.getTraveled(), newDistance);
}
```

Khi chạy thử nghiệm với JUnit, mọi thứ hoạt động tốt, có 2 kiểm thử (*testCreate* và *testSetTraveled*) đã được chạy. Tuy nhiên, có thể dễ dàng nhận thấy hai phương thức này có khá nhiều đoạn mã có thể dùng chung tài nguyên. Vì vậy có thể gom chung các thuộc

tính lại và khai báo hoặc khởi tạo trong một phương thức nào đó. Quá trình này được gọi là *refactoring*: tổ chức lại mã nguồn để cải thiện chất lượng viết code.

Các thuộc tính dùng chung có thể được khai báo và khởi tạo nó trong phương thức tạo lập, tuy nhiên, JUnit có hỗ trợ một phương thức có tên *setUp*, phương thức này được khai tự động trước khi các phương pháp kiểm thử được chạy.

Vì vậy, những gì chúng ta cần làm là khởi tạo các thuộc tính trong phương thức *setUp*:

```
protected void setUp() {
    category = new Category("Saloon");
    make = new Make("Ford");
    makes = new LinkedList<Make>();
    makes.add(make);
    modelNumber = "Blur 1.6";
    price = 30;
    carModel = new CarModel(category, make, modelNumber, price);
    traveled = 234243;
    car = new Car(carModel, traveled);
}
```

Lúc này các thuộc tính được khai báo trực tiếp trên lớp như sau:

```
private Category category;
private List<Make> makes;
private Make make ;
private String modelNumber;
private int price;
private CarModel carModel;
private int traveled;
private Car car;
```

Lúc này, hai phương thức là *testCreate* và *testSetTraveled* được viết lại một cách đơn giản hơn nhiều:

```
public void testCreate() {
    assertEquals(car.getModel(), carModel);
    assertEquals(car.getTraveled(), traveled);
}

public void testSetTraveled() {
    int newDistance = 534;
    car.setTraveled(newDistance);
    assertEquals(car.getTraveled(), newDistance);
}
```

Sau khi thực hiện các thay đổi, biên dịch và chạy thì tất cả mọi thứ vẫn hoạt động và cho ra kết quả thành công. Với phương pháp trong khuôn khổ bộ JUnit giúp đảm bảo rằng những sửa đổi vừa thực hiện là chính xác, một hình thức kiểm tra hồi quy.

Khi hoàn thành 2 lớp Car và CatestCase, các lớp sẽ như sau:

```
public class Car {  
    private CarModel model;  
    private int traveled;  
    Car(CarModel m, int t) {  
        model = m;  
        traveled = t;  
    }  
    public CarModel getModel() {  
        return model;  
    }  
    public int getTraveled() {  
        return traveled;  
    }  
    public void setTraveled(int t) {  
        traveled = t;  
    }  
}
```

```
public class CarTestCase extends TestCase {
    private Category category;
    private List<Make> makes;
    private Make make ;
    private String modelNumber;
    private int price;
    private CarModel carModel;
    private int traveled;
    private Car car;

    protected void setUp() {
        category = new Category("Saloon");
        make = new Make("Fort");
        makes = new LinkedList<Make>();
        makes.add(make);
        modelNumber = "Blur 1.6";
        price = 30;
        carModel = new CarModel(category, make, modelNumber, price);
        traveled = 234243;
        car = new Car(carModel, traveled);
    }
    public void testCreate() {
        assertEquals(car.getModel(), carModel);
        assertEquals(car.getTraveled(), traveled);
    }
    public void testSetTraveled() {
        int newDistance = 534;
        car.setTraveled(newDistance);
        assertEquals(car.getTraveled(), newDistance);
    }
}
```

#### 7.4.4 Tạo một bộ kiểm thử cho quá trình

Nhiệm vụ tiếp theo là viết các kiểm thử cho lớp Store. Như thường lệ, bắt đầu bằng việc triển khai một lớp có tên StoreTestCase. Nội dung của lớp này như sau:

```
public class StoreTestCase extends TestCase {  
    private Address address;  
    private Store store;  
    private CarModel carModel;  
    private Car car;  
  
    protected void setUp() {  
        address = new Address("9", "Ash Lane", "Greater Manchester", "SK4 3HJ");  
        store = new Store(address);  
        Category category = new Category("Vintage");  
        Make make = new Make("Mostin");  
        List<Make> makes = new LinkedList<Make>();  
        makes.add(make);  
        String modelNumber = "Wheely 1950";  
        int price = 89;  
        carModel = new CarModel(category, make, modelNumber, price);  
        int traveled = 435345;  
        car = new Car(carModel, traveled);  
    }  
    public void testCreate() {  
        assertEquals(store.getAddress(), address);  
    }  
}
```

Lớp Store được viết như sau:

```
public class Store {  
    private Address address;  
    private Set<Car> cars;  
  
    Store(Address a) {  
        address = a;  
        cars = new HashSet<Car>();  
    }  
    public Address getAddress() {  
        return address;  
    }  
    public void addCar(Car c) {  
        cars.add(c);  
    }  
}
```

Khi biên dịch và chạy kết quả của StoreTestCase, mọi thứ hoạt động tốt. Tuy nhiên, có một loạt các thử nghiệm liên quan đến các lớp cần kiểm thử như MakeTestCase, CategoryTestCase, CarModelTestCase, CarTestCase. Mỗi một bộ phải được chạy riêng rẽ, đây là một sự bất tiện khi phải chạy lần lượt, đặc biệt trong trường hợp có đổi lại mã nguồn. JUnit có hỗ trợ một đối tượng có tên test suite: là một nhóm các trường hợp kiểm thử, cho phép chạy một lần để lấy kết quả. Để xây dựng một nhóm kiểm thử nghiệm trong JUnit, cần xây dựng một lớp kế thừa từ *TestSuite* và một phương thức dạng static tên là *suite()*, phương thức này sẽ trả về tất cả các bộ kiểm. Phần mã nguồn cho phần này được triển khai như sau:

```
public class CootBusinessTestSuite extends TestSuite {  
    public static Test suite() {  
        TestSuite result = new TestSuite();  
        //result.addTestSuite(MakeTestCase.class);  
        //result.addTestSuite(CategoryTestCase.class);  
        //result.addTestSuite(CarModelTestCase.class);  
        result.addTestSuite(CarTestCase.class);  
        //result.addTestSuite(AddressTestCase.class);  
        result.addTestSuite(StoreTestCase.class);  
        return result;  
    }  
}
```

Đoạn mã trên đây có một số phần bị che đi là do các lớp kiểm thử tương ứng chưa được xây dựng. Phương thức suite() trả về kết quả là kiểu Test, tuy nhiên biến result lại được khai báo kiểu TestSuite, điều này là do kiểu TestSuite được kế thừa từ kiểu Test. Các dòng mã nguồn dạng `result.addTestSuite([...].class)` cho phép người chạy thử nghiệm tìm tất cả các phương thức kiểm thử có tên bắt đầu bằng chữ test. Biên dịch và chạy lớp *CootBusinessTestSuite* cho thấy kết quả trả về đúng cho các lớp đã được kiểm thử.

#### 7.4.5 Kiểm thử chéo các phương thức

Phản tiếp theo sẽ kiểm thử phương thức *addCar*, bằng cách xây dựng phương thức *testAddCar* trong lớp *StoreTestCase*:

