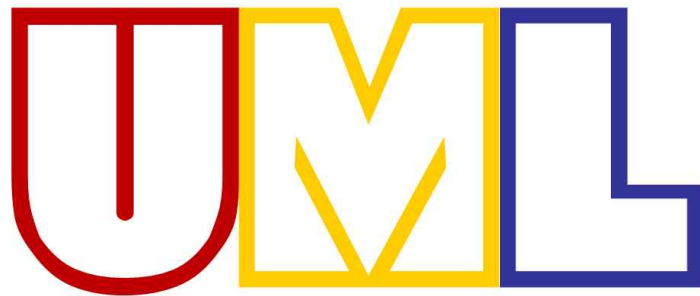


# 소프트웨어설계론

Software Design  
with



황석형

(c)sHwang, 2022.

이 자료의 무단복제/전제를 엄격히 금지합니다.



## 머리말

IT기술의 발전과 더불어서 소프트웨어산업은 21세기의 중요한 기간산업으로 자리매김되고 있습니다. 정보시스템은 현대사회의 중요한 사회적 인프라이며, 정보시스템이 없다면 국민생활과 기업활동이 성립되지 않는 것이 현실입니다. 정보시스템의 진화발전에 의해, 우리들의 생활은 상당히 편리해지고 있습니다.

가전제품, 휴대전화, 자동차 등, 우리들 주변에는 소프트웨어가 탑재되어 있지 않은 제품은 존재하지 않는다고 해도 과언이 아닐 것입니다. 그리고, 어떤 제품이나 서비스라도, 소프트웨어는 사용편의성과 편리한 기능 등이 중시되고 있으며, 소프트웨어의 품질이 제품과 서비스의 질을 좌우하므로, 결과적으로 국가 경쟁력을 결정하는 요인이 되고 있습니다.

특히, 소프트웨어산업의 성패를 좌우하는 열쇠는 IT 인재라고 생각합니다. 고품질의 소프트웨어를 고객이 원하는 형태로 개발하여 제공하기 위해서는, 고도의 소프트웨어를 설계하고 구현할 수 있는 기술자 육성이 무엇보다도 중요합니다.

이 책에서는, 객체지향의 기본개념을 토대로 가상 도서관관리시스템을 개발하기 위해 다양한 관점으로 주어진 문제를 분석하고 UML을 사용하여 설계함으로써, 전반적인 모델링 방법과 각 설계성과물 및 설계공정을 구체적이고 상세하게 설명하고 있습니다. 또한, 다양한 연습문제와 실습문제를 통해 문제해결능력과 UML의 활용능력을 동시에 습득할 수 있습니다. 물론, 이 책의 내용만으로 소프트웨어개발자가 갖추어야 할 모든 설계기술요소들을 망라하고 있는 것은 아닙니다. 따라서, 이 책에서 다루고 있는 내용들을 토대로 보다 다양한 소프트웨어공학 요소기술을 익혀가야 될 것입니다.

소프트웨어개발자는, 향후 정보화 사회를 이끌어갈 중요한 사회적 인재입니다. 독자여러분들이 소프트웨어개발자로서 갖추어야 할 필수적인 설계기술요소들을 배우고 익힌다면, 틀림없이 훌륭한 소프트웨어개발자로서 멋진 인생을 펼쳐나갈 수 있을 것이라고 생각합니다.

2022년 8월

sHwang



## 목차

제0장 객체지향의 기본개념 .....	1
0.1 서론 .....	1
0.2 객체지향의 개요 .....	4
객체지향의 기초지식 .....	4
객체지향기술의 장점 .....	6
객체지향으로 전환할 경우의 유의사항 .....	7
제1장 UML의 기본 .....	9
1.1 서론 .....	9
1.2 모델링이란? .....	9
소프트웨어개발과정 .....	9
모델링 .....	10
1.3 FUM .....	11
UML의 역사 .....	11
UML이란 .....	11
UML은 어디든지 적용할 수 있는가? .....	11
UML 도면의 종류 .....	12
Fractal .....	13
1.4 연습문제 .....	14
제2장 요구모델링 .....	16
2.1 시스템 개발 범위 .....	16
2.2 Use Case View .....	17
2.3 Actor .....	17
Actor의 추가 .....	19
2.4 Actor들 사이의 관계 .....	20
관련 .....	20
초기단계에서 너무 과다하게 표현하지 말 것 .....	21
2.5 Workflow .....	22
Activity Diagram .....	22
분기와 분기조건 .....	24
내장된 형태의 Activity .....	25
SwimLane .....	26
Object Flow .....	27
2.6 조작 및 동작의 완전성 .....	29
Walk Through .....	29
CRUD속성 .....	29
2.7 Use Case .....	30
Use Case를 찾아내는 방법 .....	32

Activity와 Use Case	33
Use Case View와 WorkFlow의 대응	34
Actor와 Use Case	34
Use Case들 사이의 관계	36
2.8 정적 모델과 동적 모델	37
2.9 Use Case 시나리오	38
2.10 Sequence Diagram	39
작성방법	39
책을 이용한다(Use Case 시나리오)	41
도면 작성 시 주의사항	43
Use Case Scenario의 주의사항	43
2.11 요구사항의 추가	45
2.12 테스트 시나리오로서의 Use Case Scenario	45
2.13 Use Case의 사전조건 및 사후조건	46
사전조건과 사후조건	46
다른 Use Case에 대한 사전조건과 사후조건	47
제3장 개념모델링	51
3.1 개념객체(Concept Objects)	51
3.2 “도서관관리시스템”의 개념객체	51
수장자료	51
이용자	53
대출과 예약	53
3.3 개념객체와 Use Case	54
3.4 생명주기(LifeCycle)	55
3.5 StateChart Diagram	56
3.6 각 개념객체의 생명주기	57
이용자의 생명주기	57
병행상태	60
왜 StateChart Diagram을 이용하는가?	60
[참고] 상태전이표	62
3.7 개념객체의 책임	63
위임과 협조상대	64
책임의 개수	65
3.8 개념객체와 Use Case Scenario	66
“시스템”의 내부에 존재하고 있는 개념객체	67
3.9 개념시나리오	67
“대출한다”의 개념시나리오	68
책임에 대한 체크	69
“반납한다”에 대한 개념시나리오	70
“예약한다”에 대한 개념시나리오	72

3.10 개념객체의 정적 구조.....	73
3.11 개념객체의 속성.....	75
기타 개념객체들의 속성.....	76
3.12 개념객체의 동작.....	77
동작의 매개변수.....	78
3.13 관련.....	79
관련에 이름을 붙이는 방법.....	80
다중도.....	80
3.14 집약과 합성.....	81
3.15 개념객체의 관련.....	82
관련에 대하여 고려해야할 사항들.....	83
3.16 시스템 전체는 1개의 객체.....	85
3.17 개념모델링의 종료.....	85
 제4장 사양모델링.....	 88
4.1 사양모델링.....	88
4.2 아키텍처.....	89
패턴.....	90
4.3 객체의 세련화.....	91
4.4 시스템 객체의 세련화.....	92
객체의 분해.....	92
실제로 분해해 보자.....	92
4.5 패키지.....	94
4.6 Concurrent Engineering.....	95
4.7 “이용자관리서브시스템”의 사양모델링.....	96
4.8 서브시스템의 인터페이스.....	96
4.9 UserManager의 인터페이스.....	97
동작명.....	97
매개변수.....	98
반환값.....	98
기타 동작.....	98
역동작.....	99
4.10 UserManager의 인터페이스(2).....	101
UML에 있어서 인터페이스.....	101
막대 사탕.....	103
4.11 서브시스템의 계약.....	104
4.12 UserManager의 계약.....	105
createUser의 사전조건/사후조건.....	105
deleteUser의 사전조건/사후조건.....	106
UML 도면에 계약을 기입한다.....	108
4.13 서브시스템의 내부구조.....	108

4.14 UserManager의 내부구조/사양요소 .....	109
객체의 세련화 .....	110
[참고] 추적가능성 .....	112
4.15 UserManager의 내부구조/구현요소 .....	112
4.16 서브시스템내의 협조 .....	114
4.17 협조(Collaboration) .....	115
createUser 협조작업에 User클래스와 allUsers객체가 참가 .....	117
4.18 사양모델링의 종료 .....	118





## 제0장 객체지향의 기본개념

### 0.1 서론

소프트웨어산업에 있어서, 특히, 대규모의 복잡한 소프트웨어개발에서는, (1)버그(Bug)가 없는 고품질의 소프트웨어를 개발, (2)고객의 요구사항변화에 유연하게 대응, (3)납기 내에 개발완료, (4)기존의 소프트웨어를 재이용 할 수 있어야 된다. 특히, 메인프레임(Main Frame)시대로부터 Client/Server환경을 거쳐서, 현재 Web기반 정보시스템에 이르기까지, 위의 4가지 사항들은 소프트웨어개발자들이 절대적으로 해결해야 할 과제로서 널리 알려져 왔다.

1970년대에는 이러한 문제점들의 해결책으로서, “구조화 분석/설계기법(Structured Analysis/Design Methodology)”이 제안되어, 상업용 소프트웨어개발분야의 주류를 이루어 현재까지 널리 사용되고 있다. 한편, 1980년대에 실리콘벨리에 있는 Xerox사의 Palo Alto 연구소(Palo Alto Research Center: PARC)에서 개발된 “객체지향기술(Object-Oriented Methodology)”은, 종래의 사고방식을 근본적으로 변화시키는 획기적인 소프트웨어의 개념으로서, 일부에서는 강력한 지지를 얻었으나, 메인프레임 전성기의 상업용 소프트웨어분야에서는 무시되어왔다. 그러나, 80년대 후반부터 클라이언트/서버시대를 맞이하여, 소프트웨어의 복잡화가 가속화되었고, 결국, 새로운 소프트웨어개발 패러다임(Paradigm)이 요구되었으며, 이에 대한 대안으로서 각광받게 된 것이 바로 “객체지향기술”이다.

객체지향기술은, 현재 많은 분야에서 소프트웨어개발기술로써 사용되고 있으며, 새롭게 발표되는 소프트웨어관련기술들 중에서 객체지향기술을 기반으로 하지 않는 기술을 찾아내기 어렵다. 객체지향의 대표적인 것으로서, Java와 C++ 등과 같은 프로그래밍언어들과 EJB와 .NET등과 같은 아키텍처가 있다. 데이터베이스분야에도 객체지향개념을 적용하는 사례가 증가하고 있으며, 소프트웨어개발방법론에도 객체지향을 전제로 하고 있다. 특히, 설계패턴(Design Pattern)과 프레임워크, 컴포넌트 등의 기술들 또한 객체지향 개념을 이용하고 있다. 이와 같이, 현재 소프트웨어업계에서는 다양한 형태로 객체지향기술이 널리 사용되고 있다.

객체지향기술을 소프트웨어 개발언어측면에서 살펴보면, 절차형 언어인 COBOL 및 C 등으로부터 객체지향언어인 Smalltalk, C++로 발전되어, 현재의 Java에 이르고 있다. 이와 같은 객체지향 개발언어에 대응하여, 90년대 초반부터 다종다양한 다수의 객체지향 소프트웨어분석/설계기법들이 제안되었다<sup>1)</sup>.

이와 같은 방법론과잉현상은, 결과적으로 일반 소프트웨어기술자들이 객체지향분석설계기법의 사용을 주저하게 되는 한 원인이 되기도 했다. 이러한 위기감을 느끼던 Grady Booch씨(Booch기법 창시자)가 이끄는 Rational Software사에서는, OMT(Object Modeling Technique)기법 창시자인 James Rumbaugh씨와 OOSE(Object-Oriented Software Engineering)기법의 Ivar Jacobson씨를 Rational Software사에 초빙하여, 통일화된 방법론 개발에 착수하였다. 그 결과물 중의 일부로서, 객체지향 방법론에 사용되는 표기법을 통일함으로써 객체지향 분석 및 설계분야에서 표준을 위한 기초로서 제안된 것이 바로 “통일모형화언어(Unified Modeling Language: UML)”이다. 이후, Rational Software사는 UML과

---

1) 한때는 50여개 이상의 기법들이 제안되어, “방법론전쟁(Methods War)”이라고 불리던 시기도 있었다.

	년대 80	85	90	95
開發方法	객체지향 분석/설계	Booch>기법		UML RUP OMG 표준화(97)
		OMT>기법		
	OOSE			
	Schlaer/Mellor>기법			
객체지향 프로그래밍	Smalltalk- 80			
	C++(85)			
				Java(95)
開發素材	Class Library	Smalltalk개발환경 NIHCL		
	Framework	MVC Interviews/ET++ IBM샌프란시스코		
	Design Pattern	Coad(92) Design Pattern(95)		
	컴포넌트 웨어 /분산객체			OLE1(91) OLE2(93) DCOM DNA VBX OCX ActiveX
				JavaBeans(97)
		CORBA 1.1(91) 2.0(94) 3.0(99)		

<그림 0-1> 객체지향의 역사

관련된 모든 권리를 포기하고, 중립적인 업계단체조직인 OMG(Object Management Group)에 제안하여 수락을 얻음으로서, UML은 전면적으로 공개된 표기법으로 제정되었고, 현재 국제표준으로서 인정되고 있다. 또한, UML은 Jacobson씨의 “Use Case기법”을 채용함으로써, 소프트웨어개발분야뿐만 아니라, BPR(Business Process Reengineering)분야의 분석기법에도 널리 사용되고 있으며, 종래의 커다란 갭이 지적되었던 경영분석과 IT분야 사이를 연결해주는 표현방법으로서 최근 비즈니스 컨설팅분야에서도 크게 주목받고 있다.

UML이 세계표준 모델링 언어로서 자리 잡은 이후, 많은 사람들이 UML에 대해 학습하였고, 프로젝트에 적용해 감에 따라, UML 사용 경험자들은 UML 자체로는 다소 부족하다고 느끼게 되어, UML을 객체지향 소프트웨어개발에 적용하기 위한 적절한 개발공정(Software Development Process)의 필요성을 느끼게 되었다. 이에, Rational Software사에서는, UML기반 객체지향 소프트웨어개발 프로세스로서, Rational Unified Process(이하, RUP)를 개발하였다. RUP는 개발조직의 작업 및 책임업무의 분담과 관리를 체계적으로 수행하는 방법을 제공함으로써, 계획된 기간과 예산 범위 내에서 사용자의 요구를 만족시키는 고품질의 소프트웨어 개발을 목표로 하고 있으며, UML과 더불어, 현재, 소프트웨어개발자에게 있어서 필수적인 기술로 인식되고 있다.

객체지향의 본질은, 현실세계에 대응시킬 수 있는 모델화 개념, 그리고 이러한 모델을 클래스로서 프로그래밍하기 위한 캡슐화 등의 체계를 프로그래밍 언어 및 개발방법론으로서 제공했다는 점에 있다. 그러나, 우수한 언어 및 방법론만으로 하루아침에 좋은 품질의 어플리케이션을 구축할 수 있는 것은 아니다. 더욱이, 소프트웨어가 거대화/복잡화됨에 따라서, 사용자의 요구사항파악 및 개발방법에 대하여 고도의 전문능력이 필요하게 된다. 우수한 어플리케이션을 개발하기 위해서는, 사용자의 요구 및 소프트웨어 개발방법에 대하여 장기간 기술이 축적되어야만 비로소 고품질의 소프트웨어개발이 가능하게 된다. 이와 같은, 기술을

형상화하여 추적할 수 있는 체계로서 컴포넌트지향기술이 탄생하게 되었다.

컴포넌트기반소프트웨어(Component Based Software) 또는 컴포넌트웨어(Componentware)란 의미는 일반적으로 객체지향기술의 원리를 이용하여 제작한 소프트웨어 모듈을 의미한다. 즉 기계 부품과 같이 소프트웨어도 하나하나의 부품으로 제작한 다음, 이러한 부품들을 조립하여 보다 복잡한 소프트웨어를 용이하게 만들 수 있을 것이라는 아이디어에서 시작한 개념이다. 이러한 단위기능의 소프트웨어 부품을 컴포넌트 소프트웨어라고 한다<sup>2)</sup>.

초기의, 이른바 제1세대 컴포넌트기술은 Windows상의 어플리케이션을 연계하여 보다 고수준의 어플리케이션을 구현하는 기구로서 제공되었다. OLE(Object Linking and Embedding)와 이를 지원하기 위한 COM(Component Object Model)이 대표적인 예이다. OLE/COM은, Client와 Server 사이에서 어플리케이션 및 부품들을 연계할 수 있는 DCOM(Distributed COM)과 OLE를 인터넷에 적용시킬 수 있는 ActiveX로 발전하였다. 한편, 기간업무를 중심으로 하여 Server측에서는 CORBA(Common Object Request Broker Architecture)를 이용하게 되었으며, Web과 Java의 출현에 의해, Web과의 연계 및 네트워크를 의식한 제3세대의 분산컴포넌트지향기술로 진화하고 있다.

대표적인 컴포넌트기반 소프트웨어개발방법론들<sup>3)</sup>의 공통점으로는 분석, 설계, 구현 시험 등의 기본 개발 단계가 점진적이고 반복적으로 수행된다는 것이다. 이는 결국 객체지향개발 방법론들이 지향해 왔던 개발프로세스의 특징들과 일치하는 것으로 컴포넌트기반 소프트웨어개발에 객체지향기술이 토대가 되고 있음을 잘 설명해 주고 있다.

Keyword	설 명
객체 (Object)	소프트웨어의 처리대상영역 속에 존재하는, 물리적/논리적/“사물”, “개념”. ·“속성(데이터)”과 이에 대한 “처리(메소드, 동작)”를 일체화시킨 것 ·“클래스”에 속하는 각각의 실체(Instance)
클래스 (Class)	객체 생성을 위한 Template(객체製造機) 같은 속성과 메소드를 갖는 객체들을 모아서, 그 특징을 정의한 것. ·클래스계층구조: 클래스를 속성 및 메소드를 기준으로 정리하여 체계화한 것 ·Tree구조 → 單一繼承(Single Inheritance) ·Network구조 → 多重繼承(Multiple Inheritance) ·SuperClass/SubClass : 클래스계층구조에 있어서, 上位/下位에 있는 클래스
계승 (Inheritance)	상위의 클래스(들)로부터, 속성 및 메소드를 하위클래스가 이어받는 것
다형성 (Polymorphism)	同一한 메시지가 서로 다른 객체에 수신되었을 때, 相異한 메소드를 실행함으로써, 각 클래스 특유의 “속성” 및 “메소드”를 정의할 수 있다는 개념.
정보은폐 (Information Hiding)	객체가 갖는 기능과 인터페이스만을 공개하고, 그 이외의 정보는 비공개한다는 개념
캡슐화 (Encapsulation)	객체의 내부에 데이터와 이에 대한 처리를 격납하고, 情報隱蔽를 구현하는 방법
메시지 전송 (Message sending)	객체에 대한 “처리” 의뢰
메소드(Method)	객체의 “처리”부분을 담당하는 함수(프로그램)로서, 메시지수신에 의해 起動됨.

<표 0-1> 객체지향 관련 키워드

2) CI Labs(Component Integration Laboratories, Inc.)의 제드 헤리스에 의하면 컴포넌트는 쉽게 만들고 또 유지보수 할 수 있을 정도로 적당히 작으면서 한편으로는 시장에 유통시켜 팔 수 있을 정도로 크며, 서로 동작성이 있는 표준 인터페이스를 지원하는 소프트웨어라고 정의하고 있다.

3) Catalysis, CBD96, SELECT Perspective, Rational Unified Process 등

이와 같은, 소프트웨어공학의 역사적 사실을 토대로, 이 장에서는, 객체지향의 기본개념에 대한 이해를 넓히고자 한다.

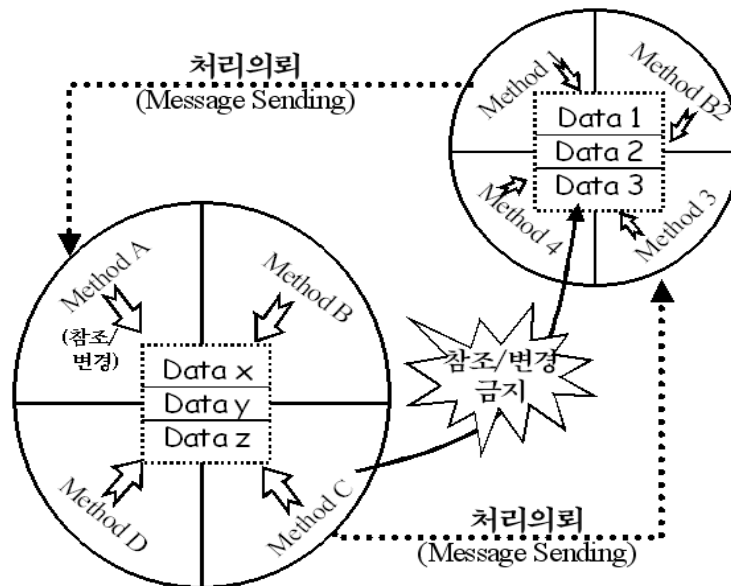
## 0.2 객체지향의 개요

### [객체지향의 기초지식]

객체지향을 이해하기 어려운 이유 중에 하나로서, 계승(Inheritance)이라든가 다형성(Polymorphism) 등과 같은 특수한 용어가 사용되기 때문이다. 또한, 객체 자체도 이해하기 어렵고 애매모호한 용어이다. 객체지향의 개념을 구성하는 주요 키워드(표0-1 참조)에 대해 살펴보도록 한다.

#### (1)객체(Object)와 정보은폐(Information Hiding)

객체란, 현실세계의 “사물” 및 역할 등을 추상화한 것이다. 소프트웨어에 있어서 객체는, 처리대상인 “데이터<sup>4)</sup>”와, 이러한 데이터를 참조/변경하는 “처리<sup>5)</sup>”를 일체화시킨 것이다. 또한, 객체지향(Object-Oriented)이란, 이와 같은 객체들을 기본구성단위로 하여 소프트웨어를 구축하는 체계이다. <그림0-2>에 객체의 기본적인 구조를 나타내었다.



<그림 0-2> 객체의 구조

이러한 구조를 캡슐(Capsule)이라고 부르며, 캡슐내의 데이터가 갖는 값에 의해서, 객체 자신의 상태와 특징을 나타낼 수 있다. 이와 같이 내부의 데이터 구조와 이에 대한 값, 그리고 메소드의 내용은 캡슐 내부에 은폐되어, 직접적인 참조/변경이 불가능하다. 이것을 정보은폐라고 부른다. 정보를 은폐함으로써, 다음과 같은 이점을 얻을 수 있다.

- 외부에 있는 객체에 의해, 객체내부의 속성이 부당하게 변경/참조될 위험성을 배제.
- 객체 내부를 변경하더라도, 외부의 다른 객체들에게 파급되는 영향을 차단.

4) 속성(屬性, Attribute) 또는 데이터멤버(Data Member)라고도 부름.

5) 메소드(Method) 또는 멤버함수(Member Function)라고도 부름.

정보은폐된 객체는, 메소드를 기동시키기 위한 인터페이스로서 메시지(Message)라고 부르는 프로토콜만을 캡슐외부에 공개하고 있다. 객체지향에 의해 분석·설계·구현된 시스템은, 객체들이 상호 메시지 송수신에 의해 처리를 의뢰함으로써, (객체간의 상호메시지수수작용에 의해)시스템을 동작시켜서 주어진 문제를 해결하도록 한다.

## (2)클래스(Class)와 인스턴스(Instance)

클래스는, 같은 속성(데이터구조)과 메소드(처리)를 갖는 객체들을 종합하여, 이에 대한 특징을 기술하여 정의한 것이다. 같은 클래스에 속하는 객체들 정의의 차이점은, 객체가 갖고 있는 속성의 값에 의해 결정된다. 객체를, 클래스에 대한 실체라는 의미로 “인스턴스(Instance)”라고 부르는 경우도 있다.

## (3)계승(Inheritance)

계승이란, 부모의 특성이 자식에게 상속되듯이, 클래스들로 구성된 계층구조를 정의하고, 상위의 클래스에 정의되어 있는 속성 및 메소드를 하위의 클래스가 이어받고, 또한, 하위의 클래스에서는 새로운 속성 및 메소드를 추가할 수도 있음으로써, 클래스의 재사용을 가능하게 해주는 기구이다. 공통적인 속성 및 메소드를 갖는 클래스들을 정리하여 상위클래스로 규정하고, 하위의 클래스들이 계승하여 재이용한다.

계승에는, 일반화-특수화(Generalization-Specialization)와 모듈의 확장이라는 2가지 의미가 있다. 일반화-특수화는, 클래스들 사이의 “is-a”관계라고도 부르며, 클래스들 사이의 의미적인 계승관계, 즉, 하위클래스의 인스턴스가 상위클래스의 인스턴스를 대체할 수 있는 집합의 포함관계가 성립하는 경우에 한하여 사용한다. 한편, 계승기능을 모듈의 확장으로서 사용하는 경우에는, 클래스 사이에 의미적인 포함관계가 없고, 상위클래스의 소스코드를 하위클래스에서 재이용하려는 목적으로 사용할 수 있다.

계승구조에 있어서, 하위클래스로부터 상위클래스를 가리킬 경우에는 슈퍼클래스(SuperClass)<sup>6)</sup>, 반대로 상위클래스로부터 하위클래스를 가리킬 경우에는 서브클래스(SubClass)<sup>7)</sup>라고 부른다. 또한, 계승에는, 단일계승(Single Inheritance)과 다중계승(Multiple Inheritance)이 있으며, 다중계승은 슈퍼클래스가 2개 이상 존재하는 경우를 가리킨다.

## (4)추상클래스(Abstract Class)와 구상클래스(Concrete Class)

추상클래스는, 개념을 정리하고, 서브클래스들 사이에 공통적인 특징으로서 규정하기 위한 범용적인 클래스이다. 따라서, 추상클래스로부터는 어떠한 인스턴스(객체)도 생성할 수 없다. 한편, 인스턴스가 존재하는 클래스를 구상클래스라고 부른다.

## (5)객체들 사이의 관계

객체지향 분석 및 설계에 있어서, 객체(또는 클래스)들 간의 관계는 매우 중요한 역할을 한다. 객체들 사이의 관계를 추상화하여 클래스들 사이에 정의함으로써, 클래스들로부터 생성된 각 객체들 사이의 메시지송수신에 의해 처리를 수행하기 위한 협조가 가능하기 때문이다. 객체들 사이의 관계는 다음과 같이 크게 2종류로 분류할 수 있다.

6) 어비클래스(Parent Class), 또는 基底클래스(Base Class)라고도 부름.

7) 자식클래스(Child Class) 또는 派生클래스(Derived Class)라고도 부름.

●관련(Association)관계: 객체들 간의 참조 및 이용관계.

●집약(Aggregation)관계: 관련관계의 일종으로서, 한쪽의 객체가 다른 쪽 객체의 부품이 되는 것과 같은 구조적인 포함관계(일종의 “part-of”관계).

집약관계는 다시, Aggregation관계와 합성(Composition)관계로 나눌 수 있으며, Composition은 Aggregation보다 강력한 형태로서, “전체”객체와 “부품”객체는 같은 생명 주기를 갖는다(Aggregation의 경우, “전체”객체가 없어져도 “부품”객체들은 생존할 수 있다).

#### (6) 다형성(Polymorphism)과 동적결합(Dynamic Binding)

다형성이란, 같은 이름의 메시지를 수신하더라도, 메시지를 수신하는 객체들에 의해 상이한 메소드를 기동시키는 특성을 가리킨다. 한편, 동적결합이란, 수신된 메시지와 이에 대응하는 메소드의 구현부분의 결합이 프로그램 실행 시에 이루어지는 기구를 말하며, 다형성과 조합하여 사용함으로써, 한 종류의 메시지를 사용하여, 기동되는 메소드의 차이점을 제거시킨 융통성 있는 모델을 구축할 수 있다.

#### [객체지향기술의 장점]

소프트웨어개발에 있어서, 객체지향기술을 도입함으로써, 다음과 같은 장점을 얻을 수 있다.

##### ●현실세계와 대응하는 좋은 구조를 갖는 소프트웨어시스템 구축가능

분석으로부터 설계를 거쳐서 구현에 이르는 개발공정에 객체지향을 적용하면, 현실세계의 사물과, 소프트웨어를 구성하는 객체를 순응적으로 대응시킬 수 있다. 객체들이 상호메시지수수작용을 통하여 전체적인 처리를 수행해 가는 객체지향의 기본개념은, 우리들이 일상적인 비즈니스를 수행하는 방법과 동일하므로, 현실세계의 상황변화에 대응하여 소프트웨어를 개발하고 유지보수할 경우에 유리하다. 또한, 각 공정들 간에 객체를 공통적인 기본단위로 하여 일관된 개발이 가능하다.

##### ●독립성이 높은 모듈개발 가능

객체지향에서는, 정보은폐의 개념을 토대로 캡슐화된 객체를 기본단위로 하여 소프트웨어를 구축하므로, 상호관련 있는 데이터와 처리를 일체화하여 관리함으로써, 복잡도를 감소시킬 수 있다.

##### ●개발공정들간의 연계가 수월

구조화기법에 의한 소프트웨어개발에 있어서는, 기능설계와 데이터설계가 분리되어 있고, 기능설계, 데이터설계, 프로그래밍 등의 각 공정들 사이에 겹이 존재한다는 문제점이 있다.

이에 대하여, 객체지향개발기법에서는, 분석/설계/프로그래밍 등의 모든 공정들이 클래스를 중심으로 한 객체에 의한 설계로 통일되어 있다. 즉, 상류공정에서 확정시킨 “객체”는, 이후의 각 공정을 거치면서 프로그래밍단계에까지 유연하게 연계된다.

##### ●다양한 재이용기술에 의한 생산성 향상

객체지향에 의한 생산성 향상의 가장 큰 요인은, 고도의 재이용성에 있다. 객체지향에서는 클래스의 계승, 프레임워크<sup>8)</sup>, 그리고 분석 및 설계 패턴<sup>9)</sup> 등을 이용하여 재이용성을 향상시키고 있다. 프로그래머 한사람이 코딩할 수 있는 step수는 같더라도, 기존의 성과물들을 재이용함으로써, 결과적으로 고도의 복잡한 시스템을 구축할 수 있다. 또한, 개발공정들 사이에 유연한 연계가 가능함으로써 개발공정 전체를 통해서 생산성을 향상시킬 수 있다. 물론, Java와 같은 객체지향언어를 사용하면, 보다 많은 생산성 향상을 기대할 수 있다.

## ●품질 향상

캡슐화와 정보은폐 등과 같은 객체지향이 제공하고 있는 특성들에 의해 품질을 향상시킬 수 있을 뿐만 아니라, 보다 경험이 풍부한 개발자들에 의해 개발된 재이용 가능한 컴포넌트를 반복적으로 사용함으로써 전체적인 품질향상을 기대할 수 있다.

### [객체지향으로 전환할 경우의 유의사항]

현재 객체지향기술을 기반으로 하는 소프트웨어개발이 주류를 이루고 있다. 실제로 지금 당장 객체지향기법으로 전환하기에는 몇 가지 난제들(구조화기법에 익숙한 개발자들의 의식 전환문제 등)이 있으므로 실천에 옮기기 곤란한 점이 존재한다. 그러나, 수천Line급 프로그램으로 모듈을 작성했던 시대로부터 구조화기법에 의한 시스템구축시대로 전환했던 역사가 있듯이, 앞으로는 구조화기법으로부터 객체지향기법으로 대전환되어 갈 것이며, 전환시키지 않으면 안된다고 생각한다. 이에 관하여, “변화하는 것”과 “변화시켜야 할 것”을 표0-2에 정리하였다.

변화하는 것	변화시켜야 할 것
<ul style="list-style-type: none"> <li>●시스템분석 및 설계 방법</li> <li>●프로그래밍언어 및 프로그래밍방법</li> <li>●컴퓨터 사용방법(CASE Tool 등)</li> </ul>	<ul style="list-style-type: none"> <li>●소프트웨어 관리방법</li> <li>●개발담당자의 사고방식</li> <li>●개발부서의 조직 및 체계</li> <li>●개발부서의 관리방법</li> </ul> <p>(한마디로 표현하여, “개발부서의 문화”)</p>

<표 0-3>객체지향으로 전환할 경우의 유의사항

8) 관련있는 클래스들을 구성하여, 미리 어느 정도 어플리케이션의 형태(半製品 상태)를 만들어 놓은 것. 프레임 워크內的 세부사항들은 미완성상태에 있으므로, 완전한 어플리케이션으로 만들기 위하여 계승기능을 이용하여 서브클래스를 정의하고, 서브클래스에 대하여 다양한 기법을 적용하여 세부사항들을 완성하여 완전한 기능을 갖춘 실행 가능한 다양한 어플리케이션으로 만들 수 있다.

9) 설계패턴(Design Pattern)은, 특정한 환경에서 일반적으로 자주 등장하는 설계 문제와 그 해답의 쌍을 기록한 것이다. 각각의 패턴은 특정한 설계 문제에 초점을 맞추어, 필요한 클래스와 그 객체들, 역할, 상호작용을 찾아 기술할 뿐만 아니라, 적용 가능한 범위와 적용 결과 및 장단점을 기록하고 있다.





# 제1장 UML의 기본

## 1.1 서론

Fractal UML Modeling(FUM)은 UML을 사용하여 소프트웨어를 작성하는 방법 중의 하나이다. 따라서 이 책에서는 다음과 같은 2가지 사항을 설명하기로 한다.

첫 번째로는, UML에 대하여, 그리고 두 번째로는 UML을 사용하여 소프트웨어를 작성하는 방법에 대하여 설명한다.

독자들 중에는, “소프트웨어 구축방법에 대하여 이미 충분히 이해하고 있으므로, UML에 대해서만 배우고 싶다.”라고 생각하고 있는 분들도 있을 것이다. 그러나 프로그래밍언어의 습득이 프로그램의 작성방법과 불가분의 관계인 것처럼, UML의 습득도 소프트웨어 작성방법과 분리하여 생각할 수 없음을 이해하기 바란다.

물론, UML을 사용하여 소프트웨어를 작성하는 방법은 매우 많이 존재한다. FUM도 그와 같은 방법들 중의 하나이지만, 특별한 방법이라고 말할 수는 없다. 부르는 명칭이 다소 상이할지는 모르겠으나, “Fractal”이란, 재귀적으로 반복하면서 사물을 만드는 것을 의미하며, Agile<sup>10)</sup> 소프트웨어개발방법으로서 자주 사용되는 방법이다. FUM은, Agile Software개발방법에 UML을 주입시킨 것으로서, FUM을 “모델중심의 Agile Software개발방법론”으로 생각해도 좋다<sup>11)</sup>.

UML이라고 하면 머릿속에 떠오르는 것이 모델링 툴이다. 모델링 툴이라고 하면, 멋있게 들리겠지만, 도형그리기 툴을 토대로 한 것으로서, 매우 고가제품으로부터 오픈소스형태로 배포되고 있는 제품에 이르기까지 다양하게 존재한다. 그러나 FUM에서는 모델링 도구의 이용은 전제로 하지 않는다. 물론, 간단히 이용할 수 있는 도구가 있으면 그것을 사용해도 좋지만, 노트 또는 빈 종이를 사용해도 상관없다. 가장 추천할만한 방법으로서, 커다란 화이트보드 또는 모조지를 사용하기 바란다. 일반적으로 모델링이란, 여러 사람들이 상호협조하면서 이루어지는 즐거운 작업이기 때문이다.

## 1.2 모델링이란?

### [소프트웨어개발과정]

지금까지의 소프트웨어개발에서는, “모델링”이라는 용어를 많이 사용하지 않았다. 소프트웨어를 작성할 경우,

1. 요구사항을 정의한다.

10) “Agile”이란, “기민한, 민첩한, 재빠른”이라는 의미를 갖는다. “Agile Software Development”란, “좋은 소프트웨어를 신속하게 군더더기 없이 작성하는 개발방법”을 뜻하는 것으로서, 최근에 상당한 주목을 받고 있는 소프트웨어개발방법이다. “eXtreme Programming” 및 “SCRAM(Software Configuration, Release And Management)”은 대표적인 Agile Software 개발방법론이다.

11) 원래, 이 책에서 설명하고 있는 것은, 일부분에 불과하다. UML에 대하여 잘 모르는 사람들도 이해하기 수월하도록 UML을 사용한 모델링 부분을 중심으로 설명한다.

2. 요구사항의 내용을 분석한다.
  3. 분석결과를 기반으로 설계한다.
  4. 설계결과를 토대로 프로그래밍한다.
  5. 작성된 프로그램이 요구사항에 맞는지 테스트한다.
- 와 같은 작업들이 주로 수행되었다.

지금까지의 소프트웨어 작성방법에서는 특히 프로그래밍에 초점을 맞추었다. 물론, 다른 작업도 중요하다. 예를 들어 구조화분석/설계 등과 같은 방법들은 지금까지도 자주 사용되고 있다.

그러나, 이와 같은 방법을 사용하여 성공적으로 마감한 소프트웨어개발프로젝트도 많이 존재하지만, “별로 신통치 않다”라는 느낌을 받고 있는 것도 사실이다.

지금까지의 방법이 신통치 않은 최대 원인은, 1부터 5까지 각 작업들이 적절하게 연계되지 못했다는 데 있다. 특히, 1부터 3까지 사용된 사고방식 및 표현방법과, 4의 프로그래밍에서 사용되는 사고방식과 표현방법사이에는 커다란 갭이 존재한다. 즉, 현실세계와 소프트웨어세계 사이의 연계가 원만하지 못하게 될 가능성이 증가하게 되었다.

모델링, 특히 객체를 사용하여 모델링을 수행하는 것은 어떠한 의미를 갖는 것일까? 이것은, 소프트웨어개발에 있어서 각각의 작업을 “공통적인 사고방식”을 사용하여, 가능한 한 갭이 발생하지 않도록 수합하려는 것이다.

여기에서, “공통적인 사고방식”이란, “객체”를 뜻한다. 현실세계로부터 소프트웨어세계에 이르기까지 개발대상이 되는 사물/개념 등을 모두 “객체”로서 취급한다. 객체라는 소재를 사용하여 현실세계의 모델을 만들고, 이것을 점차적으로 변형시켜서 소프트웨어세계에서 실제로 가동되도록 작성함으로써, 분석/설계와 프로그래밍 사이의 갭을 한층 줄여보려는 것이다.

## [모델링]

“모델”이란, 대상을 그대로가 아닌, 어떤 특정한 방향에서 어떤 특정한 관점에 맞추어, 이용자들이 이해하기 쉽도록 기술한 것이라고 할 수 있다. 플라스틱 모델 또한 모델이며, 지도도 모델이다. 그리고 CG(Computer Graphics)에 의한 표현물도 모델이다. 대상의 어떤 측면에 대해서는 매우 이해하기 쉽게 되어 있는 반면, 이외의 측면에 대한 정보는 무시한다. 이것을 “추상화(Abstraction)”라고 부른다.

소프트웨어개발에서도, 개발대상에 대하여 다양한 방향(관점)에서 모델을 만들어서 조합하거나 변형시킨다. 이와 같이 함으로써, 최종적으로 프로그램(이것 또한 일종의 모델이라고 할 수 있다)을 작성해갈 수 있다. 이와 같은 작업을 “모델링(Modeling)”이라고 한다.

위의 설명에 의하면 모델링은 프로그래밍을 포함하는 보다 큰 개념이라고 생각할 수 있다. 기존의 방법에 의하면, 분석/설계 작업과 프로그래밍 작업을 분리하여, 각각의 작업에 종사하는 사람들을 명확하게 구별할 수 있었다. 그러나 모델링이라는 관점에서는, 지금까지의 분석/설계 작업에서부터 실제로 가동되는 소프트웨어를 작성하는 작업에 이르기까지를 연속된 1개 작업으로 취급할 수 있다.

단, 아무리 “연속작업”이라고 하더라도, 모델링의 관점을 토대로 몇 가지 종류로 분할하여 생각하는 것이 일반적이다. 이 책에서는, “요구모델링”, “개념모델링”, “사양모델링”, “구

현모델링” 등과 같은 4가지 유형의 모델링을 토대로 설명한다.

## 1.3 FUM

### [UML의 역사]

UML이란, Unified Modeling Language(통일 모형화 언어), 즉 모델링언어의 일종이다. 모델링이란, 분석/설계 및 프로그래밍을 포함하는 소프트웨어 작성 작업이라고 설명했다. 따라서, 모델링언어는 소프트웨어개발과정 전체에서 사용되는 언어이다.

UML은 현재, 가장 표준적이며 광범위하게 사용되고 있는 모델링언어중의 하나이다. 원래 UML은 1994년경에 난립하고 있었던 객체지향분석/설계방법론들을 통일된 방법론으로 확립하기 위하여 Rational Software사의 Grady Booch와 Jim Rumbaugh가 활동을 개시한 것이 시발점이 되었다.

이후에 방법론을 통일시키기 어렵다는 사실로부터, 우선은 방법론에서 사용할 모델표기법을 통일시키는 방향으로 변화하였다. 그 결과, 1997년에 객체지향업계 최대단체인 OMG(Object Management Group)에 의해, UML1.1이 표준사양으로 채택되었다. 현시점에서 최신판은 UML2.1이다.

이상이다, 대략적인 UML의 역사이다. UML자신의 역사는 아직 10여년 정도에 불과한 상태라고 할 수도 있지만, 발 빠르게 변화하고 있는 소프트웨어업계에서 이미 10여년의 역사를 갖고 있다고 생각할 수 있다.

### [UML이란]

UML이 OMG표준으로 채택된 이래, 다수의 방법론들이 UML을 토대로 한 표기법으로 변경되었다. 따라서 현재는 대다수의 개발현장에서 UML을 사용하는 것이 기본사항이 되어 있다. 물론 방법론에 따라서는 UML에 독자적인 도면이나 사고방식을 추가하기도 하며, UML을 어떻게 사용할지는 방법론에 따라서 전혀 상이한 형태로 나타나고 있다.

아무튼, 중요한 것은, UML이 규정하고 있는 것은, “모델의 표기법에 관한 토대”뿐이라는 사실이다. 어떻게 모델링할 것인가, 언제 어떠한 방법으로 도면을 그릴 것인가에 관한 사항은 UML에서는 아무것도 규정하고 있지 않다. 또한, UML을 토대로 하여 다양한 확장을 할 수 있는 메카니즘이 제공되고 있어서 실제로 사용방법에 따라서 UML을 확장할 수도 있다.

그렇다면 UML의 표준은 무엇에 도움이 되는 것일까? UML에 의해 작성된 도면을 살펴보면 대략적으로 무엇을 표현하고 있는 모델인지를 누구라도 이해할 수 있다는 점이다. 또한, 이와 같은 표기법이 있다면, 최저한의 라인까지는 서적이나 관련 강좌 등을 통해서 스스로 학습할 수 있다. UML의 이용자 수가 증가함에 따라서, 관련서적이나 세미나, 도구 등도 보다 많이 그리고 보다 저가에 공급되고 있다.

### [UML은 어디든지 적용할 수 있는가?]

UML은 소프트웨어개발 분야뿐만 아니라 다양한 영역에 적용할 수 있다. UML은 객체모델의 표기법으로서, 바꾸어 표현하면, 실세계에 존재하는 사물/개념을 모두 객체로 생각할

수 있다면, 실세계에 존재하는 모든 사물/개념을 UML로 표현할 수 있다.<sup>12)</sup>

실제로 OMG는, 실세계의 모든 영역에 대하여 UML을 사용한 객체모델을 구축하려고 하고 있다. 예를 들면, “항공관제”라는 영역을 객체를 사용하여 UML로 모델화할 수 있다면, 수월하게 항공관제시스템을 구축할 수 있게 되며, 항공관제시스템들 사이에 공통적인 언어로 통신하는 것이 가능하게 된다.

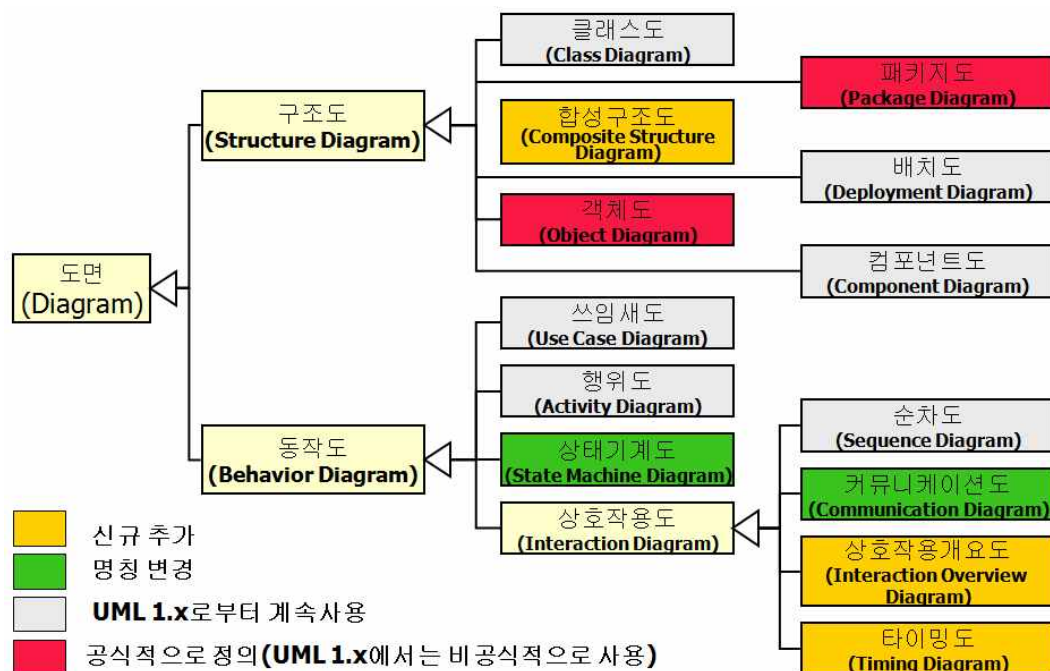
그 밖의 다른 영역에 있어서는 마찬가지이다. 일단 어떤 영역을 객체모델화할 수 있다면, 해당하는 객체모델을 자동적으로 변환하여 소프트웨어를 구축할 수 있게 된다.