```
public void testAddCar() {  
    store.addCar(car);  
    assertTrue(store.containsCar(car));  
}
```

Phương thức *assertTrue* được xác nhận là thành công nếu tham số của nó là đúng. Cần phải thêm một phương thức dạng public *containsCar* vào trong lớp Store mặc dù trong sơ đồ lớp ban đầu không có. Điều này cho thấy rằng phải có một số linh hoạt giữa giai đoạn thiết kế và mã hóa, thường được coi là một điều tốt. Phương thức *containsCar* được viết như sau:

```
public boolean containsCar(Car c) {  
    return cars.contains(c);  
}
```

Biên dịch, chạy thử nghiệm với JUnit cho kết quả đúng. Có thể chạy thử nghiệm tích hợp trên lớp *CootBusinessTestSuite*.

#### 7.4.6 Completing theStoreClass

Phương thức này, `containsAlternativeCar` sẽ trả về true nếu và chỉ nếu có một đối tượng Car khác trong lớp Store mà nó phục vụ có cùng một mục đích như một tham số: tham số cho thuê xe, điều này có nghĩa là bất kỳ xe nào cũng phải cùng một mô hình và không phụ thuộc vào số km đi được. Đầu tiên, cần thêm một kiểm thử vào `StoreTestCase` để xác nhận rằng phương thức mới Store hoạt động chính xác:

```
public void testContainsAlternativeCar() {
    int traveled2 = 4435;
    store.addCar(car);
    Car car2 = new Car(carModel, traveled2);
    assertTrue(store.containsAlternativeCar(car2));
}
```

Cuối cùng, đây là phương thức cần được thêm vào lớp Store:

```
public boolean containsAlternativeCar(Car outer) {
    for (Car inner : cars) {
        if (inner.getModel() == outer.getModel()) {
            return true;
        }
    }
    return false;
}
```

Phương thức này sử dụng một dạng vòng lặp đặc biệt của lớp Collection, duyệt để kiểm tra mọi đối tượng Car trong danh sách `cars`. Trong vòng lặp, các mô hình của các Car bên trong được so sánh với mô hình của xe bên ngoài. Kết quả chạy thử nghiệm cho thấy mã là đúng. Các mã nguồn chính thức của lớp Store và lớp `StoreTestCase` như sau:

```
public class Store {  
    private Address address;  
    private Set<Car> cars;  
    Store(Address a) {  
        address = a;  
        cars = new HashSet<Car>();  
    }  
    public Address getAddress() {  
        return address;  
    }  
    public void addCar(Car c) {  
        cars.add(c);  
    }  
  
    public boolean containsCar(Car c) {  
        return cars.contains(c);  
    }  
  
    public boolean containsAlternativeCar(Car outer) {  
        for (Car inner : cars) {  
            if (inner.getModel() == outer.getModel()) {  
                return true;  
            }  
        }  
        return false;  
    }  
}
```

```
public class StoreTestCase extends TestCase {  
    private Address address;  
    private Store store;  
    private CarModel carModel;  
    private Car car;  
  
    protected void setUp() {  
        address = new Address("9", "Ash Lane", "Greater Manchester", "SK4 3HJ");  
        store = new Store(address);  
        Category category = new Category("Vintage");  
        Make make = new Make("Mostin");  
        List<Make> makes = new LinkedList<Make>();  
        makes.add(make);  
        String modelNumber = "Wheely 1950";  
        int price = 89;  
        carModel = new CarModel(category, make, modelNumber, price);  
        int traveled = 435345;  
        car = new Car(carModel, traveled);  
    }  
    public void testCreate() {  
        assertEquals(store.getAddress(), address);  
    }  
  
    public void testAddCar() {  
        store.addCar(car);  
        assertTrue(store.containsCar(car));  
    }  
  