이와 같은 OMG의 야망을, MDA(Model Driven Architecture), 모델 구동형 아키텍처라고 부르고 있다. 이와 같은 야망이 결실을 맺을 것인가 여부는 아무도 모른다. 단, 아직 시작단계에 불과하다. 지금까지 유사한 도전이 많이 시도되었으나, 성공하지 못하고 있음은 사실이지만, 이번에도 지금까지와 마찬가지로 실패할 것이라고는 단언하지 못할 것이다.

앞에서 설명한 것은, “프로그래밍언어를 대체할 수 있는 모델링언어”라는 꿈과 같은 이야기이다. 확실히 지금의 프로그래밍언어들의 대부분은 모델을 기술하는 언어로서는 제약사항이 너무 많다. 모델링언어가 현 상태에서 좀 더 진화하면, 프로그래밍언어는 필요 없게 되지 않을까라고 생각하고 있는 사람들이 많은 것이 사실이다.

## [UML 도면의 종류]

UML을 이용하여 모델링을 수행하는 사람의 관점에서 대략적으로 설명하자면, UML은 다음과 같은 13가지 도면(Diagram)으로 구성되어 있다.



<그림 1-1> UML2.x의 도면들

12) 이와 같은 경우, 실제로는 UML의 토대가 되며 OMG에서 표준화되어 있는 MOF(Meta-Object Facility)가 사용된다. UML은 실제로 MOF 위에서 작성된다. MOF를 사용하여 객체모델을 XML(eXtensible Markup Language)로 표기하는 형식도 OMG에서 규정하고 있다. 이와 같은 형식은 XMI(XML Metadata Interchange)라고 부른다.

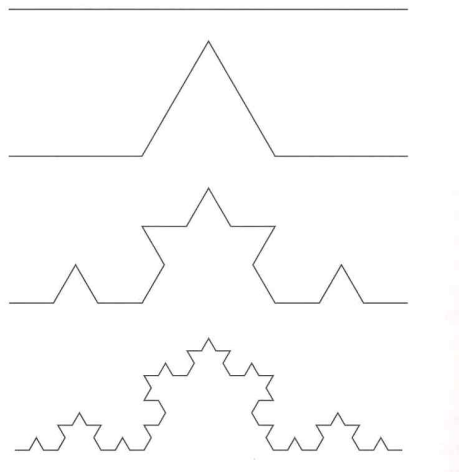
이 책에서는 위의 13종류의 도면들을 하나씩 열거하면서 상세하게 설명하지 않기로 한다. 그 대신에, 모델링을 초기단계부터 최종단계까지 수행하는 전 과정을 추적해 가면서, 어느 단계에서 어떠한 도면을 어떻게 사용하는가를 그때그때 설명해 가기로 하겠다.

UML의 해설에 대해서도, 그때그때의 상황에 따라서 필요한 만큼만 해설하기로 한다. 만약 UML 전체에 대한 상세한 사양에 대해서 알고 싶은 경우에는, OMG에서 공개하고 있는 UML 사양서를 참고하기 바란다.

## [Fractal]

FUM은 Fractal UML Modeling의 약칭이라고 설명한 바 있다. 여기서 Fractal이란 무엇일까? 상세한 부분까지 설명하게 되면 복잡해지므로 대략적으로 살펴보기로 한다. Fractal이란, 도형의 일부분을 어디까지 확대하더라도 그 내부에 동종의 복잡한 구조가 내장되어 있는 도형을 의미한다.

위와 같은 Fractal의 정의가 이해하기 어렵게 느껴지겠지만, Fractal에 근사한 도형은 실제로는 우리주변에 다양하게 존재하고 있다. 예를 들면, 해안선, 나무의 형태 등과 같은 자연계에 존재하는 다양하고 복잡한 형상들은, 실제로는 Fractal과 같은 것들이다.



<그림1-2> Fractal의 예

<그림1-2>는 “선분을 3등분하여, 그중에서 가운데 것을 집어 올린다.”라는 간단한 규칙을 반복하여 작성된 fractal의 예이다.

실제로는 이와 같은 규칙적용을 무한반복하게 되면 fractal도형이 된다.

FUM도 이와 같은 이미지를 적용하고 있다. 모델링의 대상이 되는 현실세계는, 해안선과 마찬가지로 복잡하고 변화무쌍하다. 그러나 여기에는 어떠한 법칙이 존재하고 있어서, 이러한 법칙을 잘 찾아내면 매우 아름다운 형태를 발견해 낼 수 있게 된다.

이러한 방법으로서, 단순한 직선을 조금씩 변형하는 작업을 수없이 반복해 가는 것이다. 물론, 무한반복은 불가능하지만, 실제로는 소프트웨어로서 가동시킬 수 있는 정도의 정밀도

를 갖추게 될 때까지 반복하게 되는 것이다.

이와 같은 방법이, 실제로 얼마큼이나 잘 적용되고 있는지, 적용한다면 무엇에 대하여 어떠한 작업을 fractal하게 반복하면 되는지, 순차적으로 검증해 보도록 하자.

## 1.4 연습문제

이 책에서는, 전체를 통하여 한 가지 예제를 사용하여 해설한다. 어떠한 분야의 예제를 선정하는가에 따라서, 논의되는 내용이 달라지겠지만, 이 책에서는 비즈니스 어플리케이션 분야의 예제를 사용하기로 한다.

우선, 이러한 예제를 통하여, UML자체에 대해서, 그리고 UML을 사용한 소프트웨어 작성 방법에 대해서도 대략적으로 이해할 수 있을 것이다.

이 책에서 다루게 될 예제는, A시립도서관의 정보관리시스템이다. A시립도서관에서는, 아직 도서관 직원이 카운터에서 서적 대출목록을 대조하면서 대출업무를 수행하고 있으며, 전산화는 아직 이루어지지 않은 상태이다. 이와 같은 시립도서관을 전산화하는 것이 당면과제이다.

도서관의 정보관리시스템에 관한 구체적인 내용은 매우 다양하다.

- 수장자료의 관리
- 등록된 주민들에 대한 수장자료 대출
- 업자들에게 대한 수장자료 발주
- 인접한 도서관과의 상호이용제도

등과 같이 매우 다양한 업무가 포함되어 있다. 이러한 업무들을 한 번에 전부 전산화하면 좋겠지만, 상당한 개발비용을 필요로 하며, 더욱이 각종 위험요소들이 산재해 있다.

따라서, 우선은 필요성이 높은 부분부터 전산화하기로 한다.

A시립도서관에서는 우선, 수장자료의 대출업무를 전산화하기로 결정하였다. 카운터에서 도서관 직원이 단말기를 사용하여 대출업무를 수행하며, 장차 이용자가 스스로 단말기를 조작하여 자택 등지에서 원격으로 예약할 수 있도록 하고자 한다.

대출에는 여러 가지 규칙이 존재한다. 한 번에 대출할 수 있는 수량에 대하여, 서적의 경우에는 10권까지이고, 음악 및 영상물에 대해서는 5개까지이며, 대출기간은 2주일 이내로 한정되어 있다. 이러한 규칙은 도중에 변경될 가능성도 있다. 대출업무를 전산화하기 위하여, 수장자료의 등록 및 이용자 등록도 필요하게 될 것이다.

우선, 어떠한 시스템을 구축해야하는가에 대하여 대략적인 윤곽을 살펴보는 것부터 시작하기로 하자.





## 제2장 요구모델링

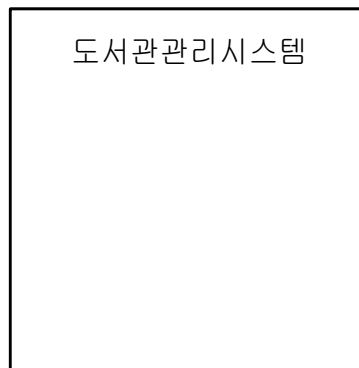
### 2.1 시스템 개발 범위

소프트웨어 시스템 개발에 있어서, 가장 먼저 해야 할 작업으로서,

- 개발하려는 시스템은 어떤 것인가?
- 이번 개발에서는 어디까지 구현할 것인가?

등을 명확하게 해야 한다. 그러나 이와 같은 작업들은 간단히 수행될 수 있는 작업들이 아니다. 실제 개발현장에서는, 시스템의 범위 및 형태가 최종 단계에 이르기까지 명확하지 않거나, 계속적으로 변경 및 수정되는 경우가 빈번하게 발생한다.

그러나, 무계획적으로 시스템을 구축할 수는 없다. 우선, 커다란 사각형을 그리고 나서 이름을 부여하는 작업부터 시작하도록 한다. 이름은 무엇이 되든 상관없다. 그려진 사각형이 지금부터 개발하려는 “도서관관리시스템”에 대한 가장 대략적인 형태가 된다.



<그림 2-1> 시스템을 나타내는 사각형

이와 같은 사각형이, 시스템개발의 출발점이 된다. 이와 같은 사각형을 “fractal”로 변형시켜서 현실세계 시스템의 형태에 근접시켜가는 것이 소프트웨어 모델링의 주요작업이다.

시스템의 형태는 크게 나누어서 2가지 힘에 의해 결정된다.

첫 번째는, “개발하려는 시스템이 현실세계에서 어떻게 사용되는가?”라는 관점에서 작용하는 힘이다. 즉, 외부로부터의 관점이다. 이와 같은 힘이 적절히 활성화되지 않으면 아무리 잘 만들어진 시스템이라고 하더라도 현실세계와 괴리된 시스템이 되어 버린다.

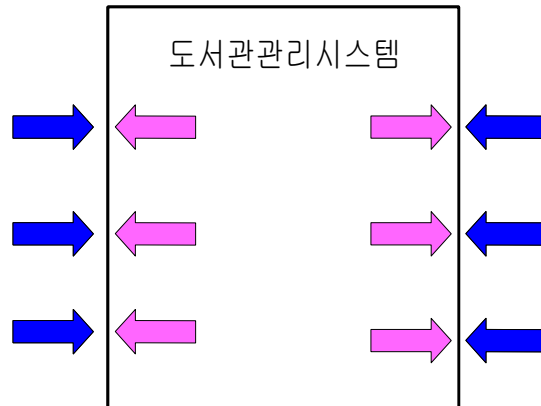
두 번째는, “개발하려는 시스템을 어떻게 구축하는가?”라는 관점에서 작용하는 힘이다. 즉, 내부로부터의 관점이다. 이와 같은 힘이 약하게 되면 아무리 우수한 아이디어라도 구현할 수 없게 된다.

이와 같은 2가지 힘, 즉, 시스템 외부로부터의 힘과 내부로부터의 힘이 적절히 균형을 맞추게 되면 비로소 “좋은 시스템”이 구축된다. 2가지 힘 중에서 어느 한쪽이 보다 강하거나 또는 보다 약하게 되면 좋은 시스템이 완성될 수 없게 된다.

물론, 실제 시스템은 그림과 같이 단순한 직선으로 만들어지는 것은 아니며, 매우 복잡한

형태를 취하게 된다.

그러면, 지금부터 이와 같은 단순한 직선들을 조금씩 자연스러운 fractal 경계로 변화시켜 보기로 하자.



<그림 2-2> 시스템을 개발하는 2가지 힘

## 2.2 Use Case View

앞서 설명했던 2가지 힘들 중에서, 우선 외부로부터의 힘, 즉, 사용자 및 현실세계로부터의 관점에서 시스템을 살펴보기로 하자<sup>13)</sup>. 주요 관점은 다음과 같은 2가지이다.

- 이 시스템에 관련 있는 사용자는 누구인가?
- 사용자는 이 시스템을 어떻게 사용하는가?

이러한 관점을 “Use Case View”라고 부른다. UML에서는 주로 “Use Case Diagram”이라는 도면을 중심으로, “Activity Diagram” 및 “Sequence Diagram”과 같은 도면들을 보조적으로 사용하여 Use Case View를 표현한다. 앞 절에서 그렸던 사각형(<그림2-1>)은 Use Case Diagram의 일부분이다.

Use Case View를 보다 명확하게 나타내기 위하여, 실제 시스템개발에서는 개발자들뿐만 아니라 사용자 및 고객의 참여가 중요하다. 사용자 및 고객은 소프트웨어에 대해서 초보자일 수도 있으나, 해당분야(도서관 운영)에 관해서는 전문가 중의 전문가이다<sup>14)</sup>.

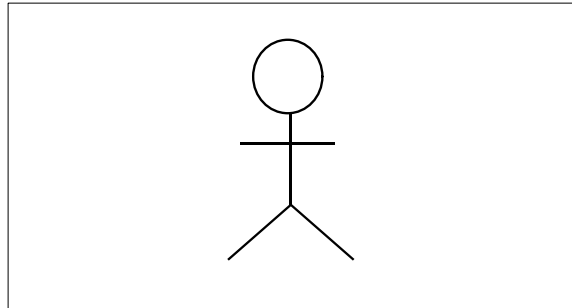
개발자는, 사용자와 고객들이 명확하게 이해할 수 있도록 Use Case Diagram을 작성해야 한다.

## 2.3 Actor

앞에서 열거한 2가지 관점들 중에서 첫 번째 관점, 즉, 개발하려는 시스템에 관련된 사용자를 “Actor”라고 부른다. Actor는 다음과 같은 아이콘(“Stickman”이라고도 부름)으로 나타낸다.

13) 이것은 개발자의 입장을 우선하는 것이 아니라, 실제로 사용하게 될 사용자들의 입장을 우선시하려는 것이다.

14) “해당분야”를 구체적으로는 “Domain”(문제영역)이라고 부르는 경우도 있다. 앞으로의 소프트웨어개발에서는 이와 같은 Domain에 가장 주목해야할 필요가 있다.



<그림 2-3> Actor

이와 같은 아이콘의 하단부에, 해당 사용자의 역할을 나타내는 이름을 붙여보자.

“이름을 붙이는 것”은, 매우 어려운 작업이다. 그러나 모델링을 하는 경우, “이름을 붙이는 작업”은 실제로는 본질적인 작업이다. 모델링이란, 절반정도는 어휘력 문제라고 해도 과언이 아닐 것이다. 그러나 이름을 붙이기 위한 작업에 30분씩이나 소요된다면 의미가 없으므로, 처음단계에서는 부담 없이 이름을 붙이기로 한다. 그리고, 나중에라도 보다 더 좋은 이름이 떠오를 경우에는 언제든지 이름을 변경시키도록 한다.

Actor가 나타내는 것으로서는, 개발하려는 시스템에 직접이해관계가 있는 사용자뿐만 아니라, 간접적으로 개발하려는 시스템과 이해관계가 있는 사람(예를 들면, 시스템의 출력결과를 살펴보는 사람, 시스템을 관리하는 사람 등) 또는 조직이나 단체 등을 Actor로 선정할 수 있다. 더욱이, 개발하려는 시스템과 통신하는 다른 시스템들 등도 Actor로서 선정할 수 있다.

#### <문제>

“도서관관리시스템”의 Actor는 무엇일까? 우선, Actor가 될 수 있는 후보들을 간단한 설명을 곁들여서 열거해 보기로 한다. 또한, Actor그림도 같이 그려보기로 하자.

#### <해답>

“도서관관리시스템”의 Actor들은 다음과 같다.

##### (1)이용자

- 도서관을 실제로 이용하는 사람
- 도서관의 소장자료를 열람하거나 대출하는 사람.

##### (2)도서관 직원

- 이용자에게 서비스를 제공하는 사람.
- 도서 대출 및 검색을 도와주는 사람.
- 소장자료의 등록과 같은 내부 작업을 수행하는 사람

##### (3)도서관

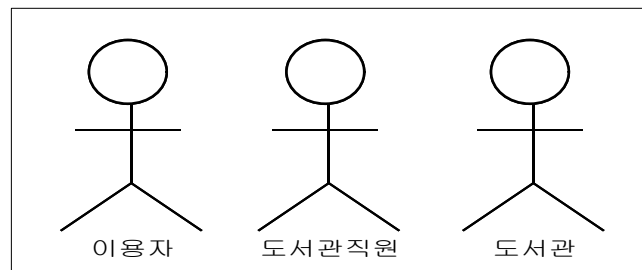
- 이용자에게 서비스를 제공하는 조직
- 도서관직원은 도서관에 소속되어 있음.

이와 같은 Actor 및 관련설명을 표 형태로 정리하면 표2-1과 같다. 반드시 Use Case Diagram과 별도로 아래와 같은 표를 작성해야 하는 것은 아니다. 그러나 시스템의 규모가 커지게 되면 모델요소들에 번호를 부여하여 일목요연하게 정리할 수 있으면 도움이 된다.

Actor #	이름	설명	비고
1	이용자	도서관을 실제로 이용하는 사람	-
2	도서관직원	이용자에게 서비스를 제공하는 사람	도서관에 소속됨.
3	도서관	이용자에게 서비스를 제공하는 조직	-

<표 2-1> “도서관관리시스템”의 Actor

다음으로는, 위에서 열거한 Actor들을 그림으로 나타내 보자.



<그림 2-4> “도서관관리시스템”의 Actor그림

이와 같은 Actor들이 완전하다고는 말 할 수 없다. 자신이 작성한 것과 다소 상이하다고 생각하는 독자들은 앞에서 설명한 Actor의 조건들을 참고해 보기 바란다.

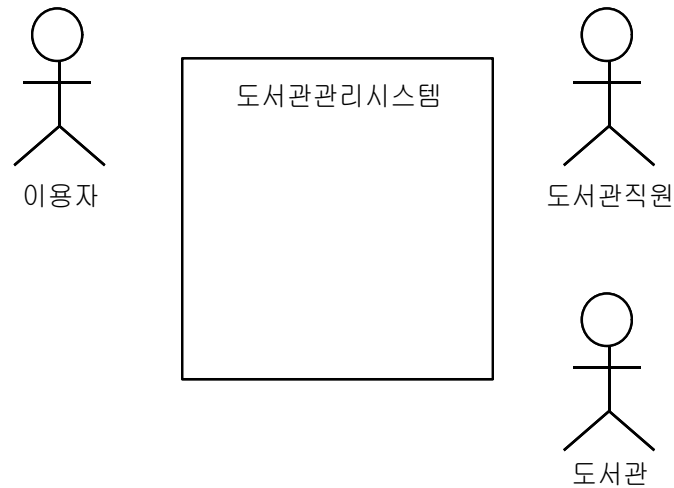
<문제> “수장자료”는 Actor인가? 그리고, 그와 같이 판단한 이유는 무엇인가?

<해답>

“수장자료”는 Actor가 아니다. 그 이유는, 수장자료는 사람도 아니고 조직도 아니기 때문이다. Actor가 되기 위해서는 “Actor”라는 이름이 의미하듯이, 시스템의 외부요소로서 존재하는 능동적인 사물/개념이어야 한다.

#### [Actor의 추가]

Actor를 발견하였다면, <그림2-1>의 시스템을 나타내는 사각형 외곽에 추가한다. 그 결과는 다음과 같다.



<그림 2-5> Actor를 추가한 Use Case Diagram

## 2.4 Actor들 사이의 관계

UML을 사용한 모델링은 기본적으로, “상자”를 그리고 나서 그 사이를 “선”으로 연결하는 작업의 반복이다. “상자”는 개발하려는 시스템, “어떤 관점에서, 어느 정도로” 살펴보았을 때의 주요한 요소들, 즉, 구성단위이다. 상자에는 여러 가지 종류가 있으며 아이콘도 각각 상이하다. 예를 들면, 앞서 그려 넣었던 Actor들은 “시스템에 관련되어 있는 외부요소”로 정의된 상자이다.

상자를 몇 개 그려 넣는 것만으로는, 가시적인 표현방법을 사용하는 장점이 없다. 요소와 요소(상자와 상자)들 사이에는 어떤 관계가 존재할 것이다. 그렇지 않다면, 각 요소들이 제각기 흩어지게 되어 시스템을 구성할 수 없게 된다.

따라서, 상자와 상자(요소와 요소) 사이에는 선을 그어서 어떤 관계가 존재함을 한눈에 알아볼 수 있도록 한다. 이것은 일반적인 프로그래밍언어에서는 찾아볼 수 없는, 모델링언어만의 특징이다.

### [관련]

다시 한 번, 앞에서 작성한 Use Case Diagram을 살펴보자.

<그림2-5>에는, 어떤 Actor들이 있는지는 알 수 있지만, 각각의 Actor들이 제각기 분산되어 있다. 만약 Actor들 사이에 어떤 관계가 존재한다면 선을 그어서 표현해 보기로 한다.

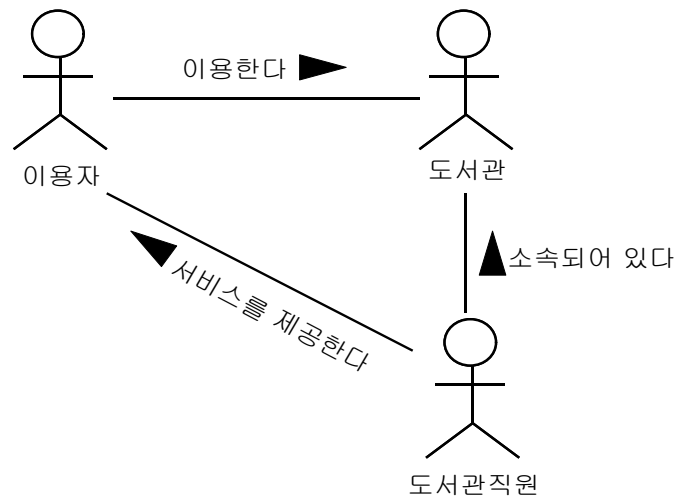
단, 너무 상세한 관계를 밝혀내기 위해 몰두하게 되면, “결국, 실세계의 사물들 사이에는 모두 관계가 존재한다!!!”라는 모델링 중독증에 빠지게 되어버린다. 관계가 존재하지 않는 사물이란 존재하지 않기 때문이다.

<그림2-5>의 경우, Actor는 3개뿐이므로, 각각의 관계를 살펴보면 다음과 같다.

- 도서관과 도서관직원 사이의 관계 : 도서관직원은 도서관에 소속되어 있다.
- 도서관직원과 이용자 사이의 관계 : 도서관직원은 이용자에게 서비스를 제공한다.
- 이용자와 도서관 사이의 관계 : 이용자는 도서관을 이용한다.

만약, 이와 같은 관계들이 중요하다고 판단되면, Actor와 Actor 사이를 실선으로 연결한다. 실선으로 연결된 모델요소들 사이의 관계를 “관련(Association)”이라고 부른다. 관련이란, “상대방에 대해서 알고 있다”, “어떠한 상호수수작용이 존재한다.”와 같은 의미를 갖는다. 상세한 사항은 나중에 설명하기로 한다.

선으로 연결된 선분위에, 어떠한 관계인가를 나타내기 위한 레이블을 기입한다. 방향성이 존재하는 경우, 다음과 같이, 주어에 해당하는 쪽에서 시작되는 삼각형 화살표머리를 표시해 두면 이해하기 수월해 진다.



<그림 2-6> Actor들 사이의 관련

#### [초기단계에서 너무 과다하게 표현하지 말 것]

현재 작성하고 있는 “도서관관리시스템”에서, 서적 및 CD 등을 판매하는 판매업자, 상호 이용협정을 체결하고 있는 다른 도서관 등도 Actor가 된다.

그러나, FUM에서는 “초기단계에서 모든 것을 갖춘 시스템”을 작성하지 않도록 한다. 개발하려는 기능을 제한하여, 한정된 기능을 구현할 수 있게 되면, 한걸음씩 진척시켜 가도록 한다. 초기단계부터 모든 것을 다 하려고 한다면, 실패할 가능성이 높아진다. 또한, 모든 기능을 다 갖춘 완벽한 시스템이 완성될 무렵에는, 초기단계에서 생각했던 것과 전혀 다르게 상황이 변화될 수도 있다.

기존의 개발방법에서는, 이와 같이 “점차적으로 기능을 추가해 가는 방식”을 사용할 경우, 매우 혼란스러운 상태로 전락하여 실패해 버리는 경우가 대부분이었다.

FUM에서는 이와 같은 방식을 사용하더라도 대부분 성공하게 되는 이점을 가지고 있다. 그 이유는 다음과 같다.

첫 번째 이유로서, 시스템 구축을 위한 소재로서 FUM에서는 객체를 사용하며, 객체 스스로가 기능추가 및 변경에 비교적 강력한 힘을 발휘한다.

또 다른 이유로서는, 현실세계를 반영한 모델링(Domain Modeling)을 수행한다는 점을 손꼽을 수 있다. 현실세계를 반영한 모델을 토대로 개발된 시스템은 기능추가 및 변경이 수월하다.

따라서, 현재 상태에서는, 이용자가 도서관의 소장자료를 이용하는 부분에만 집중하도록 한다.

## 2.5 Workflow

### [Activity Diagram]

Actor가 어떤 목적을 달성하기 위하여, 실제로 어떻게 행동하는가를 나타낸 것을 “Workflow”라고 부른다. UML에서는 “Activity Diagram”을 사용하여 나타낸다.

한 개의 Activity Diagram에는 Actor가 갖는 일련의 “행동흐름” 1개를 표현한다. 모든 Workflow를 한 장의 Activity Diagram에 전부 그려 넣게 되면 거대한 도면이 되어 혼란스러워진다. 따라서 처음에는 대략적으로 전체적인 흐름을 추적할 수 있도록 표현한다.

도면을 그리면서, “여기는 좀 더 상세하게 생각해 보자”라고 생각했다면, 그 부분만 확대한 workflow를 그려보도록 한다.

예를 들면, “도서관관리시스템”에서는 다음과 같은 3가지 커다란 workflow를 생각할 수 있다.

- 어떤 이용자가 서적을 찾아내어, 대출한 뒤에 반납하는 workflow
- 이용자가 이용자등록을 하고, 때로는 등록정보를 변경하며, 마지막에는 등록을 말소하는 workflow
- 어떤 서적에 대한 구입요청이 있어서, 발주하여, 납품되고, 이용되어, 체크결과, 마지막으로 폐기되는 workflow

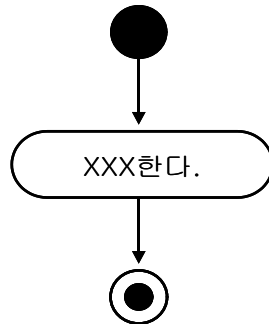
Activity Diagram을 그리기 전에, 맨 처음의 workflow(어떤 이용자가 서적을 찾아내어, 대출한 뒤에 반납하는 workflow)를 나열해 보자.

1. 이용자가 도서관에 도착한다.
2. 대출하려는 서적을 검색한다.
3. 만약, 해당 서적이 도서관에 있으면, 대출한다.
4. 집으로 가져가서 읽는다.
5. 대출기한이 지나기 전에 반납하기 위하여 도서관에 간다.
6. 서적을 반납한다.

Activity Diagram의 작성방법은, 각각의 업무 단위를 Activity 또는 Action상태라고 부르는데, 모서리 둥근 사각형으로 나타낸다. 그리고 실제로 수행되는 행위(동작 또는 작업이라고 부르기도 한다)를 간결하게 사각형 내부에 기입한다. 일반적으로 “XXX한다.”와 같은 형태로 기입하면 충분하다.

또한, 이와 같은 사각형을 시간 경과에 따라서 화살표로 연결한다. 이것을 Activity들 사이의 “전이(Transition)”라고 부른다.

가장 처음 수행되는 Activity에는 둥근 사각형 대신에, 시작을 의미하는 검은 원형 아이콘으로 표현하고, 종료를 나타내는 경우에는 2중 원형 아이콘을 사용한다(<그림2-7> 참조).

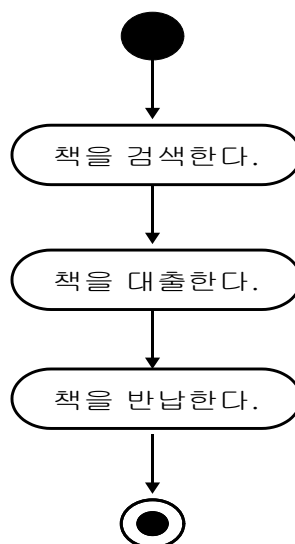


<그림 2-7> Activity Diagram

<문제> “도서관관리시스템”에서, 다음과 같은 Workflow를 Activity Diagram으로 작성하십시오.

1. 이용자가 도서관에 도착한다.
2. 대출하려는 도서를 검색한다.
3. 만약, 해당 도서가 도서관에 있으면, 대출한다.
4. 집으로 가져가서 읽는다.
5. 대출기한이 지나기 전에 도서관에 반납하러 간다.
6. 도서를 반납한다.

<해답>



<그림 2-8> 도서관 이용에 관한 Workflow



위의 Activity Diagram은 Flowchart와 매우 유사하다. 오히려 같은 것으로 생각해도 상관 없다. 위의 문제에서 설명한 Workflow들 중에서 1, 4, 5는 “도서관관리시스템”의 본질과는 관계없으므로 생략하였으나, Activity Diagram에 추가하여도 틀린 답은 아니다.

도서검색을 수행하지 않는 경우도 있을 수 있다. 개가식 서가로부터 수장자료를 직접 가져와서 대출하는 도서관도 있기 때문이다. 그러나, 이와 같은 상세한 사항은 위의 Workflow에 추가하지 않는 것으로 한다.

### [분기와 분기조건]

그러나, <그림2-8>에 그린 것과 같이 모든 것이 수월하게 작성되지 않을 수 있다.

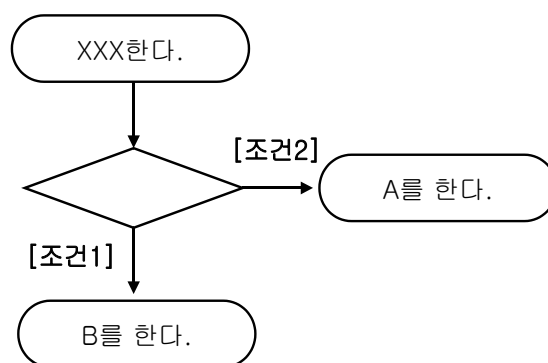
“도서를 검색”한 결과, 해당 도서가 도서관에 없을 경우 어떻게 하면 좋을까? 충분히 있을 수 있는 일이다.

만약 대출중이라면, 예약하면 될 것이다. 그리고 만약에 도서관에서 소장하고 있지 않는 경우라면, 근처 도서관에 문의해 보거나, 구입희망신청을 하면 될 것이지만, 현 단계에서는 아직 대처하지 않기로 한다. 우선, 예약하는 경우만을 생각해 보자.

Activity Diagram속에서 행동의 흐름이 도중에 분기할 경우에는, 다이아몬드 모양의 기호를 사용한다. 분기 조건은 [ ]속에 기입한다.

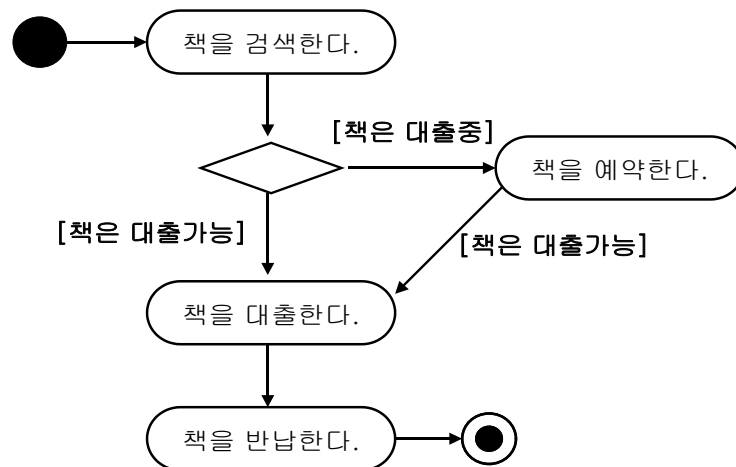
UML에서는, “[조건]”과 같이 기술하는 것이 분기조건을 표현하는 일반적인 방법이다. 이것을 “Guard”라고 부른다. “조건” 부분은 지금 단계에서는 자연언어로 작성해 두도록 하자.

실제로는, Java 등과 같은 프로그래밍언어 및 유사코드로 작성해도 좋다. 또한, UML에서는 OCL(Object Constraint Language, 객체제약언어)이라는 특별한 언어도 정의되어 있다. OCL에 대해서는 후술하기로 한다.



<그림 2-9> 분기

우선, 현 단계에서 Workflow에 예약하는 경우를 추가해 보면 <그림2-10>과 같다.



<그림 2-10> “예약한다.”를 추가한 Workflow

### [내장된 형태의 Activity]

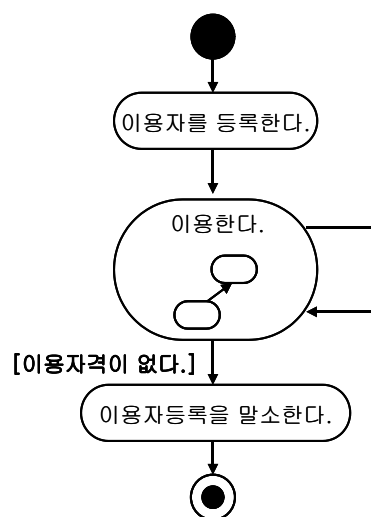
대출부분에 관하여, 위와 같이 표현해도 좋지만, 책을 대출하기 위해서는 우선 대출하려는 사람이 도서관이용자로서 도서관에 등록되어 있어야 한다. 이것은 대출과는 별개의 흐름이므로, 별개의 Workflow에 작성한다.

즉, 이용자는,

1. 이용자등록을 수행한다.
2. 이용한다(반복)
3. 이용자격이 상실되면 등록을 말소한다.

와 같은 Workflow를 수행한다. 이와 같은 3가지 workflow들 중에서 “이용한다.” 부분을 확대하면, <그림2-11>이 된다. Activity Diagram에서는, Activity가 내장된 형태로 표현될 수 있다.

그러면, 실제로 도면에 그려보기로 하자.



<그림 2-11> “이용자를 등록한다.”에 관한 Workflow

“이용한다.”Activity에 이상한 마크가 붙어있으나, 이것은 중첩되어 있음을 나타낸다.

### [SwimLane]

지금까지 기술해 온 Workflow는 1개의 Actor 즉, 도서관 이용자인만 다루어졌다. 그러나 실제로는 “이용자로서 등록한다.”와 같은 작업은,

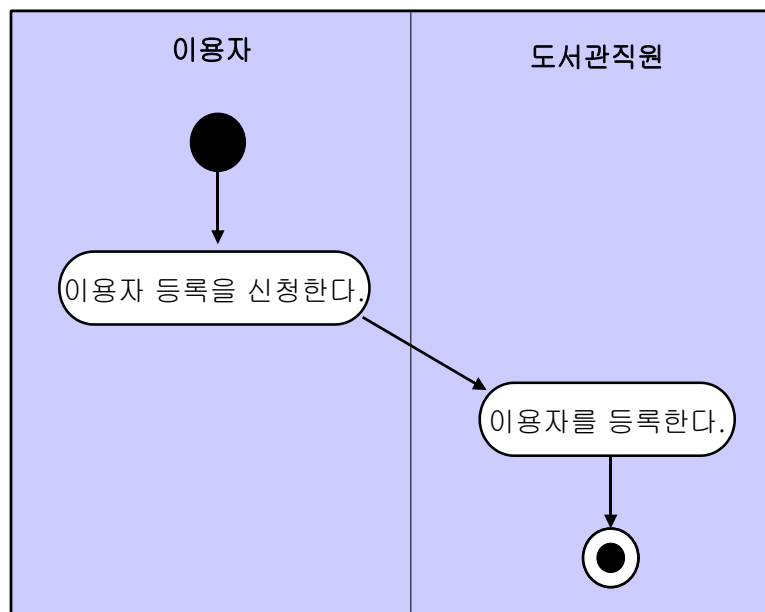
1. 이용하려는 사람이 이용자등록을 신청한다.
2. 도서관직원이 신청을 받아서 실제로 등록한다.

와 같은 흐름으로 진행될 것이다. 따라서 “이용자로서 등록한다.”Workflow에는 2개의 Actor, 즉, 이용자와 도서관직원이 관련되어 있다.

Activity Diagram에서는 구분선을 세로로 그어서 화면을 분할하고, 분할된 각 구역에 1개의 Actor를 대응시킨다. 이와 같이 분할된 구역을 경기용 수영 풀에 비유하여 SwimLane 또는 Lane이라고 부른다.

그러면, “이용자로서 등록한다.”와 같은 Workflow에서는, 관계있는 Actor가 이용자Actor와 도서관직원Actor이므로, 세로로 선을 그어서 2개의 SwimLane으로 분할한다.

“이용자로서 등록한다.”Workflow는, 다음과 같이 된다.



<그림 2-12> SwimLane을 사용하여 이용자와 도서관직원을 표현한 Activity Diagram

SwimLane을 횡단하는 Activity들 사이의 천이(업무의 흐름)에 주목해 보자. 각각의 SwimLane은 이용자Actor와 도서관직원Actor에 대응하고 있다. 2개의 Actor들 사이에

Activity의 천이(<그림2-12>에서 “이용자등록을 신청한다.”Activity로부터 “이용자를 등록한다.”Activity로의 천이)가 존재한다는 것으로부터, 대응하는 Actor들 사이에 관련관계가 있음을 알 수 있다.

이용자Actor와 도서관직원Actor 사이의 관련관계는, “도서관직원이 이용자에게 서비스를 제공한다.”에 해당한다. 즉, “이용자를 등록한다.”Workflow는, 이와 같은 관련관계를 구현한 것(서비스의 일환)이 된다.

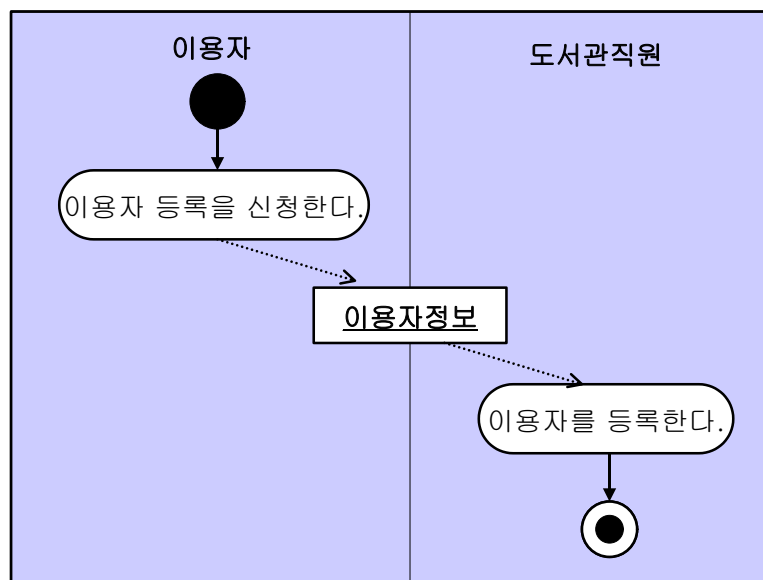
일반적으로, 상이한 SwimLane들 사이의 Activity천이는, 각 SwimLane에 대응하는 Actor들 사이의 관련관계에 대응한다. 따라서, Workflow로부터 Actor들 사이의 관련관계를 검증하는 것이 가능하게 된다.

### [Object Flow]

<그림2-12>에서는, 단 2개의 Activity만 존재하는 매우 단순한 Workflow를 나타내었다. 그러나 이와 같은 2개의 Activity에 대해서 조금 더 생각해 보면, 2개의 Activity들 사이에 “정보”가 전달되고 있음을 알 수 있다. 즉, 이용자의 정보(예를 들면, 성명, 연락처 등)를, 등록업무를 담당하고 있는 도서관직원에게 전달하지 않고서는 이용자 등록을 할 수 없다.

Activity Diagram에서는, Activity들 사이에 전달되는 정보에 대해서도 표현할 수 있다. 단, 너무 상세한 정보까지 고려해서는 안 된다. “이용자정보”는, 도서관관리시스템에서 상당히 중요하므로 살펴보기로 한다.

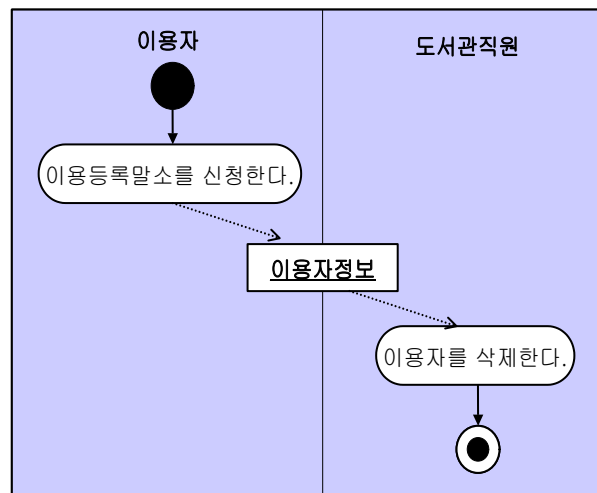
이와 같은 “정보”를 “Object Flow”라고 부르며, 직사각형으로 나타낸다. 어떠한 정보인가는, 직사각형 속에 간결하게 표시한다. Activity로부터 Object Flow로, 다시 Object Flow에서 Activity로 향하는 정보의 흐름을 점선화살표로 나타낸다(<그림2-13>).



<그림 2-13> Object Flow “이용자정보”

<문제> “이용자등록을 말소한다.”Workflow도 <그림2-13>과 같은 방법으로 작성하시오.

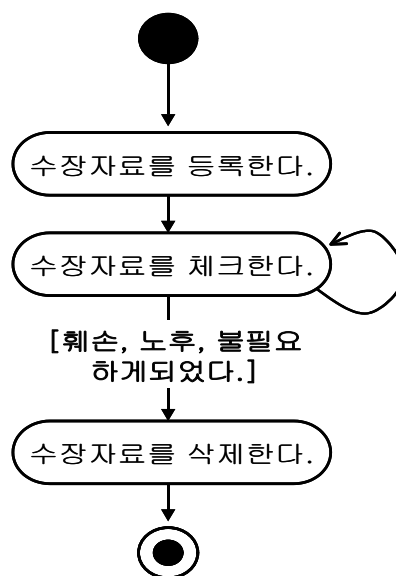
<해답>



<그림 2-14> “이용자등록을 말소한다”  
Workflow

<문제> 수장자료의 검색 및 대출을 하기 위해서는, 수장자료가 확실히 등록되어 있지 않으면 안 된다. 수장자료의 등록에 관한 Workflow를 그리시오. 도서관에서는 주기적으로 수장 자료를 체크하여, 훼손/노후/불필요하게 된 수장자료는 폐기처분하고 등록을 말소하고 있다.

<해답>



<그림 2-15> “수장자료를 등록한다.”에 관한 Workflow

이것으로서, “도서관관리시스템”에 대한 최저한의 Workflow가 완성되었다.

지금까지 생각했던 Actor와 Workflow의 모델링을, “Domain Modeling”<sup>15)</sup> 또는 “Business Modeling”이라고도 부른다. 또한, 대규모의 복잡한 시스템을 구축하는 경우, 시스템 자체를 모델링하기 전에 시스템이 사용될 현장에 대한 모델링이 중요하다.

## 2.6 조작 및 동작의 완전성

### [Walk Through]

앞에서 작성한 Workflow는 정말로 충분한 것일까? 아마도 올바르게 작성되었다고 생각할 것이다. 그러면, 실제로는, 이와 같은 산출물들을 어떻게 확인하면 될까?

조작 및 동작이 충분한가 여부를 확인하는 방법들이 몇 가지 있다. 여기서는 여러 가지 방법들 중에서 2가지만을 간단히 소개하기로 한다.

첫 번째 방법은, “Walk Through”라고 부르는 방법이다.

Walk Through는, 여러 사람들이 모인 자리에서 Role Play와 같은 작업을 수행한다. 우선, 각각의 역할을 결정한다. 앞서 다루었던 도서관관리시스템의 Workflow에서는, 도서관직원 역할을 하는 사람, 이용자 역할을 하는 사람이 필요하다. 각 역할을 담당하는 사람에게는 각 역할의 명칭을 적은 명찰을 붙이게 한다. 만약, 모인 사람들의 숫자가 역할의 수보다 적은 경우에는, 한사람이 여러 개의 역할을 겸하게 한다.

이와 같이 하여, Workflow에 표현되어 있는 대로, 각각 행동의 흐름을 따라서 수행해 보면, 정말로 잘 수행되어지는가 여부를 확인할 수 있게 된다.