    public void testContainsAlternativeCar() {  
        int traveled2 = 4435;  
        store.addCar(car);  
        Car car2 = new Car(carModel, traveled2);  
        assertTrue(store.containsAlternativeCar(car2));  
    }  
}
```

## 7.5 Kết chương

### Tài liệu tham khảo

## CHƯƠNG 8. MỘT SỐ KỸ THUẬT TRONG TRIỂN KHAI VÀ BẢO TRÌ HỆ THỐNG

### 8.1 Giới thiệu

Quá trình xây dựng phần mềm bao gồm các giai đoạn khác nhau từ tiếp nhận yêu cầu, phân tích, thiết kế đến mã hóa và kiểm định. Thông qua sự chặt chẽ và liên kết của các quá trình, phần mềm dần đi vào hoàn thiện và được chuyển giao cho phía người sử dụng. Việc chuyển giao bao gồm các quá trình cài đặt, cung cấp tài liệu hệ thống và bảo trì hệ thống trong một thời gian nhất định. Trong chương này, chúng ta sẽ đề cập đến các nội dung: cấu hình, triển khai và bảo trì phần mềm.

### 8.2 Cấu hình phần mềm

Mục tiêu của kỹ sư phần mềm là xây dựng phần mềm với tính linh động cao, có thể phù hợp dễ dàng với các thay đổi trong suốt quá trình hoạt động của phần mềm; bên cạnh đó, việc đảm bảo thời gian hoàn thành các thay đổi trong một khoảng thời gian nhỏ nhất cũng rất quan trọng.

Các hoạt động cấu hình phần mềm được xây dựng để xác định, kiểm soát các thay đổi, đảm bảo việc thực thi đúng đắn cho các thay đổi và báo cáo sự thay đổi tới những người có liên quan tới hệ thống. Ở đây, chúng ta cần phải phân biệt rõ ràng giữa **hỗ trợ phần mềm (software support)** và **quản lý cấu hình phần mềm (software configuration management - SCM)**. Hoạt động **hỗ trợ phần mềm** bao gồm các hoạt động hỗ trợ của kỹ sư phần mềm sau khi phần mềm được bàn giao cho khách hàng. **Cấu hình phần mềm** theo dõi, kiểm soát sự hoạt động từ khi một dự án phần mềm được bắt đầu và kết thúc khi phần mềm không còn hoạt động.

Như vậy, **cấu hình phần mềm** là một chuỗi các hoạt động được phát triển để quản lý sự thay đổi trong suốt vòng đời của phần mềm máy tính. Nó có thể được xem như một “**hoạt động bảo hiểm**” chất lượng của phần mềm trong suốt quá trình phần mềm hoạt động.

#### 8.2.1 Bối cảnh và các thành phần liên quan

Trong bối cảnh công ty cần phát triển một phần mềm cho một khách hàng nào đó, các hoạt động cấu hình sẽ liên quan đến người quản lý dự án, người này sẽ giao nhiệm vụ cho các nhóm; một người quản lý cấu hình, người sẽ giao các hoạt động cấu hình và các chính sách; một nhóm các kỹ sư phần mềm, những người sẽ phát triển và bảo trì sản phẩm phần mềm và người sử dụng, những người sẽ sử dụng phần mềm. Trong bối cảnh đó, giả sử một

sản phẩm nhỏ được phát triển bởi một nhóm gồm 6 kỹ sư bao gồm khoảng 15000 dòng code và vai trò của từng thành viên sẽ được xác định như sau:

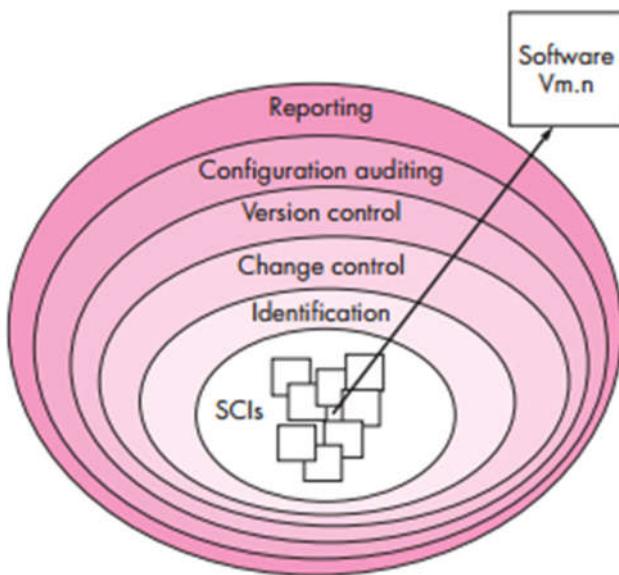
- **Người quản lý dự án:** là người giao nhiệm vụ cho các nhóm phần mềm, đảm bảo sản phẩm được hoàn thành trong một khung thời gian nhất định. Dĩ nhiên, quản lý dự án cũng bao gồm kiểm soát quá trình phát triển, kiểm tra phần mềm và các ý kiến của khách hàng.
- **Người quản lý cấu hình:** quản lý các phương pháp cấu hình và các chính sách. Mục đích chính là đảm bảo việc tạo mới, thay đổi và kiểm chứng code theo các phương pháp và chính sách, cũng như đưa ra các thông tin cho quá trình try cập. Để thực hiện việc kiểm soát code, người quản lý cấu hình phải giới thiệu các cơ chế cho việc yêu cầu thay đổi, tính toán và chứng thực sự thay đổi đó. Cụ thể là, một danh sách các tác vụ được tạo và phổ biến đến các kỹ sư.
- **Các kỹ sư phần mềm:** phát triển và bảo trì sản phẩm phần mềm. Các kỹ sư không nhất thiết phải giao tiếp với nhau để xây dựng hay kiểm tra code. Tuy nhiên, việc giao tiếp và phối hợp trong công việc được khuyến khích; đặc biệt là việc sử dụng các công cụ để xây dựng sản phẩm một cách phù hợp.
- **Người dùng:** người sử dụng phần mềm. Theo đó, người dùng có thể dựa trên các phương thức được định sẵn để yêu cầu thay đổi các chức năng và xác định lỗi trong phần mềm.

### 8.2.2 Quá trình cấu hình phần mềm

Quá trình cấu hình phần mềm xác định một chuỗi các công việc (task) với bốn mục đích chính sau đây:

- Đảm bảo chất lượng phần mềm được bảo trì;
- Đảm bảo tất cả các thành phần cho quá trình cấu hình phần mềm;
- Quản lý sự thay đổi trên một hay nhiều thành phần;
- Tạo điều kiện để xây dựng phần mềm với nhiều phiên bản khác nhau;

Các tác vụ của quá trình cấu hình phần mềm bao gồm xác định các đối tượng, kiểm soát sự thay đổi, kiểm soát phiên bản, kiểm tra việc cấu hình và báo cáo (hình dưới đây).



#### 8.2.2.1 Xác định các đối tượng

Để xác định và quản lý các thành phần cấu hình phần mềm, các đối tượng được đặt tên riêng và sau đó được tổ chức thành nhóm sử dụng phương pháp hướng đối tượng. Có hai loại đối tượng có thể được xác định là: các đối tượng cơ bản (basic objects) và các đối tượng tổng hợp (aggregate objects). Mỗi đối tượng có một tập hợp các tính năng riêng để xác định nó là duy nhất như tên, mô tả, danh sách tài nguyên,...

- Đối tượng cơ bản: là một đơn vị của thông tin được tạo trong quá trình phân tích, thiết kế, mã hóa hay kiểm thử.
- Đối tượng tổng hợp: là một tập các đối tượng cơ bản và các đối tượng tổng hợp khác.

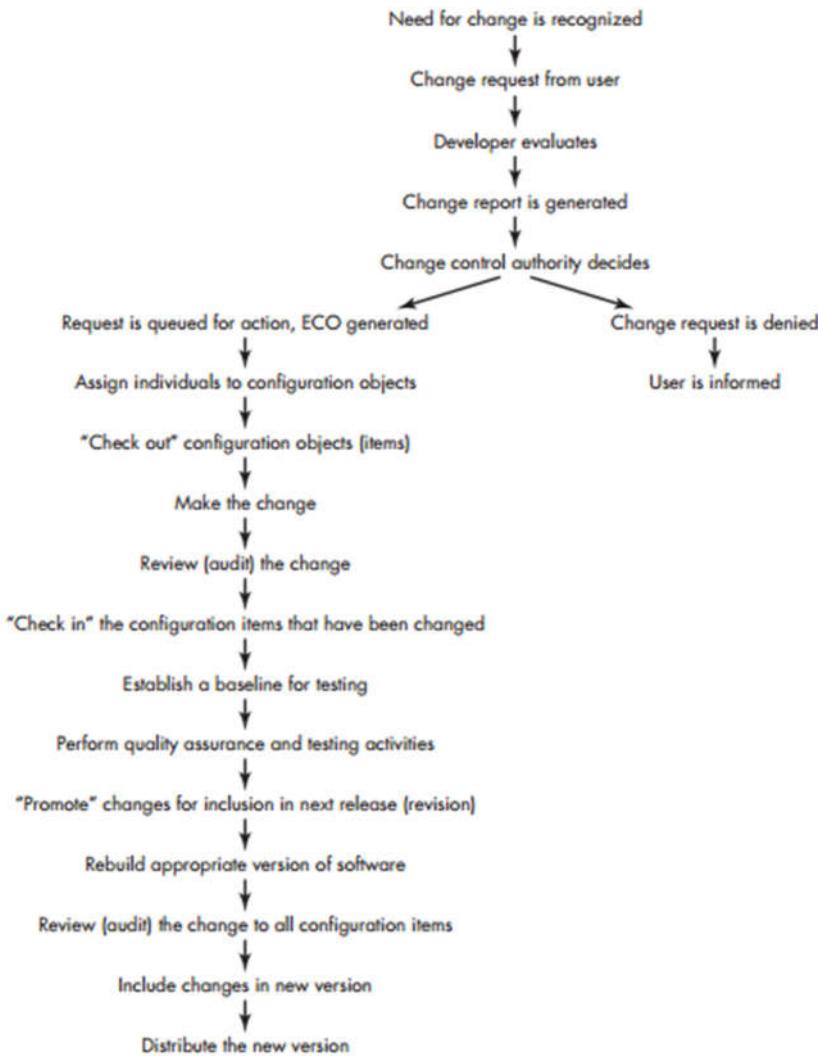