이와 같은 작업은, 확실히 재미는 있지만, 확장성<sup>16)</sup>은 없다. 즉, 세밀한 부분까지 모두 시행해 볼 수는 없고(일반적으로 중요하다고 생각되는 부분만 시행함), 규모가 큰 Workflow의 경우 검증이 어렵다는 것이 사실이다.

### [CRUD속성]

또 다른 방법으로서, “CRUD속성”이라고 부르는, 사물을 토대로 조작 및 동작이 만족되어 있는가 여부를 검증하는 방법이 있다.

CRUD속성이란, Create(생성), Retrieve(참조), Update(갱신), Delete(삭제)를 뜻하는 것으로서, 대부분의 정보는 이와 같은 조작이 불가능하면 안 된다. 물론, “책을 대출 한다”라는 동작은, 그대로는 CRUD속성으로 설명할 수 없으며, 개중에는 갱신이 불가능한 종류의 데이터도 있다.

---

15) 지금부터 구축할 시스템 자체에 대해서가 아니라, 해당 시스템을 사용하는 현장에서 어떤 일이 일어나는가? 현장의 구조는 어떻게 되어 있는가? 등을 상세하게 모델링하는 것.

16) 시스템의 확장성. 이용자 및 부하의 증가에 유연하게 시스템을 대응시킬 수 있는 능력. 같은 소프트웨어로 소규모 시스템에서 대규모시스템에 이르기까지 대응할 수 있는 능력을 가리키는 경우도 있다.

앞에서 살펴본 Workflow에는, CRUD속성을 만족하고 있는가? 예를 들면 수장자료에 관해서는 표2-2와 같다.

속성	Activity	비고
Create	수장자료를 등록한다.	-
Retrieve	수장자료를 검색한다.	-
Update	없음.	수장자료정보는 한번 등록되면 변경시키지 않을 것이다.
Delete	수장자료를 삭제한다.	-

<표 2-2> 수장자료의 CRUD속성

이용자에 관해서는, 표2-3과 같다.

속성	Activity	비고
Create	이용자를 등록한다.	-
Retrieve	없음.	서적 대출에 의해 묵시적으로 사용되며, 그 이외의 경우에는 개인정보보호를 위해 참조할 수 없다.
Update	???	-
Delete	이용자등록을 말소한다.	-

<표 2-3> 이용자의 CRUD속성

이와 같이 검증해 보면, “이용자정보를 갱신한다.”라는 Activity를 간과했던 것 같다.

왜 간과했을까? “이용자를 등록한다.”, “이용자등록을 말소한다.”와 같은, 이용자에 관한 2가지 Workflow를 맨 처음부터 나누어 생각했던 것이 실패 원인일지도 모르겠다. “수장자료를 관리한다.”Workflow와 같이, 이용자에 관해서도 Lifecycle(처음부터 끝까지)을 총괄하는 Workflow를 체크해야만 한다.

## 2.7 Use Case

그 다음으로 고려해야 할 사항으로서, Use Case View의 설명에서 거론했던 2가지 관점 중에 하나인, “사용자는 이 시스템을 어떻게 사용하는가?”이다. 이와 같은 관점을, UML에서는 “Use Case”라는 것으로 표현한다. 도면에 그릴 경우에는 타원을 사용한다. “Use Case”는, 개발하려는 시스템이 Actor에게 제공해야 하는 “사용자기능”을 나타낸다.



<그림 2-16> Use Case

Use Case는, 사용자에게 있어서 시스템이 어떻게 느껴지는가를 기능적 측면에서 표현한 것이다. “사용자기능” 대신에 “서비스” 또는 “트랜잭션” 등과 같은 용어를 사용하는 것이 보다 적절한 경우도 있으나, 모두 같은 의미를 갖는다.

Use Case에 있어서 중요한 점은, 다음과 같은 2가지를 손꼽을 수 있다.

- 반드시 Actor와 관계를 갖고 있어야 한다.
- Actor에 있어서 의미 있는(Actor에 어떠한 가치를 부여하는) 기능이어야 한다.

위의 2가지 특징을 만족하지 않게 되면 사용자의 시점이라고 할 수 없다.

예를 들면, 은행의 ATM에서 “현금을 출금한다.”라는 기능은, 사용자에게 있어서 의미가 있으므로 Use Case가 될 수 있다. 그러나 “화면의 현금인출 버튼을 체크한다.”라는 기능은, 사용자에게 어떠한 가치도 줄 수 없으며, 또한, “지폐를 쉐다.”라는 기능은, 사용자와 직접관계가 없는 기능이므로, 모두 Use Case가 되지 않는다.

Use Case의 이름은, 대부분의 경우, “XXX가 YYY를 ZZZ한다.”와 같은 형식을 취한다. Actor가 주어(XXX)가 되는 경우가 많으며, 목적어(YYY)가 되는 경우도 있다. Actor가 주어임에 틀림없는 경우에는 XXX는 생략해도 좋다.

<문제> “도서관관리시스템”의 Use Case는 무엇인가? 목록을 작성해 보시오.

<해답>

이용자 Actor가,

- 수장자료를 대출한다.
- 수장자료를 반납한다.
- 수장자료를 예약한다.
- 수장자료를 검색한다.

도서관직원 Actor가,

- 이용자를 등록한다.
- 이용자를 삭제한다.
- 이용자정보를 갱신한다.
- 수장자료를 등록한다.
- 수장자료를 삭제한다.

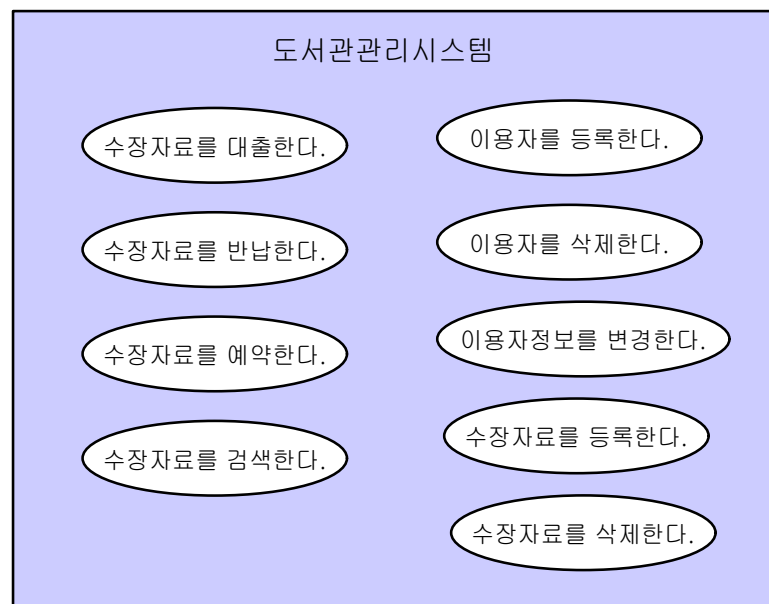
표 형태로 하면 다음과 같다.



Use Case #	이름	Main Actor
1	수장자료를 대출한다.	이용자
2	수장자료를 반납한다.	이용자
3	수장자료를 예약한다.	이용자
4	수장자료를 검색한다.	이용자
5	이용자를 등록한다.	도서관직원
6	이용자를 삭제한다.	도서관직원
7	이용자정보를 갱신한다.	도서관직원
8	수장자료를 등록한다.	도서관직원
9	수장자료를 삭제한다.	도서관직원

<표 2-4> “도서관관리시스템”의 Use Case  
위의 표에서 “Main Actor”로 되어 있는 곳은 “주어가 되는 Actor”로 해석해도 좋다.

#### [Use Case를 찾아내는 방법]



<그림 2-17> “도서관관리시스템”의 Use Case

어떻게 해서 표2-4의 Use Case들을 찾아낼 수 있었을까? 물론, 필자가 문제를 작성하였으므로 당연히 손쉽게 해답을 알고 있었을지도 모른다.

그러나, 필자가 위와 같이 찾아낸 Use Case들은, 실제로 도서관에서 일어나고 있는 사항들을 관찰하거나, 도서관 직원들로부터 이야기를 듣거나, 상상력을 발휘하거나, 또는 고안해낸 것들이다. 이 부분은, 일반적으로 시스템개발에서 요구사항을 찾아내는 경우와 다른 점이 없다.

단, Use Case를 생각할 때 주의해야 할 점으로서,

- 가능한 한 단순하게 생각할 것
- 본질에 초점을 맞출 것

등이 있다. 요구사항을 생각할 때, 이것저것 고려해 보기 시작하면 끝이 없다. 맨 처음에는 본질(예를 들면, 책을 대출하는 것)을 가능한 한 잘 표현하는 것에 집중하고, 기타 사항(“이용자의 검색은 불가능해도 괜찮은가?”)을 고려한다거나, 문제를 복잡하게 만들지 않도록 한다(예를 들면, “예약한 사람이 타 지역으로 이사 가면 어떻게 하는가?”).

그리고, 실제로 도서 대출에 있어서, 시스템에 입력하는 사람은 도서관직원Actor일지도 모르며, “도서를 대출한다.”라는 가치를 얻는 것은 이용자 Actor이므로, 여기서는 이용자 Actor의 Use Case로 정하기로 한다.

### [Activity와 Use Case]

여기서 잠깐 재미있는 사실을 발견할 수 있다. Workflow에서 나타났던 Activity들과 <그림2-17>의 Use Case가 매우 유사하다는 사실을 알 수 있다. 이것은 우연의 일치일까?

Use Case는 시스템과 Actor들의 상호수수작용(동작)을 나타내고 있다. Workflow는, Actor들이 어떤 목적을 달성하기 위한 일련의 행동을 나타내고 있다. 이는 Use Case와 Activity가 중복될 수도 있다는 의미를 갖는다. 오히려 Activity들 중에서 몇 개는 자동화되어 “도서관관리시스템”이 제공하는 Use Case가 된다고 생각해도 좋다.

이와 같이 되면, Use Case는 각각 어떤 Workflow에 출현하지 않으면 오히려 이상하게 된다. 만약, 어떤 Workflow에도 출현하지 않는 Use Case는 Actor에게 아무런 가치도 제공하지 않게 되므로 곤란해진다.

다른 Use Case에 대해서도 생각해 보도록 하자.

<문제> 당신이 생각하고 있는 Use Case에 대하여, 대응하는 Activity가 있는지 여부를 확인해 보시오.

<해답>

필자가 생각해 낸 Use Case 및 Activity는 다음과 같다.

WorkFlow	Activity	Use Case	Main Actor
전체	이용자를 등록한다.	-	-
전체	이용한다.	-	-
전체	이용자등록을 말소한다.	-	-
수장자료를 이용한다.	책을 검색한다.	수장자료를 검색한다.	이용자
수장자료를 이용한다.	책을 예약한다.	수장자료를 예약한다.	이용자
수장자료를 이용한다.	책을 대출한다.	수장자료를 대출한다.	이용자
수장자료를 이용한다.	책을 반납한다.	수장자료를 반납한다.	이용자

이용자를 등록한다.	이용자등록을 신청한다.	-	-
이용자를 등록한다.	이용자를 등록한다.	이용자를 등록한다.	도서관직원
이용자등록을 말소한다.	이용자등록말소를 신청한다.	-	-
이용자등록을 말소한다.	이용자를 삭제한다.	이용자를 삭제한다.	도서관직원
수장자료를 관리한다.	수장자료를 등록한다.	수장자료를 등록한다.	도서관직원
수장자료를 관리한다.	수장자료를 체크한다.	-	-
수장자료를 관리한다.	수장자료를 삭제한다.	수장자료를 삭제한다.	도서관직원
(CRUD속성에 의하여)	사용자정보를 갱신한다.	사용자정보를 갱신한다.	도서관직원

<표 2-5> Use Case와 Activity

<문제> Activity중에는 Use Case가 될 수 없는 것들도 있다. 그 이유는 무엇일까?

<해답>

Use Case는 시스템의 기능을 나타내고 있다. 따라서 Use Case가 될 수 없는 Activity는 시스템화 되지 않는 기능을 나타내고 있다고 생각할 수 있다.

또 한 가지 이유로는, “상세한 Activity를 정리한 Activity”도 대응하는 Use Case를 갖지 못한다. 이에 대해서는 나중에 설명하기로 한다.

#### [Use Case View와 WorkFlow의 대응]

지금까지 살펴본 바에 의하면 WorkFlow를 생각해 봄으로써, Use Case의 후보가 되는 것들을 조사할 수 있었다. WorkFlow를 그리는 이유 중에 하나는 Use Case를 추출하기 위해서 이다.

다시 한 번 Use Case View와 WorkFlow의 대응을 정리해 보자. WorkFlow를 그릴 때에 살펴보았던 Actor와의 대응관계도 살펴보기로 하자.

WorkFlow	Use Case View
SwimLane	Actor
Activity(중의 일부)	Use Case
Activity들 사이의 천이	Actor들 사이의 관련
Object Flow	?

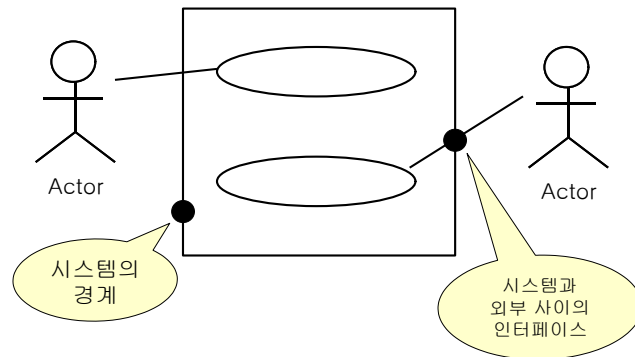
<표 2-6> Use Case View와 WorkFlow의 대응관계

WorkFlow로 표현되는 Object Flow가, Use Case View에서는 무엇에 대응하는지는 나중에 설명하기로 하겠다.

#### [Actor와 Use Case]

현재 개발하려는 시스템을 나타내는 “사각형”의 바깥쪽부분에는 Actor들이 있고, “사각형” 내부에는 Use Case들이 포함되어 있다. 시스템을 나타내는 사각형은, 어느 곳까지가

시스템으로 구현되어야 하는 부분이고, 어디까지가 시스템의 외부인지를 나타내는 “경계”가 되고 있다.



<그림 2-18> 시스템의 경계

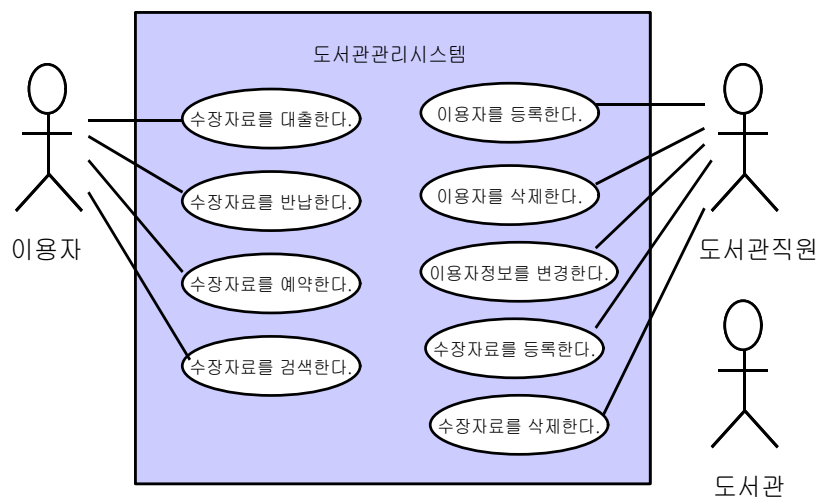
반대로 설명하자면, 현재 개발하려는 “도서관관리시스템”은 9개의 Use Case들의 집합이라고 생각할 수 있다.

Use Case에는 해당 Use Case의 주어 및 목적어가 되는 Actor가 반드시 존재한다. Use Case와 이에 관련된 Actor들 사이에 선을 그어보자. 이러한 선분을 UML에서는 “관련”이라고 부른다. 특히, Actor와 Use Case의 관련을 “통신관련”이라고도 부른다. Actor와 시스템 사이에 어떤 형태의 통신(상호작용)이 있음을 나타내고 있다. 이러한 선분과 시스템의 경계선과의 교차점이 시스템 외부인터페이스에 해당한다.

<문제> 지금까지 그렸던 Actor와 Use Case들 사이에 관련을 나타내는 선분을 그려보시오.

<해답>

Actor와 Use Case들 사이에 관련을 나타내는 선분을 그려보면 다음과 같다.



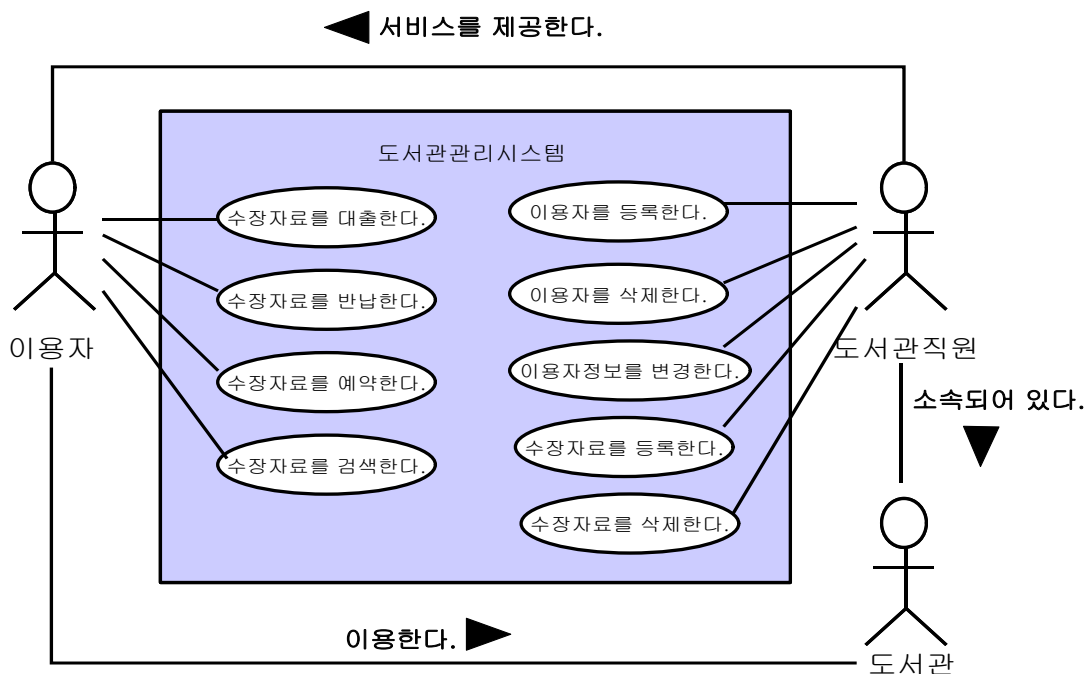
<그림 2-19> Actor와 Use Case들 사이에 관련을 나타내는 선분을 그린다.

위 그림에서는, 우연하게도 모든 Use Case들이 한 개의 Actor와 관련되어 있으나, 일반

적으로 Actor와 Use Case의 대응은 다대다 관계이다.

여러분들이 작성한 도면에는 선분이 그려져 있지 않은 Actor 및 Use Case가 존재하는가? Use Case와 관련이 없는 Actor는 시스템과 아무런 관계가 없는 존재가 된다. 또한, Actor와 관련이 없는 Use Case는, 아무도 사용하지 않는 기능이 되어 버린다. 따라서 독립되어 있는 Actor와 Use Case는 존재할 수 없다.

그렇다면, “도서관”이라는 Actor에 직접 연결된 Use Case가 존재하지 않으므로, 잘못된 도면일까? 아니다. 다만, 도서관과 도서관직원 사이에는 관련이 있을 것이므로, 간접적으로 연결되게 된다.



<그림 2-20> Actor와 Use Case들 사이, 그리고, Actor들 사이에 관련을 추가한 도면

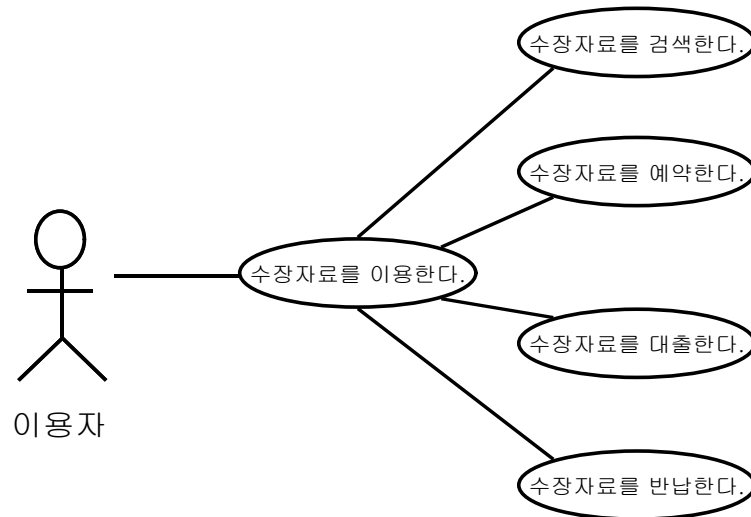
## [Use Case들 사이의 관계]

Actor와 Use Case 사이에 “관련관계”가 존재하듯이, Use Case와 Use Case사이에도 여러 가지 관계가 존재한다. UML에서는, 확장(extend), 일반화(generalization), 포함(include) 등과 같은 3가지 종류의 관계가 정의되어 있으나, 실제로 이와 같은 3가지 관계를 구별하지 않고 사용하는 경우도 있다<sup>17)</sup>.

여기서는 “관련관계”만을 사용하도록 한다. 여러 개의 Use Case들을 모아서 만들어진 Use Case라든가, 어떤 Use Case를 여러 개의 Use Case로 분해한 경우에는 해당 Use Case들 사이에 실선을 그어서 “관련”되어 있음을 표현한다.

17) 바꾸어 설명하면, 대부분의 모델링에서는 이와 같이 난해한 사용방법에 대하여 구별하여 사용하지 않더라도 적절히 모델링할 수 있다. 모델링에 필요한 것은, 이해와 센스, 그리고 적절한 선에서의 타협이다.

예를 들어, <표2-5>의 WorkFlow에는 “책을 검색한다.”, “책을 예약한다.”, “책을 대출한다.”, “책을 반납한다.”와 같은 Activity들(이것들은 그대로 Use Case에 대응한다)을, “수장 자료를 이용한다.”에 모아서 정리할 수 있다.



<그림 2-21> 정리된 Use Case

따라서, Use Case를 이와 같이 정리할 수도 있다. 단, 이와 같은 정리기능을 과용하지 않도록 주의하기 바란다. 특히, Actor와 Use Case 사이가 4단계 또는 5단계가 되어버리면 의외로 복잡해져서 오히려 이해하기 어렵게 되는 경우도 있다.

## 2.8 정적 모델과 동적 모델

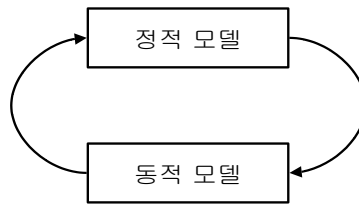
지금까지 작성했던 모델들 중에서 Use Case Diagram은 시간의 개념이 포함되지 않는 모델<sup>18)</sup>이며, Activity Diagram은 시간의 개념이 포함된 모델이었다. 시간의 개념이 포함되지 않은 정적인 구조만을 나타내는 모델을 “정적 모델”이라고 부른다. 한편, 시간의 개념이 포함된 동적인 움직임을 나타내는 모델을 “동적 모델”이라고 부른다.

UML을 사용한 모델링의 특징들 중에 한가지로서, 정적인 모델링과 동적인 모델링을 상호 반복하면서 모델링을 진행해 가는 특징이 있다.

동적인 모델링이란, 정적모델을 머릿속에서 또는 지면위에서 움직여보거나 시뮬레이션해보는 것이라고 할 수 있다<sup>19)</sup>. 이렇게 해 봄으로써, 정적인 모델링에서 간과해버린 사항들을 동적인 모델링에서 발견해 내거나, 반대로 동적인 모델링에서는 발견해 낼 수 없었던 사항들을 정적인 모델링에서 찾아낼 수 있게 된다.

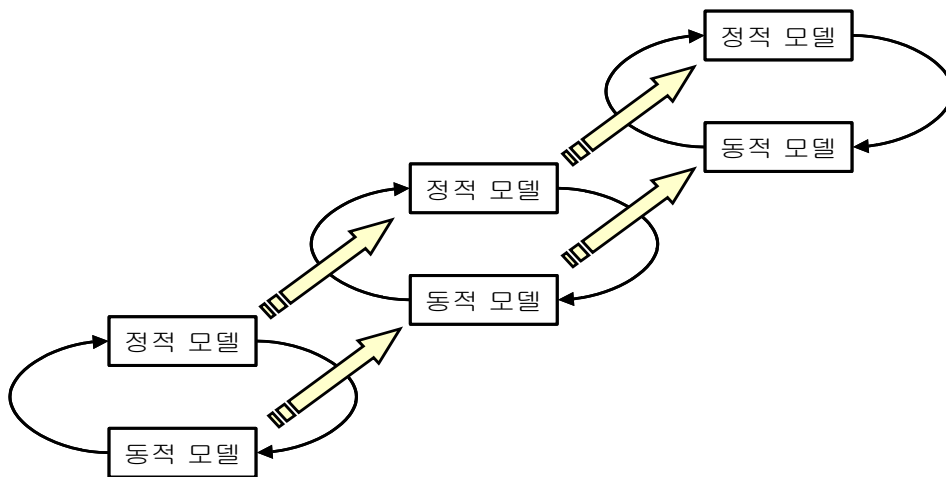
18) 보다 정확하게 말하면, “Use Case”는 구조가 아닌 동작을 나타내고 있으므로 UML에서는, 정적인 모델에 포함되지 않는다. Use Case가 동작을 나타내고 있음은 확실하지만, Use Case에는 “시간”의 개념이 일절 포함되어 있지 않으므로 FUM에서는 정적모델로 취급한다.

19) 여기서는, 머릿속 또는 지면위에서만 시뮬레이션 할 수 밖에 없으나, 모델을 실제로 “움직여 보는” 기능을 갖춘 도구(Tool)들이 다수 존재한다.



<그림 2-22> 정적모델과 동적모델

이와 같이 2종류의 상이한 모델링을 상호 반복함으로써, 상호간의 정당성을 검증할 수 있게 된다. 또한, 정적/동적 모델은 상호 보완적인 관계가 있다.



<그림 2-23> 정적모델과 동적모델의 관계

현재의 작업에서는, 맨 처음에 Actor를 찾아내었다. 이것은 정적인 모델링에 해당한다. 그 다음에는 Actor가 어떻게 동작하는지, 즉, 책을 대출하거나 반납하는 행동이 실제로는 어떻게 이루어지는지를 WorkFlow에 의해 Activity Diagram을 사용하여 살펴보았다. 이것이 동적인 모델링에 해당한다. 그 후에 WorkFlow의 Activity를 토대로 하여, Use Case를 찾아내었다. 이것은 정적인 모델링에 해당한다.

이와 같이, 정적인 모델링과 동적인 모델링을 반복하면서 모델링을 진행하게 된다.

## 2.9 Use Case 시나리오

앞서 살펴보았던 WorkFlow는, 도서관관리시스템을 구축하고 있는 현장(도서관)에서 무슨 일이 일어나고 있는지를 명확하게 하기 위한 메모에 해당한다. WorkFlow에 등장하는 등장 인물이 바로 Actor들이다.

Actor들이 정보(Object Flow)를 상호 수수하면서 어떤 액션을 일으켜서 업무를 수행하고 있다. 현재 개발하려는 시스템은 이와 같은 상호수수작용과 어떻게 관계하고 있는 것일까? 즉, Actor에 시스템이 부가될 경우의 동작을 어떻게 모델링하면 좋을까?

한 가지 방법으로서, 앞에서 살펴보았던 WorkFlow에 “시스템”이라는 SwimLane을 추가해 보는 것이다. 이렇게 함으로써, Activity Diagram을 사용하여 표현할 수 있게 된다.

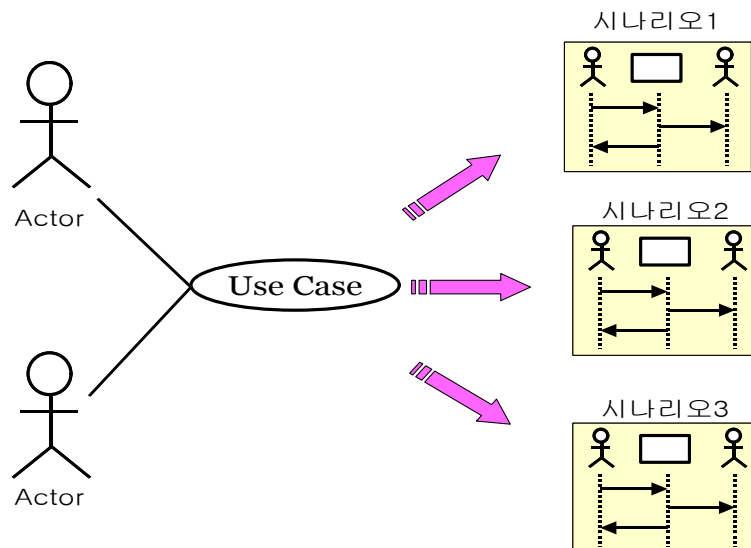
또 다른 방법으로서, “Use Case Scenario”를 이용하는 방법이 있다. 일반적으로 모델링에서는, Use Case Scenario 방법을 사용한다.

Use Case마다 Actor와 시스템이 어떻게 상호작용을 하는지를 시간의 흐름에 따라서 표현한다. 영화 또는 드라마의 시나리오와 매우 유사하므로, “Use Case의 시나리오”라고도 부른다(“스토리”라고도 부르기도 한다.).

Use Case Scenario는, UML에서 “Sequence Diagram”을 사용하여 작성되지만, 자연언어를 사용하여 일반적인 시나리오처럼 작성하는 경우도 있다. Graphical User Interface를 사용하는 시스템의 경우, 화면 이미지들을 나열하고 해당 설명을 옆에 기입하는 경우도 있다. 즉, 일정한 형식 없이 작성하면 된다. 여기서는 Sequence Diagram을 사용하여 표현하기로 한다.

Use Case시나리오에 등장하는 요소들은, Use Case와의 관련관계를 나타내는 선분에 의해 연결된 Actor들과 시스템이다. 그리고 일반적인 경우, 한 개의 Use Case에 대하여 여러 개의 시나리오를 표현한다. “일반적으로 처리되는 경우”와 “에러가 발생하는 경우”, “특별한 경우” 등이 있다. 이것들 중에서 가장 일반적인 시나리오를 “Main Scenario”, 그 밖의 것을 “Sub Scenario”라고 부르는 경우도 있다.

Use Case는 시나리오들의 집합이라고도 할 수 있다.



<그림 2-24> Use Case와 Scenario

## 2.10 Sequence Diagram

### [작성방법]



UML에서는 Sequence Diagram을 사용하여 Use Case시나리오를 표현하기도 한다. Use Case시나리오를 작성하기 전에, Sequence Diagram이란 무엇인지에 대하여 설명하기로 한다.

Sequence Diagram은, 객체들<sup>20)</sup> 사이의 상호수수작용을 시간 순으로 표현하는데 적합하다. Activity Diagram과 다른 점은, 자기 자신의 내부 Action이 아니라, 다른 객체와의 사이에서 발생하는 상호수수작용을 중시한다는 점이다. 또한, Activity Diagram이 일반적인 상황을 나타내고 있는 것에 대해서, Sequence Diagram은 특정한 상황에서 특정한 객체가 어떻게 동작해 왔는가(Trace)를 나타내고 있다는 특징을 갖고 있다.

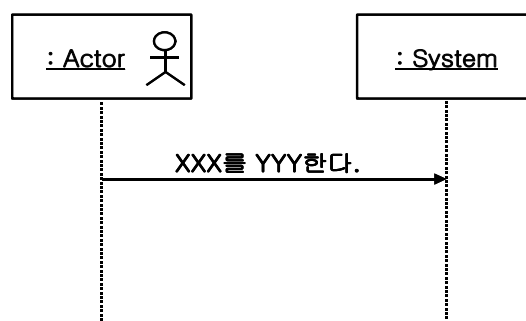
Sequence Diagram에서는, 세로축은 위에서 아래로의 시간경과를 나타낸다. 시간을 나타내는 눈금이 정의되어 있지는 않으므로, 단순하게 아래쪽으로 내려가면서 시간이 경과하고 있다는 사실정도만 나타낸다.<sup>21)</sup>

가로축에는, Use Case시나리오에 참가하는 객체를 나타내는 사각형을 나열한다. 객체의 나열 순서에는 의미가 없으므로, 이해하기 쉬운 순서로 나열하도록 한다. 객체의 이름은 사각형 내부에 밑줄을 그어서 표기한다. 객체이름 표기법에는 몇 가지 표기규칙이 있으나, 여기서는 상세하게 알 필요는 없다.

각각의 사각형으로부터 아래쪽으로 점선을 긋는다. 이와 같은 점선을 “생명선(Life line)”이라고 부른다. 해당 객체의 “일생”을 나타내는 선이라는 의미이다.

어떤 객체A로부터 다른 객체B로의 작용 또는 동작촉발이 있는 경우, A의 생명선에서 B의 생명선으로 화살표선을 긋는다. 이때의 작용 또는 동작촉발을 “메시지”라고 부르며, 작용 또는 동작촉발의 내용을 선분 위에 기입한다.

객체에서 상호수수작용이라고 하면, 기본적으로는 “메시지 교환”을 의미하지만, Actor 및 시스템에 메시지교환은 존재하지 않는다. 따라서 Actor로부터 시스템으로의 요구 및 입력 등이 Actor 또는 시스템으로의 실선화살표가 되며, 시스템으로부터 Actor로의 응답 또는 출력이 시스템으로부터 Actor로의 실선화살표가 된다.



<그림 2-25> Sequence Diagram의 예

20) 여기서 “객체”란, 클래스가 아닌 인스턴스를 가리킨다(엄밀하게 말해서 “Role”이다.). 객체의 일부분 또는 객체의 어떤 측면, 그리고 이런 시나리오에서 이러한 객체에게 주어진 역할, 등과 같은 의미를 갖는다.

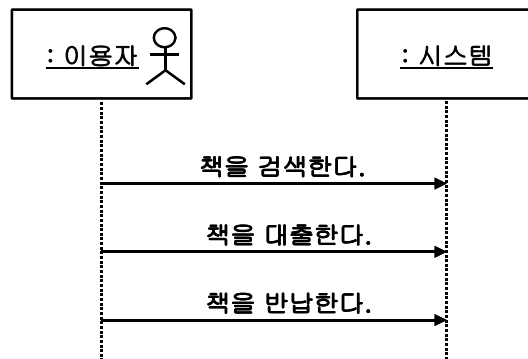
21) 만약, 시간을 Sequence Diagram 내에 표현하고자 한다면, 적당하게 시각을 나타내는 눈금을 첨부하거나, 주석 문을 붙여서 시간을 알 수 있도록 표현한다.

### [책을 이용한다(Use Case 시나리오)]

지금까지 살펴보았던 바와 같이, 도면을 그리기 전에 Use Case를 한개 선택하여, 해당 시나리오를 작성해 보도록 하자. <그림2-21>의 Use Case Diagram에서, 책을 이용하는 부분부터 시작하도록 한다. 우선, Main Scenario, 즉, 가장 주요한 시나리오부터 살펴보도록 하자.

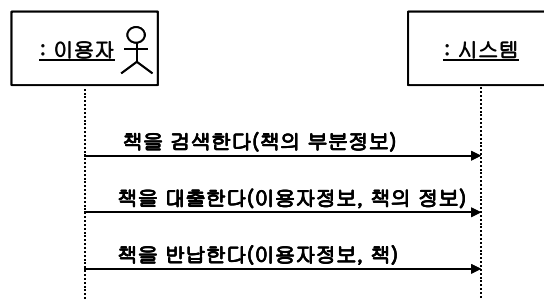
1. (이용자 --> 시스템) 책을 검색한다.
2. (이용자 --> 시스템) 책을 대출한다.
3. (이용자 --> 시스템) 책을 반납한다.

너무 단순하다는 느낌도 있지만, 처음에는 이 정도 수준에서부터 시작하기로 하자. 여기서, “시스템”이란, 현재 구축하려는 “도서관관리시스템”을 의미한다. Sequence Diagram은 다음과 같다.



<그림 2-26> Use Case Scenario(“책을 이용한다.”)

객체들 사이의 상호수수작용에 관하여, 어떠한 정보가 교환되는지를 나타낼 수도 있다. Activity Diagram에서는 Object Flow를 사용하여 나타냈지만, Sequence Diagram에서는 메시지의 매개변수를 사용하여 나타낸다. 예를 들면 다음과 같다.

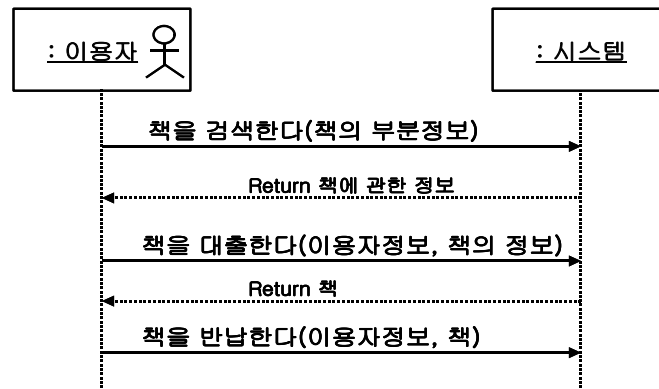


<그림 2-27> 매개변수를 갖춘 Use Case Scenario

메시지와 더불어서 전달되는 정보의 내용에 대해서는 현재시점에서는 상세한 부분까지 살펴보지 않아도 괜찮다. “XXX에 관한 정보”와 같은 정도로 파악해 두는 것으로 충분하다.

사람에 따라서는 어떤 작용에 대한 결과에 의해 반환되는 정보도 시나리오에 기입하는 경우도 있다. 점점 상세한 사항이 파악됨에 따라서 기입해도 좋고, 전혀 기입하지 않는 사람

도 있다. 만약, 기입한다면 점선화살표선을 사용하여 표현하도록 한다. 그러나 불필요하다고 판단될 경우에는 생략해도 좋다.



<그림 2-28> 반환되는 정보를 기입한 Use Case

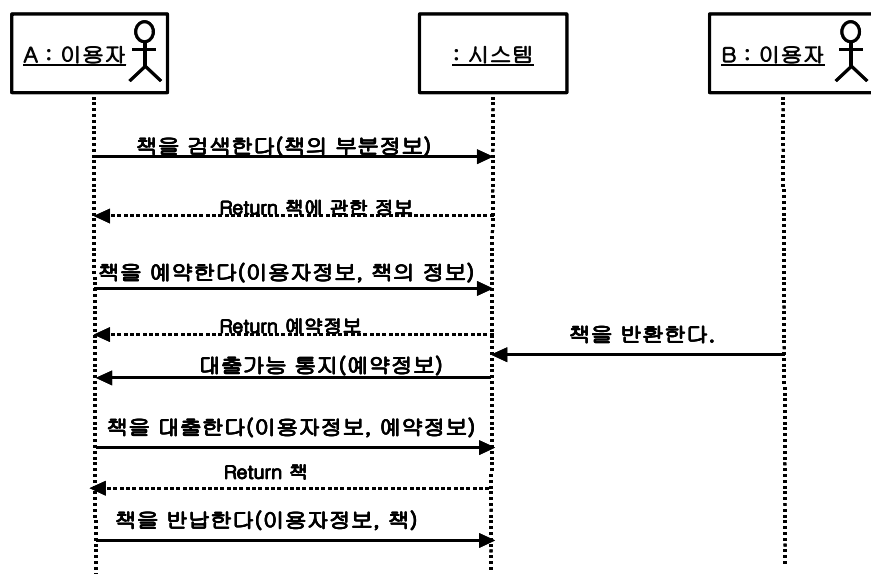
#### Scenario

<문제> 지금까지 작성한 것은 주로 Main Scenario로서, 찾고자 하는 책이 검색되어 대출할 수 있는 경우를 다루었다. 그러면, Sub Scenario로서, 대출하려는 책이 이미 대출중이어서 예약하는 경우에는 어떤 식으로 표현하면 좋을까? <그림2-28>을 참고해서 Sequence Diagram을 작성하시오.

<해답>

Activity Diagram을 사용하여 WorkFlow를 표현할 경우, “분기”를 사용하여 한 장의 도면 속에 여러 가지 경우에 대응하는 WorkFlow를 통합하여 표현할 수 있었다. 그러나, Sequence Diagram을 사용하여 Use Case Scenario를 작성할 경우, 기본적으로는 각 시나리오마다 각각의 도면을 작성한다.

필자는 다음과 같은 Use Case Scenario를 작성해 보았다.



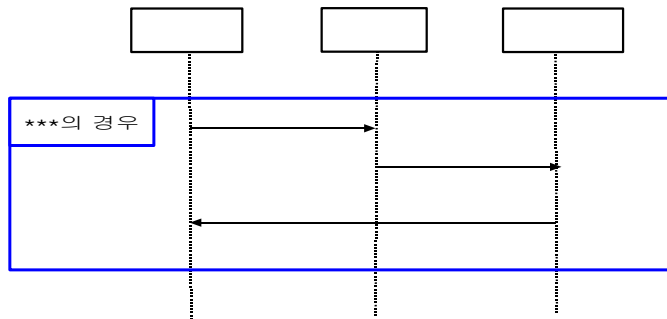
<그림 2-29> 예약하는 경우의 Use Case Scenario

### [도면 작성 시 주의사항]

이와 같이 해서 실제로 작성해 보면, 대략적으로 같아 보이지만 조금씩 서로 다른 Sequence Diagram들을 많이 작성하지 않으면 안 된다. 그러나 어느 정도까지는 어쩔 수 없이 유사한 도면을 작성하지 않을 수 없다.

수작업에 의해 도면을 작성할 경우, 지루하고 귀찮으므로 한 장씩 도면을 분할하는 것이 좋을 것이다. UML 툴을 사용하는 경우에는, Cut & Paste기능을 이용하면 손쉽게 작성할 수 있을 것이다. 화이트보드를 사용하는 경우에는, (디지털카메라 또는 인쇄 등을 이용하여) 중간 중간 기록을 하면서 도면을 작성하도록 한다.

만약, 한 개의 도면에 여러 개의 시나리오들을 기입하고자 한다면 다음과 같이 작성하도록 한다.



<그림 2-30> 여러 개의 시나리오들을 한 개의 도면에 작성하는 경우

실제로 위와 같은 형식은, UML2.0에서 채택된 표기방식이다. 따라서 현재 사용 중인 UML 도구들에서는 이와 같은 형식을 제공하지 않을 수 있지만, 수작업에 의해 작성하는 경우에는 사용해 보는 것도 좋을 것이다.

### [Use Case Scenario의 주의사항]

Use Case Scenario를 작성할 경우, 몇 가지 주의해야 할 사항이 있다.

첫 번째로서, 유저에게 있어서 가치 있는 업무흐름을 맨 처음부터 마지막까지 추적해야 한다는 점이다. 앞의 예에서는, 책을 검색하여 대출한 후에 반환하는 것까지 추적해 보았다.

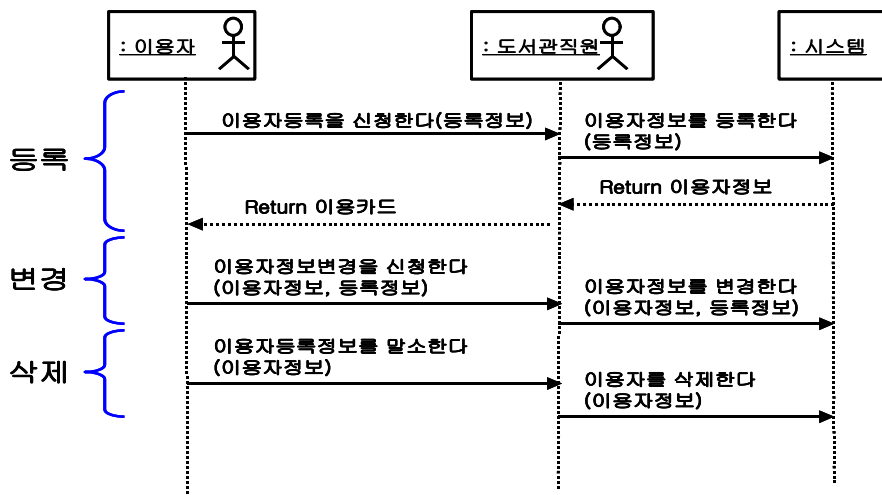
두 번째로서, Use Case Scenario는 유저 및 고객이 살펴보고 이해할 수 있도록 작성되어야 한다는 점이다. 이를 위해서, 시스템 내부에서는 어떻게 구현할지 등과 같은 내부적인 사항은 Use Case Scenario에 포함되어서는 안 된다. 유저 및 고객의 입장/관점에서 작성하는 것이 중요하다. 가능하다면 유저 및 고객과 함께 Review해 보는 것이 좋다.

세 번째 사항으로서, 정보의 흐름을 체크해서 일관성이 있는지, 그리고 모순 또는 단절되는 곳이 없는지를 살펴보아야 한다는 것이다. 이와 같은 조사는, Use Case Scenario를 사용해서 Use Case의 동적인 검증을 수행하는 것에 해당한다.