#### 8.2.2.2 Kiểm soát phiên bản

Kiểm soát phiên bản hợp nhất các thủ tục và công cụ để quản lý các phiên bản khác nhau của các đối tượng (cấu hình) được tạo ra trong quá trình xây dựng phần mềm.Thêm vào đó, việc kiểm soát phiên bản cũng thực thi khả năng ghi dấu để cho phép các nhóm theo dõi các trạng thái bên ngoài liên quan đến đối tượng trong quá trình cấu hình phần mềm.

Một số các hệ thống kiểm soát phiên bản thiết lập một tập các thay đổi. Đó là một bộ sưu tập về tất cả sự thay đổi (với bất kỳ lý do gì, ai đã thay đổi và thay đổi khi nào?) được yêu cầu để tạo ra một phiên bản mới. Điều này cho phép xây dựng một phiên bản khác của phần mềm bằng cách chỉ ra những thay đổi được áp dụng trên sự cấu hình ban đầu.

### 8.2.2.3 Kiểm soát sự thay đổi

Đối với một dự án lớn, việc không kiểm soát sự thay đổi có thể dẫn đến sự hỗn độn. Trong nhiều dự án, việc kiểm soát sự thay đổi được thực hiện dựa trên sự kết hợp giữa sức người lẫn các công cụ tự động. Quá trình kiểm soát thay đổi được minh họa như hình dưới đây:



Khi một yêu cầu thay đổi được đưa ra, các đánh giá về kỹ thuật, sự ảnh hưởng giữa các bên liên quan, các tác động tổng thể trên các đối tượng, các chức năng của hệ thống và chi phí của việc thay đổi được đánh giá. Kết quả cuối cùng của việc đánh giá được trình bày dưới dạng một báo cáo, và nó được quyết định bởi “một chứng thực cho việc thay đổi” (có thể là một người hay một nhóm), chịu trách nhiệm đưa ra quyết định (có thay đổi hay không) hay độ ưu tiên của sự thay đổi (nếu có nhiều sự thay đổi). Trong trường hợp sự thay đổi được thực hiện, một ECO (Engineering Change Order) được đưa ra cho mỗi một sự

thay đổi được thông qua. ECO mô tả việc thay đổi, các ràng buộc liên quan và các điều khoản cho việc xem xét (review) và kiểm soát. Các đối tượng được thay đổi có thể được đặt trong 1 thư mục được kiểm soát bởi các kỹ sư phần mềm. Kèm theo đó, hệ thống kiểm soát phiên bản được cập nhật cho mỗi lần thay đổi, các đối tượng thay đổi có thể được check-out khỏi dự án và được áp dụng các cơ chế kiểm soát thích hợp để phát hành cho một phiên bản phần mềm mới.

#### **8.2.2.4 Kiểm soát việc cấu hình**

Việc xác định các đối tượng, kiểm soát thay đổi và kiểm soát phiên bản giúp chúng ta bảo trì hệ thống theo thứ tự đã được định nghĩa trong quá trình phát triển phần mềm. Tuy nhiên, làm thế nào để đảm bảo các thay đổi của hệ thống được thực hiện? Câu trả lời là: xem xét kỹ thuật (technical review) và kiểm soát cấu hình phần mềm (software configuration audit).

Xem xét kỹ thuật làm rõ tính đúng đắn của việc cấu hình các đối tượng được định nghĩa; trong khi đó, kiểm soát cấu hình phần mềm bổ sung việc xem xét kỹ thuật bằng cách đánh giá các đặc điểm của việc cấu hình đối tượng được phát sinh nhưng chưa được xem xét trong quá trình xem xét kỹ thuật.

#### **8.2.2.5 Báo cáo trạng thái**

Báo cáo trạng thái là một trong những công việc cấu hình phần mềm để trả lời cho các câu hỏi:

- Phần mềm đã thay đổi những gì ?
- Ai đã thực hiện việc thay đổi đó ?
- Việc thay đổi xảy ra khi nào ?
- Những ảnh hưởng của việc thay đổi là gì ?

Tại mỗi thời điểm, một quá trình cấu hình phần mềm được gán mới hay định nghĩa, một báo cáo trạng thái cấu hình được tạo ra để chấp nhận các sự thay đổi cấu hình, đầu ra của nó được đặt trên một cơ sở dữ liệu online hay một website. Vì vậy, người phát triển phần mềm hay nhóm hỗ trợ có thể truy cập sự để thay đổi thông tin bằng các từ khóa tìm kiếm.

### **8.2.3 Cấu hình cho các ứng dụng Web**

Trong phần trước, chúng ta đã đề cập đến các cấu hình cho các ứng dụng ở dạng tổng quát. Trong phần này, chúng ta sẽ đề cập rõ hơn trong việc áp dụng việc cấu hình đó cho các ứng dụng Web.

### 8.2.3.1 Các vấn đề

Khi các ứng dụng Web trở nên quan trọng trong kinh doanh và phát triển nhanh chóng, chúng cần được cấu hình để quản lý sự phát triển đó. Việc cấu hình website ngoài việc kiểm soát hiệu suất của website thì có thể ảnh hưởng tới các vấn đề trong kinh doanh của một công ty như *việc không chứng thực cho việc đăng thông tin của một sản phẩm mới, sự sai sót hay sự “nghèo nàn” của các chức năng ảnh hưởng đến người truy cập đến một trang web, hay các lỗ hổng an ninh mà có thể gây nguy hiểm cho các hệ thống nội bộ của công ty, và hậu quả về kinh tế hay các tác hại khác.*

Các kỹ thuật và giai đoạn của SCM được mô tả ở phần trước đều có thể áp dụng để phù hợp với các ứng dụng Web. Tuy nhiên, các quá trình cần phải được điều chỉnh để phù hợp hơn với tính chất đặc đáo của ứng dụng Web.

- **Content:** một ứng dụng Web bất kỳ có thể chứa một mảng các nội dung text, đồ họa, applets, script, tập tin media,.... Một thách thức lớn là việc tổ chức các nội dung trên thành các nhóm đối tượng để cấu hình và sau đó thiết lập các kiểm soát phù hợp cho từng loại đối tượng. Với mục đích là mô hình các nội dung của ứng dụng Web sử dụng các kỹ thuật mô hình hóa, kèm với tập các thuộc tính của các đối tượng. Các tính chất động/ tĩnh và tuổi thọ dự kiến của nó là các ví dụ về các thuộc tính được yêu cầu để thiết lập một SCM hiệu quả.
- **People:** vì một phần lớn đáng kể của việc phát triển ứng dụng web tiếp tục được tiến hành một cách đặc biệt, bất kỳ người nào liên quan đến ứng dụng Web đều có thể tạo nội dung. Nhiều người tạo nội dung không có kiến thức cơ bản về kỹ thuật và được hoàn thành không biết về việc quản lý cấu hình.