마지막으로, Use Case Scenario의 복잡도에 따라서 확대 또는 축소해야 한다는 점이다. Use Case Scenario는, 가능한 한 한눈에 파악할 수 있을 정도의 크기로 작성해야 한다. 이를 위해서는, 복잡하게 되어버린 Use Case Scenario에 대해서는 분리 가능한 부분을 추출하여 별개의 Sequence Diagram에 작성한다. 만약 분할하는 것이 적절하다고 판단되면, 이에 맞추어서 Use Case를 분할해도 좋다.

<문제> “이용자를 등록한다.” Use Case의 시나리오를 작성해 보시오. “이용자를 삭제한다.”와 “이용자정보를 변경한다.” Use Case의 시나리오도 같이 작성해 보시오.

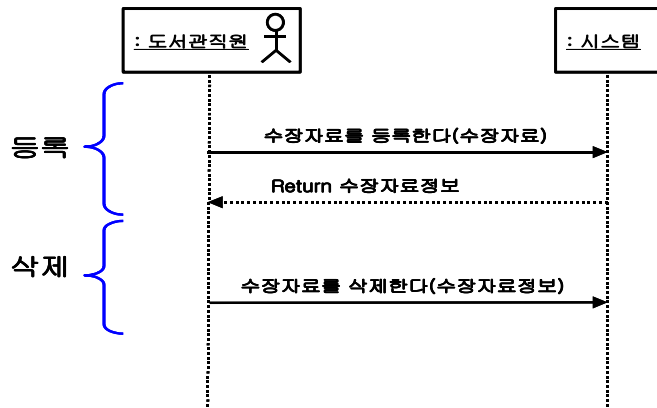
<해답>



<그림 2-31> 이용자의 등록/변경/삭제에 관한 Use Case Scenario를 표현한 Sequence Diagram

<문제> “수장자료를 등록한다.”, “수장자료를 삭제한다.” Use Case의 시나리오를 작성하시오.

<해답>



<그림 2-32> 수장자료의 등록/삭제에 관한 Use Case Scenario를 표현한 Sequence Diagram

## 2.11 요구사항의 추가

지금까지 작성한 Use Case Diagram, Activity Diagram, Sequence Diagram 등을 A시립도서관의 담당자와 함께 Review해 보면, 이전에는 전혀 언급하지 않았던 여러 가지 사항들이 도출된다. 실제로, 반납기한이 지나도 반납되지 않을 경우에는 대응책을 마련해야 하는 등과 같은 도서관이용규칙이 몇 가지 존재할 것이다. 이와 같은 사항들은, 실제 개발에서 빈번하게 발생한다. 특히, 개발을 수행해 온 모델에는, 대출과 관련하여 책의 경우에는 10권 이내, 음악 및 영상물의 경우에는 5개 이내로 대출할 수 있다는 이용규칙을 아직까지 반영시키지 못하고 있음을 상기해 주기 바란다.

따라서, 도서관 담당자에게 다시 한 번 이와 같은 도서관이용규칙을 열거해 보도록 하였다. 어떤 규칙은 실현 불가능할 수도 있으며, 간과해 버린 것도 있을 수 있으므로, 주의해서 살펴보도록 하자.

- ▶ 한사람이 한번만 등록할 수 있다. 즉, 중복해서 등록할 수 없다(그러나, 사용자 등록정보는 신청자 스스로 작성해서 신고하도록 되어 있다.)
- ▶ 등록할 수 있는 사람은, A시 거주자 또는 근무지가 A시로 되어 있는 사람들에게 한정된다.
- ▶ 등록에 필요한 사항으로는, 성명, 주소, 성별, 생년월일, 연락처(전화번호 또는 e-mail 주소), A시의 주민이 아닌 사람의 경우에는, 근무지명과 근무지 주소, 연락처가 추가됨.
- ▶ 등록한 사람에게는 이용자 카드를 발급한다. 분실했을 경우에는 재발급한다.
- ▶ 이용자격이 상실되면 신고하여 변경한다.
- ▶ 대출 가능한 소장 자료는, 책 및 잡지류의 경우 10권 이내, 음악 및 영상물은 5개 이내로 제한되며, 최장 2주일까지 대출가능. 반납기한이 휴관일인 경우에는 익일까지 반납한다.
- ▶ 반납기한을 넘기고도 반납하지 않은 대출물이 있을 경우에는, 반납완료시까지 새로운 대출은 불가능하다.
- ▶ 예약할 수 있는 자료의 개수는 제한 없음. 도서관에 소장되어 있지 않은 서적이라도 예약가능(즉, 구입희망신청 가능).
- ▶ 예약한 자료가 완비되었을 경우, 예약등록순서에 의해 연락처로 연락한다. 연락한 후 2주일이 경과해도 대출하지 않았을 경우에는 다음 예약자에게 연락함.

## 2.12 테스트 시나리오로서의 Use Case Scenario

앞서 작성했던 Use Case Scenario에 대하여, Use Case Scenario를 현 상태대로 진행해 가는 방법과, 상세한 요구사항에 맞추어서, 상세한 Use Case Scenario를 작성한 후에 진행하는 방법을 생각할 수 있다.

A시립도서관의 담당자로부터 앞서 열거했던 도서관이용에 관한 몇 가지 규칙을 도출하였다. 그러나 필자의 예감으로는, 이러한 규칙은 앞으로 얼마든지 추가적으로 도출될 것 같으며, 얼마든지 변경될 수 있다.

“어떠한 규칙이 존재한다.”라고 하는 것은, 시스템에 있어서 본질적인 사항이지만, 각각의 규칙 자체는 실제로 그만큼 본질적이지 않다. 그럼에도 불구하고, 이와 같은 규칙들을 모두 올바르게 Sequence Diagram으로 표현하려고 한다면, 막대한 시간과 비용이 소모될 것이다. 이미 “무엇을 하면 좋을지”, 즉, 주요한 Use Case는 파악되었다. 따라서 Use Case Scenario 작성은, 이정도로 마무리해 두는 것이 좋을 것이다.

한편, 위와 같은 규칙들에 대응되는 Use Case Scenario를 각각 명확하게 작성하는 것도 실제로는 의미가 있다. 왜냐하면, 여기에 작성된 Use Case Scenario를 사용하여 테스트 시나리오(Test Scenario)<sup>22)</sup>로 사용할 수 있다. 불행하게도 앞서 작성했던 Sequence Diagram으로는 그대로 테스트 시나리오로서 사용할 수는 없다.

위에서 열거했던 규칙들에 대응하는 Use Case Scenario를 작성함으로써, 현시점에서 테스트 시나리오를 완성시켜서 고객에게 인수테스트(Acceptance Test)로서 사용하게 할 수 있다. 즉, “도서관관리시스템”의 완성(납품)조건을 현시점에서 결정할 수 있게 되는 것이다.

여러분들은 위의 2가지 방법 중 어느 것을 선택할 것인가?

필자의 생각으로는, 이와 같은 선택에 있어서 아래의 3가지 조건을 고려하여 결정해야 한다고 생각한다.

- 이와 같은 규칙들은, 앞으로 얼마나 변화/추가될 것인가?
- 모델링을 하는 사람들은, 문서로서의 모델을 얼마나 중시하고 있는가?
- 사용하고 있는 UML 도구에서는, Use Case Scenario로부터 자동적으로 테스트 시나리오를 생성가능한가?

불행하게도, 마지막 조건을 만족하고 있는 UML 도구는 현시점에서는 거의 존재하지 않는다. 앞의 2가지 조건은, 프로젝트와 관련된 정책 및 고객의 요구에 따라서 만족/불만족이 결정된다.

## 2.13 Use Case의 사전조건 및 사후조건

### [사전조건과 사후조건]

Use Case를 보다 명확하게 작성해 보도록 한다. Use Case는 Use Case Scenario들의 집합이다. 따라서 Use Case를 명확하게 하기 위해서, Use Case Scenario를 보다 상세하게 작성하는 방법도 있다.

여기서는, Use Case를 명확하게 하는 방법으로서, 조금 다른 기법을 사용하기로 한다. 즉, 각 Use Case마다 “사전조건”과 “사후조건”이라는 2가지 조건을 생각해 보는 방법이다.

사전조건이란, 해당 Use Case가 실행되기 위해 필요한 조건을 나타낸다. 사전조건이 모두 만족되어야지만 Use Case가 실행가능하게 된다.

예를 들면, “수장 자료를 대출한다.”Use Case에 대하여,

1. 대출하려는 사람이 이용자로서 등록되어 있을 것.

22) Test Scenario란, 사용자 테스트(User Test)를 위한 Test항목을 의미한다.

2. 대출하려는 사람에게 반납기한이 초과된 미반납자료가 없을 것.
3. 대출 가능한 수량 이내이어야 함.
4. 수장자료가 대출 가능해야 함.

와 같은 조건들 모두가 만족된다면, 대출을 허가해도 좋은 상태가 된다.

이에 대하여, 사후조건이란 해당 Use Case가 실행된 후에 만족되어야만 하는 조건을 의미한다.

예를 들어, “수장자료를 대출한다.” Use Case가 실행된 후에는,

1. 대출자의 대출완료점수가 증가한다.
2. 수장자료는 대출자에게 대출된 상태가 된다.

와 같이 된다. 그밖에, 위의 Use Case가 실행되었다고 해서 특별하게 변경되는 사항은 없다.

각 Use Case마다 사전조건과 사후조건을 추가하게 되면, 해당 Use Case가 실제로는 무엇을 하는 것인지 명확해진다<sup>23)</sup>. 사전조건과 사후조건을 작성하기 위해서는, 일반적으로 자연언어를 사용해도 좋다. 그러나 숙련된 사람이라면, OCL(Object Constraint Language) 또는 Java언어와 유사한 유사코드를 사용해도 좋다.

단, 주의해야 할 사항으로서, “조건”을 기입하는 것이지 “해야 할 일”을 기입하는 것이 아니라는 점에 충분히 주의해야 한다.

#### [다른 Use Case에 대한 사전조건과 사후조건]

다른 Use Case에 대해서도 사전조건/사후조건을 살펴보도록 하자.

우선, “이용자를 등록한다.” Use Case의 사전조건은,

1. 이용자는 아직 등록되어 있지 않을 것.
2. 이용자는 등록할 자격을 갖추고 있을 것.

이 된다. 위의 조건 중에서, “이용자는 A시 거주자 또는 근무처가 A시”가 아니라, 일부러 “이용자는 등록할 자격을 갖추고 있을 것”이라고 애매모호하게 작성하였다. 이 부분이 포인트이다. 어느 정도까지 상세화 시킬지는 모델링 담당자의 센스에 따라 차이가 나겠지만, 상세한 사항에 대해서는 후반부로 미루는 것이 현시점에서 자유도를 높일 수 있는 방법이 된다.

사후조건은,

1. 이용자는 등록되어 있다.

가 된다.

다음으로는, “예약한다.” Use Case의 경우를 살펴보자.

23) “사전조건/사후조건”이라는 개념은 Use Case에 국한되지 않고, 동작을 명확하게 하기 위하여 매우 좋은 방법이지만, 현재의 UML에서는 그다지 많이 사용되지 않고 있다. Use Case에 사전조건/사후조건을 설정하는 것은 UML에는 없는 독자적인 방법이다.



사전조건은,

1. 대상 자료가 수장되어 있지 않거나 대출중이다.
2. 이용자가 등록되어 있다.

사후조건은,

1. 대상 자료의 예약리스트에 이용자가 추가된다.
2. 이용자는 대상 자료를 예약하였다.

가 된다.

각각의 조건은, “XXX상태” 또는 “XXX이다”가 된다. 여기서, “XXX한다”로 해버리면 너무 앞서가는 것이 된다.

“자료를 검색한다.”Use Case의 사전조건/사후조건은 어떻게 될까?

실제로 검색 또는 참조하는 경우, 사전조건/사후조건은 존재하지 않는 경우가 많다. 검색 및 참조를 수행하더라도 실세계는 아무런 변화가 없다. 검색만 하고 싶은데 부작용이 발생하면 곤란하게 된다. 사후조건이 없는 경우가 일반적이다.

“수장자료를 반납한다.”Use Case의 경우, 사전조건은,

1. 이용자가 해당하는 수장자료를 대출하고 있다.
2. 해당하는 수장자료는 대출중이다.

또한, 사후조건은,

1. 이용자가 해당하는 수장자료를 대출하고 있지 않다.
2. 해당하는 수장자료는 대출중이 아니다.

가 된다. 본래 반납된 수장자료를 누군가가 예약하고 있었다면 예약자에게 연락해야 한다. 그러나 이것은 “해야 할 일”이지 “조건”이 아니므로, 기술할 수 없지만, 작성한다면 사후조건은,

3. 해당하는 수장자료는 예약되어 있지 않다. 또는 해당 자료를 맨 처음 예약한 사람에게 연락하고 있다.

가 될 것이다. 이와 같은 “조건으로서 작성하기 어려운 경우가 있다”는 점이 결점이 될 수 있다.

<문제> “이용자정보를 변경한다.”Use Case와 “이용자를 삭제한다.”Use Case의 사전조건/사후조건을 작성하시오.

<해답>

“이용자정보를 변경한다.”Use Case

▶사전조건:

1. 이용자가 등록되어 있다.
2. 변경하더라도 이용자의 자격을 만족하고 있다.

▶사후조건:

1. 이용자정보가 신청된 정보와 동일하다.

“이용자를 삭제한다.”Use Case

▶사전조건:

1. 이용자가 등록되어 있다.

▶사후조건:

1. 이용자가 등록되어 있지 않다.

<문제> “이용자를 삭제한다.”시점에서, 이용자가 책을 대출하고 있다면 사전조건을 어떻게 하면 될까?

<해답>

만약, “이용자를 삭제한다.”단계에서, 이용자가 책을 대출하고 있는 경우에는, 사전조건에

2. 이용자는 대출하고 있는 자료가 없다.

를 추가하는 것이 좋을 것이다.



## 제3장 개념모델링

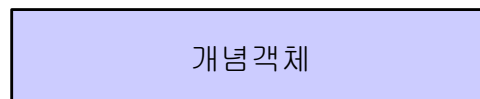
### 3.1 개념객체(Concept Objects)

“개념객체”란, 개발하려는 시스템에서 키가 되는 정보, 기본적인 개념을 지칭한다. “도서관관리시스템”에 있어서 중요한 개념객체의 대부분은, 실제로 지금까지의 단계에서 이미 살펴본 것들이다. 예를 들면, WorkFlow에 출현했던 Object Flow, Use Case의 목적어, Use Case Scenario의 메시지에 부과되었던 매개변수, Use Case의 사전조건/사후조건에 출현했던 목적어 등이다. 물론, 이와 같은 것들이 모두 개념객체가 되는 것은 아니지만, 유력한 후보들임에는 틀림없다.

드디어 여기서 처음으로 “객체”라는 용어가 출현했다<sup>24)</sup>. 객체는, 시스템을 구축할 때 사용하는 레고블럭과 같은 것이다.

잠시 복습해보자. “객체란, 내부데이터와 이를 조작하는 프로그램을 가지고 있으며, 외부로부터 메시지를 수신하여 동작하는 계산단위”이다. 메모리와 CPU를 갖는 소형 계산기라고 생각해도 좋고, “보다 영리한 구조체”라고 생각해도 좋다. 개념객체는, 특히 시스템의 중심이 되는 객체이다.

개념객체를 추출한 다음에는 사각형 내부에 이름을 기입하여 작성하도록 한다<sup>25)</sup>. 우선, “도서관관리시스템”의 기본이 되는 개념객체를 순차적으로 생각해 보도록 하자.



<그림 3-1> 개념객체

### 3.2 “도서관관리시스템”의 개념객체

#### [수장자료]

도서관이라면 “책”이 중심이 된다. 책이 없는 도서관은 상상할 수도 없을 것이다. 그러나, 도서관에는 책 이외에도, “잡지”, “음악 및 영상물” 등이 있으며, 이들 모두가 같은 부류로서 “수장자료”라고 부르기도 한다. 도서관에 수장되어 있는 자료라는 의미이다. 지금부터는, 정식으로 “수장자료”라고 부르기로 한다. 이러한 수장자료에는 여러 가지 종류가 있다.

이와 같은 수장자료의 다양성에 대해서는 도서관으로부터 얻어진 요구사항에도 잘 나타나

24) “객체”라고 표현했지만, 실제로는 객체들의 집합(분류)인 “클래스”가 보다 적합한 표현이다. “개념클래스”라고 부르는 것이 보다 정확할지도 모른다.

25) 원래, 클래스를 나타내는 UML의 아이콘은 3개 영역으로 분할된 사각형이다. 분할된 영역들 중에서 제일 위쪽에 이름을 기입하고, 나머지 2개 영역에는 “속성”과 “조작”을 기입한다.

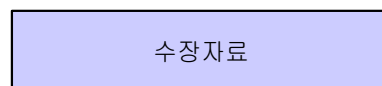
있다. WorkFlow의 Object Flow에도 나타나 있으며, Use Case 및 Use Case Scenario 등에서도 살펴볼 수 있다.

그런데, “수장자료정보”라고 표현되기도 하는데, 어느 쪽 표현이 보다 알맞은 것일까?

객체라는 개념을 중심삼고 사물을 생각하거나 모델링함으로써 얻을 수 있는 여러 가지 이점들 중에서, “실세계의 사물(객체)이 여기에 존재하는 것처럼 여겨서 시뮬레이션하여 살펴볼 수 있다”라는 점이다. 따라서, “수장자료정보”보다는 “수장자료”라고 부르는 것이, 본래의 객체에 보다 근접한 개념이다.

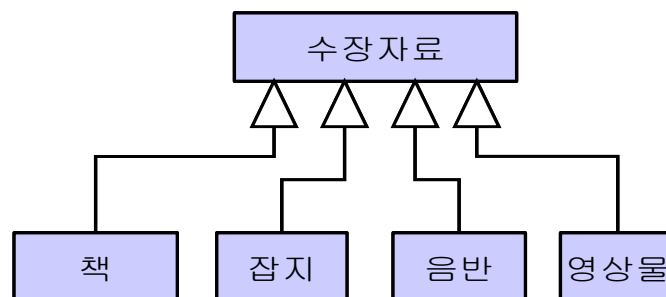
“정보”라는 수식어가 붙어있지 않다고 해서, 구별할 수 없다고는 말 할 수 없다. 따라서, 지금부터는 가능한 한 “XX정보”라는 표현을 사용하지 않도록 하겠다. 물론, 이와 같이 “정보”라는 수식어를 붙이지 않으면 불안하다는 사람들은 “XX정보”라는 표현을 그대로 사용해도 무관하다.

UML을 이용하여 “수장자료”를 표현하면 다음과 같다.



<그림 3-2> 수장자료

또한, “수장자료에는 여러 가지 종류가 있다”라는 사실을, UML에서는 다음 <그림3-3>과 같은 화살표시를 사용하여 표현한다.



<그림 3-3> 수장자료에는 다양한 종류가 있음을 나타내는 UML 도면

객체를 사용하여 사물을 생각할 때에 좋은 점은, “여러 가지 종류”의 “여러 가지”라는 개념에 대하여 필요하지 않는 한, 고려하지 않아도 좋다는 점이다. 빌려주거나 빌려오거나 하는 것은 “책, 잡지, 등등”이 아니라, “수장자료”라고 생각하기만 하면 된다.

또한, “영상물”이라고 하더라도, DVD 또는 VHS 등 여러 종류가 있으며, “각 영상물은 어떻게 처리하는가?”에 대해서는 필요하게 되었을 경우에 생각해보아도 좋다.

이와 같이, “수장자료에는 책 등, 여러 종류가 있다.”라는 사실을, UML에서는 “수장자료 클래스는 책 클래스를 일반화한 것이다”라고 표현한다. Java 등의 객체지향프로그래밍언어

에 있어서, “계승”과 거의 같은 개념이다.

### [이용자]

다음으로 고려해야 할 사항으로서, “책을 대출하는 사람”, 즉, “이용자”이다. 이용자에 대해서는 여러 종류가 있지는 않다. UML을 사용하여 작성하면 다음과 같다.



<그림 3-4> 이용자에 대한  
UML표기

여기서 주의해야 할 사항으로서, Actor로서의 “이용자”와 여기서 다루고 있는 개념객체로서의 “이용자”가 서로 다르다는 점이다. 이용자Actor는 사람을 나타내고 있다. 이에 반하여, 이용자개념객체는 시스템 내부에 있는 가상의 사람을 나타낸 것이다. 실제로, 시스템에서 조작하는 것은 개념객체로서의 이용자가 된다.

### [대출과 예약]

도서관에 책이 있고, 이용자가 존재하는 상태에서, “대출한다”, 또는 “대출해 준다”에 대하여 살펴보아야 한다. “도서관관리시스템”의 목적은, 책 대출을 시스템화하는 것이다.

그러면, “대출한다”라는 행위를, 개념객체로 정의하면 어떻게 될까? 그러나, 지금까지 객체지향을 학습해 온 사람들 중에는 “대출한다”라는 동사를 객체로 정의하면 안된다고 배웠다”라고 주장하는 사람이 있을지도 모른다.

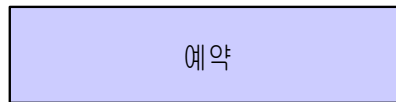
이에 대하여, 다음과 같이 생각하면 어떨까? “대출하는 일”, 즉 “대출”이라고 명사화하여 생각했을 때, 정보가 포함되어서 시스템에 있어서 의미가 있게 되고 자율적인 것으로서 생각할 수 있게 되지 않을까? 만약, 그렇다면, 객체로서 인정해도 된다.

실제로 도서관에서 대출을 한다면, 대출기록이 남겨지게 된다. 따라서, 이것 또한 객체로서 인정해도 좋다고 생각한다.



<그림 3-5> “대출”에 대한 UML  
도면

동일한 이유에서, “예약한다”라는 것도 객체로 인정해도 좋다.



<그림 3-6> “예약”에 대한 UML 도면

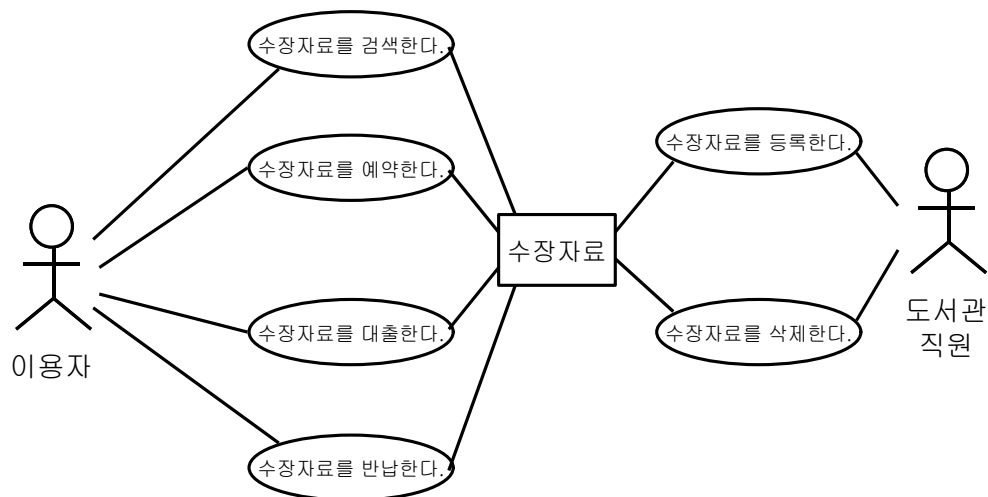
이로써 드디어 “사람이 책을 대출한다”를 위한 소재가 모두 준비되었다.