- **Scalability:** các kỹ thuật và sự kiểm soát không có khả năng mở rộng không được áp dụng cho một ứng dụng Web. Đó không phải là dành cho một ứng dụng Web đơn giản thông thường để tăng sự kết nối với cá hệ thống thông tin đã tồn tại, các cơ sở dữ liệu, trung tâm dữ liệu và các công thông tin được thực thi. Khi kích thước và độ phức tạp tăng lên, các thay đổi nhỏ có thể sâu-rộng và các ảnh hưởng không thể lường trước được. Vì vậy, sự chặt chẽ của các cơ chế kiểm soát nên tỉ lệ thuận với quy mô của ứng dụng.
- **Politics:** Ai sở hữu một ứng dụng Web? Câu hỏi này thật sự quan trọng để quản lý và kiểm soát các hoạt động của website. Trong một vài trường hợp, người phát triển Web không thuộc bất kỳ một tổ chức IT nào tạo nên sự khó khăn trong việc giao tiếp và quản lý các hoạt động.

Các kỹ thuật quản lý cấu hình cho ứng dụng Web vẫn đang tiếp tục phát triển. Một quá trình SCM có thể là quá cồng kềnh để quản lý cho ứng dụng Web, nhưng một phiên bản mới của các công cụ SCM được thiết kế đặc biệt cho phát triển Web đã nổi lên trong một vài năm trở lại đây. Các công cụ này thiết lập một quá trình mà phù hợp với các thông tin đã có, quản lý sự thay đổi của các đối tượng, cấu trúc chúng theo cách cho phép biểu diễn nó ở phía người dùng và sau đó cung cấp nó cho môi trường client để hiển thị.

#### **8.2.3.2 Các đối tượng cấu hình của ứng dụng Web**

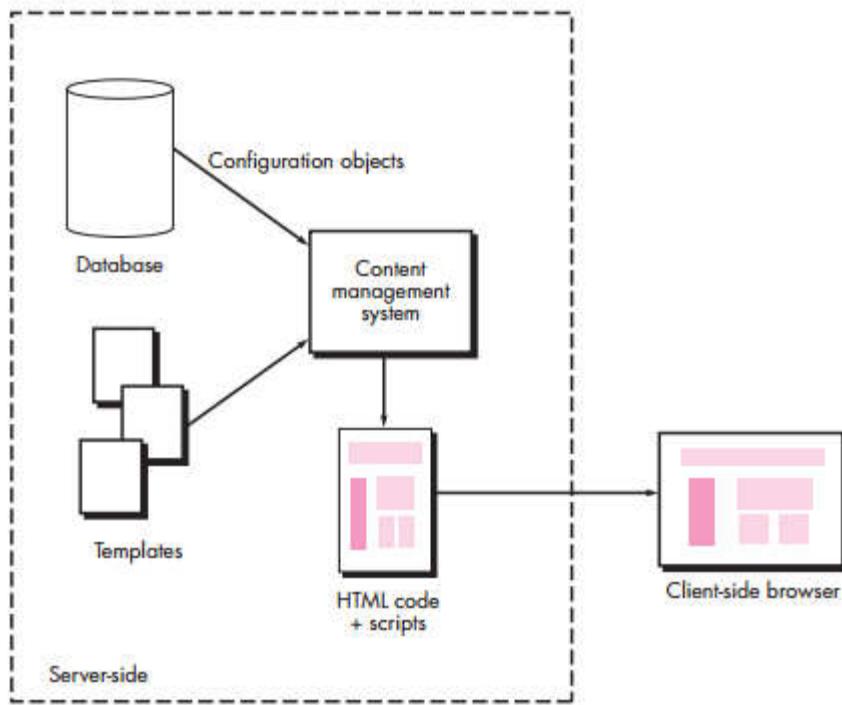
Ứng dụng Web bao gồm một vùng rộng các đối tượng cấu hình như nội dung, đồ họa, hình ảnh, âm thanh, các nội dung hay các đối tượng giao diện. Các đối tượng đó có thể được xác định theo nhiều cách thức khác nhau cho việc tổ chức. Tuy nhiên, các quy ước dưới đây phải lưu ý để đảm bảo rằng các nền tảng tương thích được duy trì: chiều dài tên tập tin nên giới hạn trong vòng 32 ký tự; nên tránh việc sử dụng các hỗn hợp các ký tự hay tất cả đều hoa; .... Thêm vào đó, việc sử dụng liên kết tới các đối tượng nên dùng đường dẫn tương đối.

Tất cả các ứng dụng Web có định dạng và cấu trúc. Các tập tin được định dạng là bắt buộc bởi việc tính toán môi trường trong các nội dung được lưu trữ. Tuy nhiên, định dạng hiển thị là được định nghĩa bởi kiểu định dạng và các luật thiết kế được thiết lập cho ứng dụng Web. Cấu trúc nội dung định nghĩa một kiến trúc nội dung, nó định nghĩa cách hiển thị nội dung của các đối tượng tới phía người dùng.

#### **8.2.3.3 Quản lý nội dung**

Quản lý nội dung liên quan tới việc quản lý cấu hình trong các ngữ cảnh thiết lập sự hiển thị nội dung của các đối tượng để hiển thị tới một người dùng, và sau đó là cung cấp nó cho môi trường client để hiển thị.

Phương pháp phổ biến nhất là xây dựng một ứng dụng Web với một hệ thống quản lý nội dung động. Điều đó là, người dùng có thể yêu cầu bất kỳ thông tin gì trên ứng dụng Web. Ứng dụng Web truy vấn cơ sở dữ liệu, định dạng thông tin theo sự truy vấn và hiển thị nó cho người dùng.



Trong hầu hết các trường hợp cơ bản, một hệ thống quản lý nội dung được tích hợp từ 3 hệ thống con (subsystem): ***a collection subsystem, a management subsystem, and a publishing subsystem.***

- ***Collection subsystem:*** bao hàm tất cả các hành động được yêu cầu để tạo, truy xuất nội dung, và các chức năng kỹ thuật là cần thiết để chuyển nội dung về dạng có thể hiển thị bởi ngôn ngữ đánh dấu (mark-up language (HTML)) và tổ chức nội dung thành các gói để có thể hiển thị bên phía người dùng.  
Việc tạo và chứng thực nội dung thông thường thường thực hiện song song với các hoạt động phát triển Web khác và thường được tiến hành bởi các nhà phát triển nội dung không liên quan đến kỹ thuật. Hoạt động này kết hợp các thành phần của việc tạo và nghiên cứu và được hỗ trợ bởi các công cụ mà cho phép tác giả nội dung mô tả nội dung theo các chuẩn được sử dụng trong các ứng dụng Web.
- ***Management subsystem:*** thực thi một repository bao gồm các thành phần sau:
  - ***Content database:*** thông tin cấu trúc mà được thiết lập để lưu trữ tất cả các nội dung
  - ***Data capabilities:*** các chức năng cho phép CMS tìm kiếm các nội dung cụ thể, lưu trữ và lấy lại các đối tượng, và quản lý các tập tin cấu trúc đã được thiết lập cho nội dung.

- *Configuration management functions*: các thành phần chức năng và luồng công việc liên quan tới việc hỗ trợ việc xác định các đối tượng, kiểm soát phiên bản, quản lý thay đổi, kiểm soát và báo cáo.
- **Publishing subsystem**: nội dung phải được trích xuất từ repository, chuyển thành một định dạng có thể dễ hiểu để xuất bản và được định dạng để có thể truyền tới trình duyệt ở client.

#### 8.2.3.4 Quản lý sự thay đổi

Để thực hiện hiệu quả việc quản lý thay đổi, người ta sử dụng triết lý “code and go” để tiếp tục quản trị việc phát triển ứng dụng Web, các quá trình kiểm soát thông thường phải được thay đổi để phù hợp hơn với các ứng dụng Web. Mỗi sự thay đổi nên được phân loại thành 1 trong 4 nhóm sau:

- Nhóm 1 (Class 1): thay đổi chức năng hay nội dung để sửa lỗi hay tăng cường nội dung hay chức năng cục bộ.
- Nhóm 2 (Class 2): thay đổi chức năng hay nội dung mà ảnh hưởng đến nội dung hay chức năng của các thành phần khác.
- Nhóm 3 (Class 3): thay đổi chức năng hay nội dung có ảnh hưởng rộng rãi đến ứng dụng Web.
- Nhóm 4 (Class 4): thay đổi thiết kế mà ảnh hưởng trực tiếp đến việc cảnh báo một hay nhiều danh mục của người dùng.

#### 8.2.3.5 Quản lý phiên bản

Vì các ứng dụng Web ngày càng tiến hóa theo thời gian nên một số lượng các phiên bản khác nhau có thể tồn tại tại cùng một thời điểm. Một phiên bản tồn tại thông qua internet tới người dùng; một phiên bản khác có thể trong giai đoạn kiểm chứng trước khi triển khai; phiên bản thứ ba đang được phát triển và cập nhật nội dung. Các đối tượng cấu hình phải được định nghĩa một cách rõ ràng để có được sự liên quan với các phiên bản phù hợp. Để kiểm soát điều đó, một quá trình kiểm soát phiên bản được yêu cầu.

- Một trung tâm lưu trữ (repository) cho dự án Web nên được thiết lập
- Mỗi lập trình viên tạo một thư mục để lưu trữ các công việc của họ.
- Các đồng hồ trên tất cả các máy trạm làm việc của lập trình viên nên được đồng bộ.
- Khi các đối tượng cấu hình mới được phát triển, hay các đối tượng đã tồn tại được thay đổi, chúng nó nên được đưa vào trung tâm lưu trữ

- Khi các đối tượng được đưa vào hay lấy ra từ các trung tâm một cách tự động, thì nên lưu vết các thời gian truy xuất

#### 8.2.3.6 Rà soát đánh giá và báo cáo

Tất cả các đối tượng được đánh dấu là vào hay ra khỏi trung tâm (repository) sẽ được ghi dấu trong log và có thể xem tại bất kỳ thời điểm nào. Một báo cáo log hoàn chỉnh có thể được tạo để tất cả các thành viên của nhóm phát triển Web có cách để tìm ra sự thay đổi.Thêm vào đó, một email cảnh báo tự động có thể được gửi tại mỗi thời điểm một đối tượng được đưa vào hay lấy ra khỏi repository.

### 8.3 Triển khai phần mềm trên các hệ thống

### 8.4 Một số nguyên lý trong bảo trì phần mềm

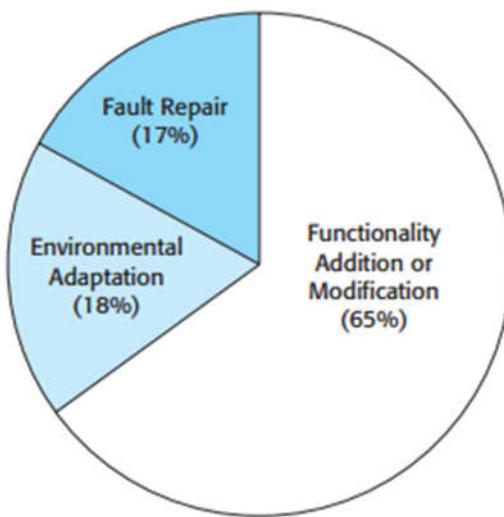
Bảo trì phần mềm là một quá trình thay đổi hệ thống sau khi nó được bàn giao. Thuật ngữ thường được áp dụng để tùy chỉnh phần mềm khi nó được chia ra để các nhóm phát triển trước và sau khi bàn giao. Sự thay đổi này làm cho phần mềm có thể dễ dàng thay đổi để sửa các lỗi, nhiều cách để sửa các lỗi thiết kế hay nâng cao tính đúng đắn của các mô tả kỹ thuật hay phù hợp với các yêu cầu mới. Sự thay đổi được thực thi bằng cách sửa đổi các thành phần đã có của hệ thống và thêm các thành phần mới vào hệ thống, nếu cần thiết.

Có 3 kiểu bảo trì phần mềm khác nhau:

- **Fault repairs:** các lỗi mã hóa thường dễ dàng để sửa chữa; các lỗi thiết kế là rất khó để sửa chữa (và cũng thường đắt đỏ hơn) khi có thể phải viết lại một số thành phần của chương trình. Các lỗi yêu cầu (yêu cầu thay đổi) là đắt nhất để sửa chữa bởi vì liên quan tới việc thiết kế lại những thành phần cần thiết.
- **Environmental adaptation:** Kiểu bảo trì này được yêu cầu khi một số phương tiện môi trường của hệ thống như phần cứng, nền tảng phần mềm hệ thống, hay các phần mềm hỗ trợ khác thay đổi. Khi đó, ứng dụng hệ thống phải được định nghĩa để phù hợp với sự thay đổi của môi trường.
- **Functionality addition:** Kiểu bảo trì này là cần thiết khi các yêu cầu hệ thống thay đổi trong sự phản hồi của các tổ chức hay hình thức kinh doanh thay đổi. Sự ảnh hưởng tới phần mềm của việc bảo trì này thường lớn hơn so với các kiểu bảo trì khác.

Trên thực tế, không có sự phân biệt rõ ràng giữa 3 loại bảo trì này. Khi bạn đặt phần mềm vào một môi trường mới, bạn có thể thêm những chức năng để phù hợp với các tính năng của môi trường mới. Các lỗi phần mềm là thường gặp bởi vì người dùng sử dụng phần mềm bằng nhiều cách không thể dự đoán trước được. Việc thay đổi hệ thống để phù hợp với cách làm việc của họ là cách tốt nhất để sửa các lỗi.

Đã có vài nghiên cứu của bảo trì phần mềm về mối quan hệ giữa bảo trì và phát triển và sự khác biệt giữa các hoạt động bảo trì. Vì sự khác nhau của thuật ngữ, các chi tiết của các nghiên cứu không thể so sánh. Các cuộc điều tra rộng rãi đồng ý rằng chi phí bảo trì phần mềm cao hơn rất nhiều so với phát triển mới. Họ cũng đồng ý rằng nhiều chi phí bảo hành thường được dùng cho việc thực thi yêu cầu mới hơn là sửa lỗi. Biểu đồ dưới đây hiển thị phần xấp xỉ của các chi phí bảo trì. Các chi phí cụ thể có thể khác nhau tùy theo các tổ chức nhưng chi phí sửa lỗi hệ thống là không phải là chi phí bảo trì đắt nhất.



Các chi phí liên quan của bảo trì và phát triển mới khác nhau từ lĩnh vực ứng dụng tới một lĩnh vực khác. Guimaraes đã tìm ra rằng các chi phí bảo trì cho các hệ thống ứng dụng kinh doanh có thể so sánh rộng rãi với chi phí phát triển hệ thống.

Với các hệ thống nhúng thời gian thực, chi phí bảo trì tăng lên 4 lần. Độ tin cậy và hiệu suất của các yêu cầu của các hệ thống có sự liên kết chặt chẽ và dễ nhiên khó khăn thay đổi. Mặc dù chúng được ước lượng hơn 25 năm, nó không chắc chắn rằng sự phân phối chi phí cho các loại hệ thống khác nhau đã thay đổi đáng kể.

Nó thường là chi phí hiệu quả để đầu tư cho quá trình thiết kế và thực thi một hệ thống để giảm các chi phí cho những thay đổi trong tương lai. Việc thêm các chức năng sau khi giao sản phẩm là rất đắt đỏ bởi vì cần tốn thời gian để xem lại hệ thống và phân tích xung

đột của những thay đổi được yêu cầu. Vì vậy, công việc được hoàn thành trong quá trình phát triển sẽ làm cho phần mềm dễ hiểu hơn và giảm chi phí khi thay đổi.

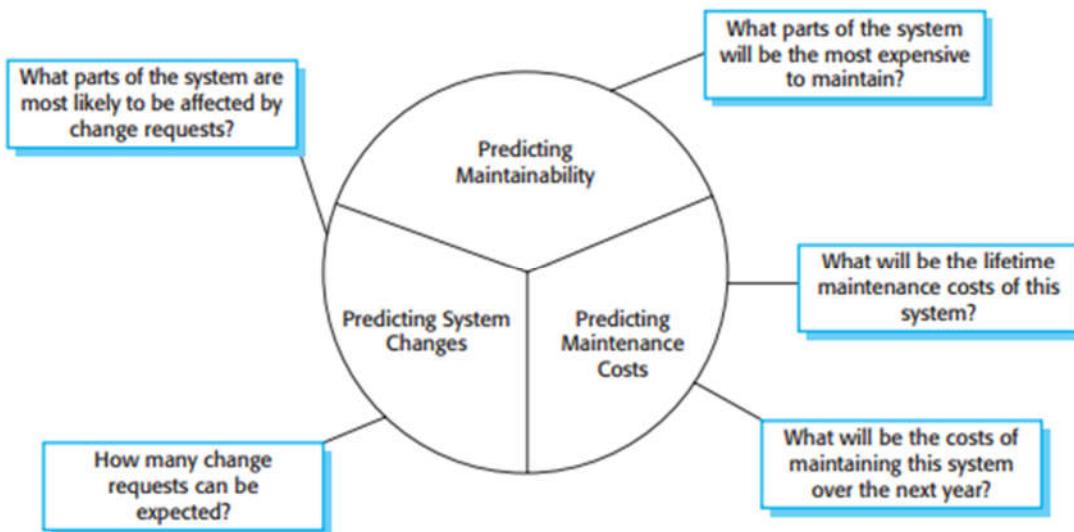
Những ước tính này chỉ mang tính lý thuyết nhưng không ngại rằng việc phát triển phần mềm làm cho nó dễ bảo trì hơn là có hệ quả, khi các chi phí trong suốt hoạt động của phần mềm được tính. Đây là lý do cho việc “tái xây dựng” được phát triển nhanh chóng. Nếu không có tái cấu trúc (**refactoring**), việc thay đổi code sẽ rất khó và chi phí sẽ cao. Tuy nhiên, trong việc xây dựng kế hoạch phát triển, hiếm khi người ta đầu tư bổ sung vào việc cải thiện code trong quá trình phát triển. Điều này chủ yếu là do cách các tổ chức “chạy” nguồn vốn của doanh nghiệp. Việc đầu tư trong bảo trì làm tăng chi phí “ngắn hạn”, đó là những gì đã được đo lường từ trước. Thật không may, những lợi ích lâu dài không thể cùng lúc được đo lường, vì vậy các công ty không muốn chi tiền cho một tương lai không rõ ràng. Nó thường đắt hơn để thêm chức năng sau khi một hệ thống đã hoạt động hơn là thực thi các chức năng tương đương trong quá trình phát triển. Các lý do là:

- **Team stability:** Sau khi một dự án hoàn thành, thông thường các nhóm phát triển sẽ được tách rời để tham gia vào các dự án khác. Một nhóm mới hay một cá nhân chịu trách nhiệm cho việc bảo trì hệ thống không thể hiểu hệ thống hay nền tảng để quyết định thiết kế hệ thống. họ cần thời gian để hiểu hệ thống đã có trước khi thực thi những thay đổi.
- **Poor development practice:** hợp đồng bảo trì hệ thống thường tách riêng biệt với hợp đồng phát triển hệ thống. Hợp đồng bảo trì có thể được đưa cho công ty khác (với công ty phát triển phần mềm). Chính yếu tố này, cùng với sự thiếu ổn định của đội ngũ đã dẫn đến kích thích cho một nhóm phát triển để viết các bảo trì cho phần mềm. Nếu một nhóm phát triển có thể cắt giảm để tiết kiệm công sức trong quá trình phát triển cho phù hợp với công sức của họ thì phần mềm sẽ khó để thay đổi trong tương lai.
- **Staff skills:** đội ngũ bảo trì thường không có kinh nghiệm và phù hợp với lĩnh vực của ứng dụng. Sự bảo trì có một hình ảnh không đẹp giữa các kỹ sư phần mềm. Nó được xem là quá trình ít kỹ năng hơn phát triển hệ thống và nó thường được phân bổ cho các nhân viên ở cấp thấp nhất. Hơn thế nữa, các hệ thống cũ có thể được viết bằng các ngôn ngữ lập trình lỗi thời. Đội ngũ bảo trì có thể không có nhiều kinh nghiệm trong việc phát triển trong những ngôn ngữ đó và phải học những ngôn ngữ này để bảo trì hệ thống.
- **Program age and structure:** khi các thay đổi được đưa vào chương trình, cấu trúc của chương trình sẽ “suy giảm”. Do đó, chương trình sẽ trở nên khó hiểu và thay đổi hơn.

Ba nguyên nhân đầu tiên có nguồn gốc từ sự thật mà nhiều tổ chức vẫn xem xét việc phát triển và bảo trì được phân chia rõ ràng. Việc bảo trì được xem như nhóm hoạt động thứ hai và nó không có động cơ để sử dụng tiền trong quá trình phát triển để giảm chi phí thay đổi của hệ thống. Các giải pháp mang tính lâu dài cho vấn đề này là chấp nhận các hệ thống hết hạn sử dụng nhưng vẫn được tiếp tục dùng, trong một vài trường hợp là vô thời hạn.

#### 8.4.1 Dự đoán sự bảo trì

Các nhà quản lý ghét sự kinh ngạc, đặc biệt nếu kết quả đó là các chi phí không mong muốn (ở mức cao). Vì vậy, nên cố gắng dự đoán những thay đổi của hệ thống có thể được đề xuất và những phần của hệ thống khó có thể bảo trì nhất. Bạn cũng nên cố gắng để dự đoán chi phí bảo trì toàn bộ cho một hệ thống thông qua một khoảng thời gian. Biểu đồ dưới đây mô tả những dự đoán và các câu hỏi liên quan:



Việc dự đoán số lượng các yêu cầu thay đổi cho một hệ thống yêu cầu một sự hiểu biết về mối quan hệ giữa hệ thống và những tồn tại của nó với môi trường bên ngoài. Một vài hệ thống có mối quan hệ vô cùng phức tạp với môi trường bên ngoài và việc môi trường thay đổi không thể tránh khỏi sự thay đổi của hệ thống. Để đánh giá mối quan hệ giữa hệ thống và môi trường của nó, bạn nên đánh giá:

- Số lượng và độ phức tạp của các giao diện hệ thống.
- Số lượng các yêu cầu dễ thay đổi của hệ thống.
- Quy trình nghiệp vụ trong khi hệ thống được sử dụng.

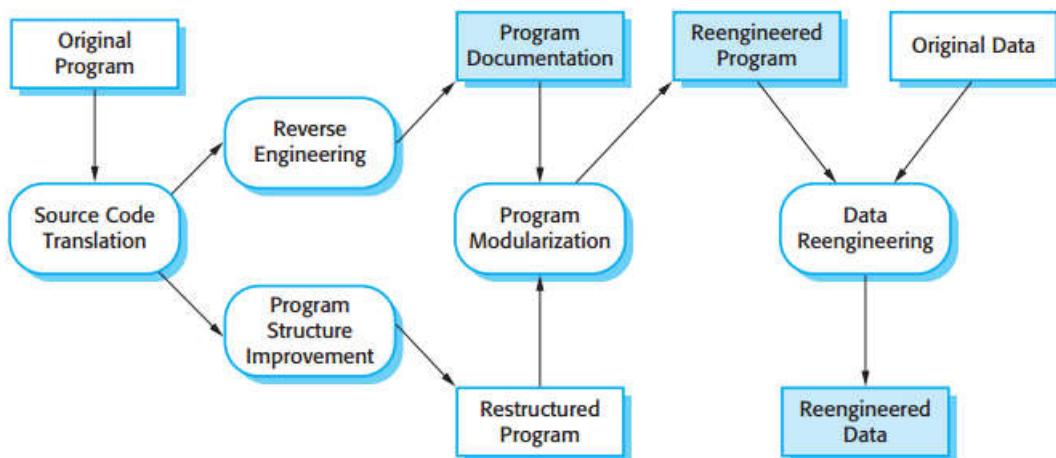
### 8.4.2 Tái cấu trúc phần mềm

Như đã đề cập ở phần trước, quá trình nâng cấp phần mềm liên quan đến việc hiểu chương trình để thay đổi chúng và sau đó thực thi các sự thay đổi này. Tuy nhiên trong nhiều hệ thống, đặc biệt là các hệ thống cũ thì rất khó để hiểu và thay đổi chúng. Các chương trình có thể đã được tối ưu hóa hiệu suất hay tiết kiệm chi phí là điều dễ hiểu, hay, theo thời gian, cấu trúc chương trình ban đầu có thể hỏng bởi một loạt các thay đổi.

Để làm cho các hệ thống dễ dàng để bảo trì, có thể tái cấu trúc những hệ thống để cải tiến cấu trúc của nó và hiểu thêm về hệ thống đó. Tái cấu trúc có thể bao gồm việc viết lại tài liệu cho hệ thống, cấu trúc lại kiến trúc của hệ thống, chuyển ngôn ngữ lập trình sang ngôn ngữ lập trình mới, định nghĩa và cập nhật cấu trúc và giá trị dữ liệu của hệ thống. Các chức năng của chương trình là không đổi và bình thường hóa, bạn nên tránh các thay đổi lớn đến cấu trúc của phần mềm. Hai lợi ích quan trọng từ việc tái cấu trúc hơn là thay thế phần mềm là:

- **Giảm rủi ro:** đây là một rủi ro cao trong việc phát triển lại phần mềm kinh doanh quan trọng. Các lỗi có thể gặp phải trong các chức năng của phần mềm hay các vấn đề trong quá trình phát triển. Việc chậm trễ giới thiệu phần mềm mới đồng nghĩa với việc kinh doanh bị trì trệ, mất doanh thu và chi phí mở rộng phần mềm có thể tăng thêm.
- **Giảm chi phí:** chi phí cho việc tái cấu trúc thường sẽ ít hơn chi phí cho việc phát triển phần mềm mới.

Sơ đồ là một mô hình tổng quát cho quá trình tái cấu trúc phần mềm. Đầu vào của quá trình này là một chương trình cũ và đầu ra chương trình tương đương đã được cải tiến và tái cấu trúc.



- **Source code translation:** sử dụng công cụ chuyển đổi, chương trình được chuyển từ một ngôn ngữ lập trình cũ sang một phiên bản mới của ngôn ngữ đó hay sang một ngôn ngữ lập trình khác.
- **Reverse engineering:** chương trình được phân tích và rút trích thông tin từ code. Việc này giúp ta có được các tài liệu về cách tổ chức và các chức năng của chương trình. Quá trình này thường được thực hiện tự động.
- **Program structure improvement:** việc kiểm soát cấu trúc của phần mềm được phân tích và định nghĩa để tạo thuận lợi cho việc đọc và hiểu. Việc này có thể thực hiện tự động nhưng một vài quá trình có thường được yêu cầu thực hiện thủ công.
- **Program modularization:** nhóm các chức năng thành nhóm với tiêu chí phù hợp hay không cần thiết. Đây là quá trình thủ công.
- **Data reengineering:** quá trình xử lý dữ liệu bởi chương trình được thay đổi để cho thấy sự thay đổi của chương trình. Điều này có nghĩa là định nghĩa lại cấu trúc của cơ sở dữ liệu và chuyển cơ sở dữ liệu hiện có qua một cấu trúc mới.

Thật sự, việc tái cấu trúc không nhất thiết phải thực hiện trên tất cả các bước ở sơ đồ trên. Không cần phải chuyển mã nếu mà vẫn dùng ngôn ngữ lập trình của chương trình. Hay, việc tái cấu trúc dữ liệu chỉ cần khi cấu trúc dữ liệu trong chương trình thay đổi trong quá trình tái cấu trúc hệ thống. Việc giảm thiểu tối đa việc tái cấu trúc các bước sẽ giảm thiểu tối đa thời gian cập nhật và chi phí cho chương trình.

#### 8.4.3 Bảo dưỡng chương trình bằng tái cấu trúc

Refactoring là quá trình của việc cải tiến chương trình nhằm giảm sự suy thoái của chương trình thông qua việc thay đổi. Nó có nghĩa là cải tiến cấu trúc của chương trình, giảm độ phức tạp, hay làm cho nó dễ hiểu hơn. Refactoring đôi khi được xem xét để giới hạn việc áp dụng cho sự phát triển các chương trình hướng đối tượng nhưng những nguyên tắc có thể áp dụng tới bất kỳ phương pháp phát triển nào. Khi bạn refactoring chương trình, bạn không nên thêm các chức năng mà nên tập trung vào việc cải tiến chương trình. Do đó, bạn có thể nghĩ rằng refactoring như quá trình bảo dưỡng để giảm các vấn đề thay đổi trong tương lai.

Mặc dù tái cấu trúc chương trình và refactoring đều hướng đến việc làm cho chương trình dễ hiểu và dễ thay đổi hơn, nhưng đó là hai điều khác nhau. Tái cấu trúc chương trình đưa ra sau khi một hệ thống đã được bảo trì nhiều lần và chi phí bảo trì tăng dần. Bạn sử dụng các công cụ tự động để thực hiện và tái cấu trúc hệ thống cũ để tạo một hệ thống mới với nhiều sự duy trì. Refactoring là tiếp tục quá trình của việc cải tiến thông qua quá trình

phát triển và cải tiến. Nó được thiết kế để tránh việc suy thoái cấu trúc và mã hóa để giảm chi phí bảo trì hệ thống.

Refactoring là một phần vốn có của các phương pháp nhanh nhẹn dựa trên các thay đổi. Chất lượng chương trình vì vậy suy thoái một cách nhanh chóng, vì vậy các nhà phát triển refactor chương trình của họ để tránh sự suy thoái đó.Thêm vào đó, refactoring không phụ thuộc vào các hoạt động và có thể được dùng với bất kỳ mục đích nào để phát triển chương trình. Một vài tình huống có thể được cải tiến thông qua refactoring bao gồm:

- **Duplicate code:** các đoạn mã giống nhau có thể ở nhiều vị trí khác nhau trong chương trình. Nó có thể được xóa và thực thi như một phương thức hay chức năng mà được gọi lại khi được yêu cầu.
- **Long methods:** nếu một phương thức quá dài, nó nên được thiết kế lại với các phương thức ngắn hơn.
- **Switch (case) statements:** nó thường trùng nhau, nơi mà chuyển phụ thuộc vào loại của các giá trị.
- **Data clumping:** kết hợp dữ liệu xuất hiện khi các nhóm dữ liệu giống nhau xuất hiện tại một vài nơi trong chương trình. Điều này có thể được thay thế với một đối tượng đóng gói tất cả dữ liệu.
- **Speculative generality:** điều này xuất hiện khi các nhà phát triển sinh ra các mã chương trình mà chúng có thể được yêu cầu trong tương lai.

Refactoring, được đưa ra trong quá trình phát triển chương trình, nó như là một cách để giảm chi phí bảo trì của chương trình trong một thời gian dài. Tuy nhiên, nếu bạn làm việc trên một chương trình đã bảo trì và cấu trúc của nó đã xuống cấp đáng kể thì việc refactor nó là một không thể. Bạn có thể nghĩ đến việc thiết kế refactoring, điều này là vô cùng đắt đỏ và là một vấn đề khó khăn. Nó liên quan đến việc định nghĩa các mẫu thiết kế và thay thế mã chương trình đã có bằng mã chương trình thực thi cho các mẫu thiết kế đó.

## 8.5 Kết chương

### Tài liệu tham khảo

1. Roger S.Pressman, “*Software Engineering: A Practitioner’s Approach*”, McGraw-Hill, Chapter 22, page 584 – 613
2. Ian Sommerville, “*Software Engineering (Ninth Edition)*”, Addison-Wesley, Chapter 9, page 242 - 252