### 3.3 개념객체와 Use Case

앞에서 작성했던 Use Case Diagram을 참조하여 Use Case Diagram에 개념객체가 어떻게 관련되는지를 살펴보기로 하자.

개념객체를, Use Case Diagram 내부에 기입하고, 개념객체와 관련 있는 Use Case와의 사이에 선을 긋는다. 이와 같은 선은, “관련관계”를 나타낸다.<sup>26)</sup>

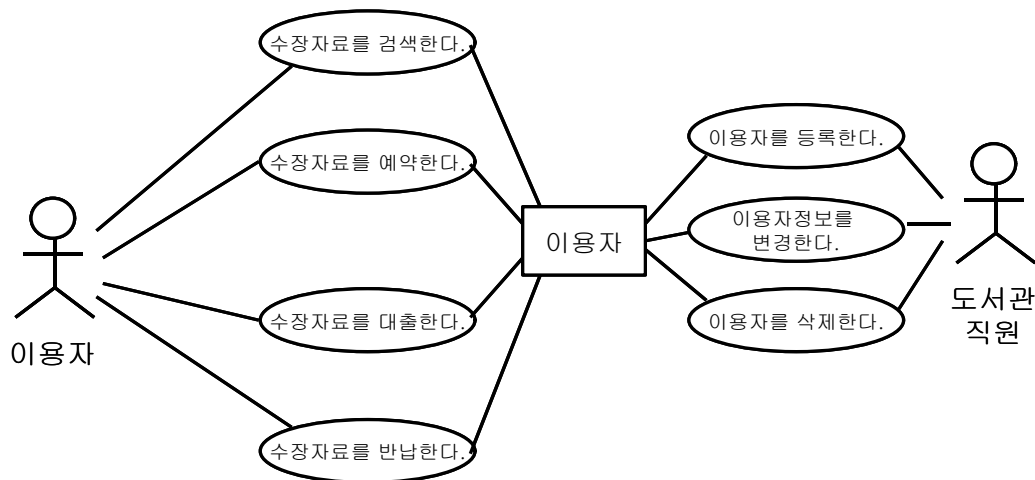
예를 들면, “수장자료”개념객체를 중심으로 생각해 보면, Use Case Diagram 내부에서 수장자료의 장소는 다음과 같이 된다.



<그림 3-7> Use Case Diagram 내부에 표현된 “수장자료”개념객체

<그림3-7>에서, “수장자료”개념객체는 각 Use Case의 목적어가 되어 있다. “이용자”에 대해서는 어떻게 될까?

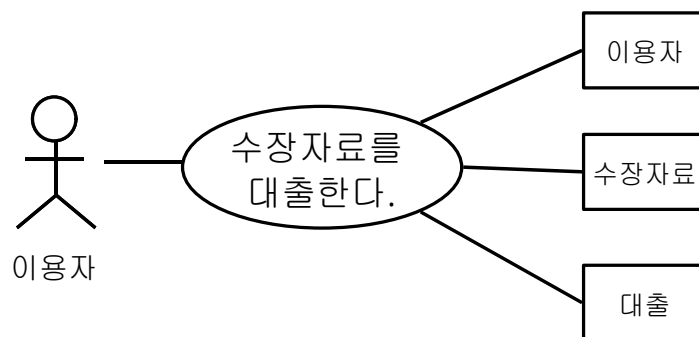
26) 단, “관련”이라고 하더라도, “무슨 관련?”이라고 생각할지도 모른다. 그러나, 여기서는 그렇게 엄밀하게 생각할 필요가 없다. Use Case Scenario에 나타난 매개변수, 사전조건/사후조건 또는 Use Case에 표현된 주어 및 목적어 정도로 이해하면 된다.



<그림 3-8> Use Case Diagram내부에 표현된 “이용자”개념객체

<그림3-8>에서는, “이용자”개념객체가 Use Case의 주어 또는 목적어가 된다. 주어가 되든 목적어가 되든 Use Case와 강력한 관계로 관련되어 있다.

반대로 이번에는, Use Case를 중심으로 살펴보기로 하자. 예를 들면, “수장자료를 대출한다”Use Case를 중심으로 살펴보면, 다음과 같이 될 것이다.



<그림 3-9> “수장자료를 대출한다”Use Case를 중심으로 표현한 예

다른 Use Case에 대해서도 일반적으로 이와 같은 형태가 된다.

“도서관관리시스템”의 개발에 있어서, 맨 처음에는 시스템의 외부에 대하여 “Actor, WorkFlow”, 그 다음으로는 시스템과 외부 경계에 대하여 “Use Case, Use Case Scenario”와 같은 순서로 살펴보았다. 지금 수행하고 있는 것은 시스템의 내부, 즉 객체의 가상세계로 한발자국 들어온 것이라고 할 수 있다.

### 3.4 생명주기(LifeCycle)

인간뿐만 아니라, 모든 생물 및 사물들 등 대부분의 객체들에게는 일생이 있다. 어느 날



어떤 별에서 태어나서 다양한 경험을 쌓고(다른 객체들과 메시지를 교환하고), 좌절과 성장을 거듭하다가(자신의 값과 상태를 변화시키면서), 그리고 어느 날 인가 저세상으로 가는 것이다. 개념객체도 이와 마찬가지로이다.<sup>27)</sup>

대부분의 객체들은, 일생동안 다양한 상태로 변화한다. 인간의 경우, 태어나서 유아기를 거쳐서 유치원/보육원에 가게 되고, 초·중·고·대학교를 거쳐 취직하여, 결혼하게 되어 자녀를 갖게 된다. 물론, 인생은 일직선처럼 되지 않는 경우도 있다. 한번 결혼한 후 다시 독신 상태로 되돌아오기도 한다.

이와 같이 객체가 어떠한 “상태”를 갖는지, 무엇을 계기로 해서 어떤 상태에서 다른 상태로 변화하는지를 UML에서는 “StateChart Diagram”이라는 도면을 사용하여 표현한다. 즉, StateChart Diagram은 객체의 일생(LifeCycle)을 나타낸다.

Use Case Scenario를 작성할 때 사용한 Sequence Diagram은, 주로 객체의 집합 사이에서의 상호수수작용을 나타낸다. 따라서, 한 개의 객체에 대한 일생을 나타내는 StateChart Diagram과는 표현하려는 관점이 서로 다르다.

WorkFlow를 작성하는데 사용했던 Activity Diagram도, 여러 개의 SwimLane에 걸쳐서 살펴보면 Sequence Diagram과 유사한 점이 있다. 그러나, 각각의 SwimLane을 제거해 보면 StateChart Diagram과 유사하다. 서로 다른 점으로서는, Activity Diagram이 “액션” 즉, 행위의 변화를 나타내고 있음에 반해, StateChart Diagram은 “상태”의 변화를 나타내고 있다.

StateChart Diagram이 “XXX초등학교시절”->“YYY중학교시절”->... 과 같이 표현하지만, Activity Diagram은, “XXX초등학교입학”->“YYY중학교입학”->... 과 같이 표현한다고 생각해도 좋다.

Activity Diagram을 사용하여 객체의 일생을 표현하는 것도 불가능하지는 않지만, 너무 복잡해진다. 따라서, 일반적으로, StateChart Diagram을 사용한다. 분석 및 설계 초기단계에서는, “무엇을 하는가?”보다는 “어떠한 상태인가?”를 살펴보는 것이 중요하기 때문이다.

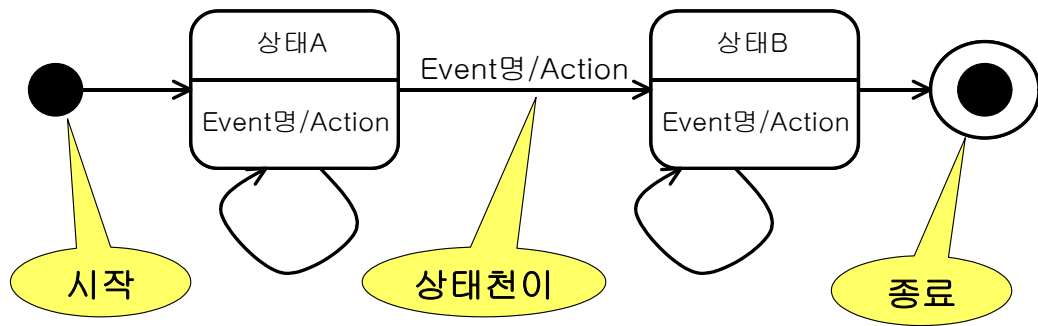
객체들 중에는, 너무 변화무쌍하거나 변화의 추이가 아날로그화 되어 “상태”로서 파악할 수 없는 것들도 있다. 예를 들면, 은행의 “계좌”객체를 살펴보자. 계좌의 잔고를 생각해 보면, 1원단위로 변화하므로, 상태를 파악하기 매우 어렵다. 이와 같은 객체를 StateChart Diagram으로 표현하기에 부적합하다. 이와 같은 “계좌”객체의 경우에는 “계좌가 비어있는” 상태, “잔고가 있는”상태, “대출”상태 등으로 생각하는 것이 좋을 것이다.

### 3.5 StateChart Diagram

StateChart Diagram은, 대략적으로 그려보면 다음과 같다.

---

27) 객체들 중에는 “일생”이라는 개념에 해당하지 않는 것들도 있다. 예를 들면, “값 객체”라고 부르는 객체가 이에 해당한다. 값 객체란, Integer 또는 Date 등과 같이 한번 값이 정해지면 계속 그 값을 유지하는 객체들을 가리킨다. 이와 같은 객체들도, 언젠가 태어나서, 언젠가는 소멸하는 것은 다른 객체들과 마찬가지로이지만, 살아있는 동안 계속해서 내부 값이 변화 없이 보존된다는 특징을 갖는다.



<그림 3-10> StateChart Diagram

객체의 일생은 검은 원형 아이콘에서 시작되어, 이중원형 아이콘에서 종료된다. 이와 같은 시작 및 종료상태는 표현하지 않더라도 충분히 알 수 있는 경우에는 생략해도 좋다. 이와 같이 시작과 종료를 나타내는 것은 Activity Diagram과 유사하다.

“상태”는 둥근 모서리를 갖는 사각형으로 나타낸다. 각 상태를 나타내는 사각형은 2개 구역으로 분할하여, 윗부분에는 상태의 이름을 기입한다. 상태의 이름은, “...하는 중” 또는 “...하는 상태” 등과 같이 표현한다. 특히, 상태의 이름은 동작(동사)이 아님에 주의해 주기 바란다.

아래쪽 부분에는, 해당 상태로 천이되어 들어오거나 나갈 때에 무엇이 일어나는가를 표기하며, 상세한 사항은 추후에 설명하기로 하겠다. 처음에는 윗부분에 상태의 이름만 기입해도 좋다.

해당 객체가 어떠한 상태를 갖는지 대략적으로 파악되었다면, 이번에는 어떤 상태에서 어떤 상태로 변화하는가(이것을 상태천이라고 부른다)를 살펴보기로 한다. 상태천이는 각 상태를 나타내는 사각형들 사이에 화살표선으로 나타낸다. 화살표선 위에는 어떠한 이유에서 상태가 변화 하는가 또는 무슨 일이 일어나면 상태가 변화하는가(이것을 Event라고 부른다)를 표현한다.

한 개의 상태에서 여러 개의 상태들로 천이되는 경우도 있으며, 한 개의 상태에 여러 개의 상태가 내포되어 있는 경우도 있다.

### 3.6 각 개념객체의 생명주기

#### [이용자의 생명주기]

“도서관관리시스템”의 각 개념객체들은 어떠한 일생을 갖고 있을까? 차례대로 StateChart Diagram을 작성해 보도록 하자.

우선, “이용자”개념객체부터 살펴보자. 맨 처음에 어떠한 상태가 있을 수 있는지 생각해 보면,

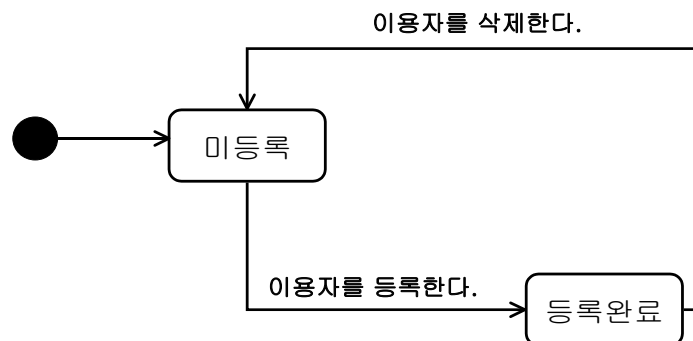
1. 이용등록하지 않은 상태
2. 이용등록 한 상태

3. 아무것도 대출하지 않은 상태
  4. 책 등을 대출한 상태
  5. 기한을 넘기고도 반납하지 않은 상태
- 와 같은 상태들을 생각해 낼 수 있다.



<그림 3-11> 이용자의 상태

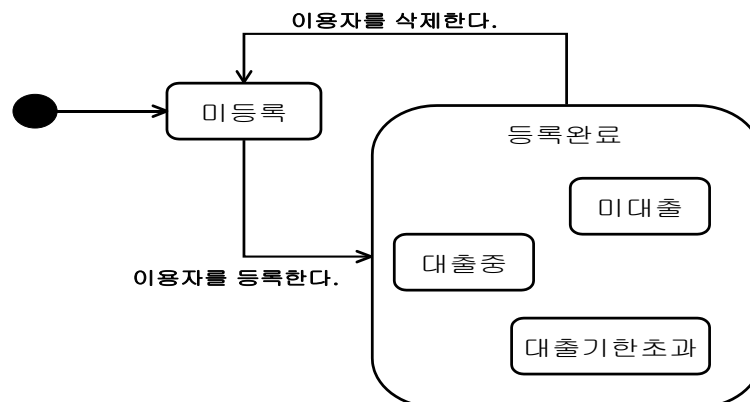
이용자가 생성되었을 때는 등록되지 않은 상태(미등록 상태)일 것이다. “이용자를 등록한다”를 수행함으로써 비로소 등록완료 상태가 된다. 또한, 자격이 상실되면, “이용자를 삭제한다”를 수행하게 되고, 다시 미등록 상태로 되돌아가게 된다. 이와 같은 천이가 반복될 것이다.



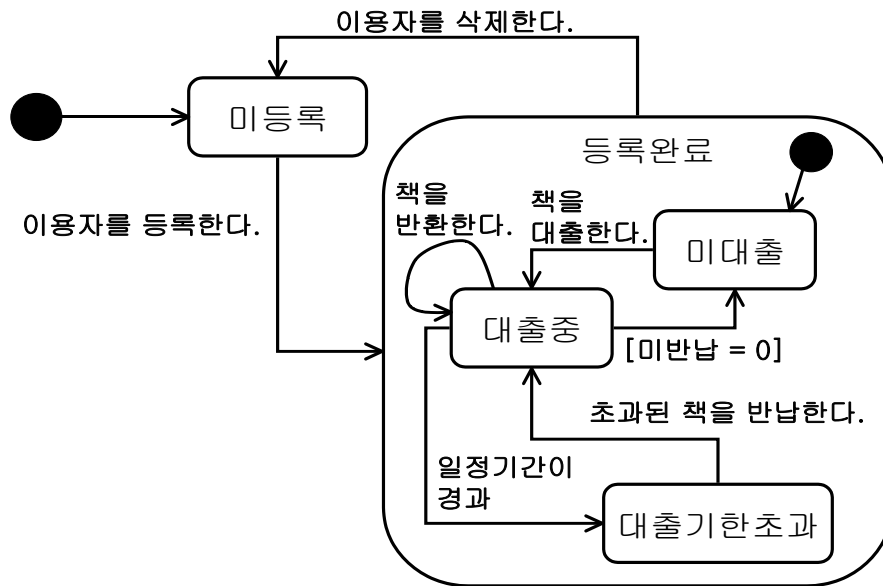
<그림 3-12> 이용자의 LifeCycle(1)

이밖에 나머지 3개 상태는, “등록완료상태”일 때에만 유효하다. 이와 같은 경우에는, 나머지 3개 상태들을 “등록완료상태”속에 내포시켜서 표현한다.

그런 다음에, 내포된 3개의 상태들 사이의 천이를 표현하면 다음과 같다.



<그림 3-13> 이용자의 LifeCycle(2)



<그림 3-14> 이용자의 LifeCycle(3)

앞서 살펴보았던 바와 마찬가지로, 여기서도 “Guard”를 사용하였다(<그림3-14>의 [미반납 = 0]부분. Guard를 만족하면 천이가 일어남을 의미한다).

여기서 중요한 3가지 사항은 다음과 같다.

- (A) Event가 Use Case 등과 일관성을 갖추고 있는가?
- (B) 상태가 Use Case의 사전조건/사후조건 등과 일관성을 갖추고 있는가?
- (C) 천이를 추적해 보면, 올바른 LifeCycle을 재현할 수 있는가?

이용자가 관계하는 Use Case로서,

1. 수장자료를 대출한다.
2. 수장자료를 반납한다.
3. 수장자료를 예약한다.
4. 이용자를 등록한다.
5. 이용자를 삭제한다.
6. 이용자정보를 변경한다.

등이 있었다. <그림3-14>에 표현한 Event는, 거의 위와 같은 Use Case에 포함되어 있다. <그림3-14>에 있는 Event들 중에서, “일정기간이 경과”라는 Event는 시간의 경과를 나타내고 있는 “Time Event”라고 부르는 것이므로, 여기서는 표현하지 않아도 상관없다.

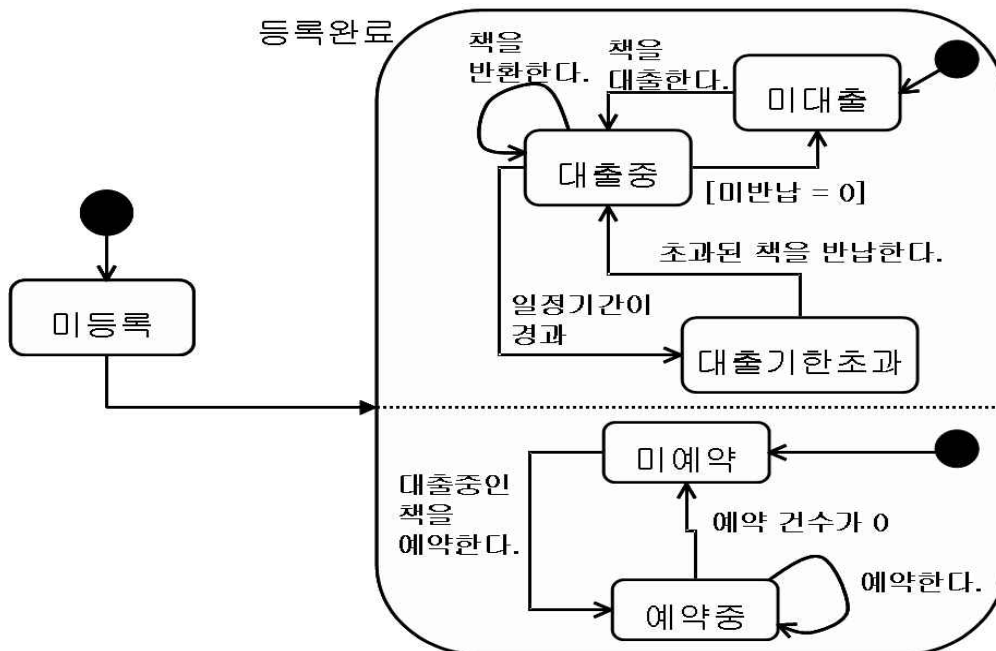
반대로, “수장자료를 예약한다”Use Case,와 “이용자정보를 변경한다”Use Case는, <그림 3-14>의 StateChart Diagram에서 표현할 곳이 없음을 알 수 있다.

“수장자료를 예약한다”라는 Event는, StateChart Diagram에 어떻게 기입하면 좋을까? “대출중”상태와 “미대출”상태에 덧붙여서 “예약중”상태와 “미예약”상태가 있다. 더구나, 이와 같은 2종류의 상태는 조합할 수 있어서, “대출중이면서 예약중인” 상태도 있을 수 있고, “미대출 중이지만 예약중인”상태도 있을 수 있다.

## [병행상태]

이와 같이 “독립된 상태들의 집합”을 보다 간단하게 취급하기 위하여, “병행상태”라는 개념이 있다.

내포된 상태를 점선으로 분할하고, 각각이 독립적으로 자유롭게 상태천이를 할 수 있다는 개념이다. 위의 예에서는, “예약”부분과 “대출”부분으로 분할한다. 이와 같이 분할하게 되면, 조합하여 생각하지 않아도 된다. 엄밀하게 생각해 보면, “상태 수 X 상태 수”가 되므로 매우 복잡하게 되어버린다.



<그림 3-15> 이용자의 LifeCycle(4)

2번째 주요사항(B)는 어떠한가? Use Case의 사전조건/사후조건에 나타난 상태는 전부 표현되어 있는가?

3번째 주요사항(C)는, 사실 소규모 시스템에서도 복잡해져서 인간이 눈과 머리로 추적할 수 있는 범위를 초월해 버리는 경우가 대부분이다. 따라서, 원래는 UML 도구를 사용하는 것이 좋다.<sup>28)</sup> 이와 같은 도구를 사용하면, 여러 가지 조건을 설정하여 실제로 StateChart Diagram을 동작시키거나 일시정지 시켜서 상태를 확인할 수도 있다.

## [왜 StateChart Diagram을 이용하는가?]

StateChart Diagram을 사용하여 객체의 생명주기(Life Cycle)를 살펴보는 것은, 객체의

28) 오픈소스로 제공되는 UML 도구들로서는, 앞서 소개한 IIOSS가 MDF(Model Debugging Facility)와 같은 형태로, StateChart Diagram의 실행기능을 제공해 준다. 그밖에도 Rhapsody(<http://www.ilogix.com/>) 등과 같은 상용CASE Tool들 중에는 이와 같은 기능들을 제공하는 것도 있다.

성질을 명확하게 하기 위해 매우 도움이 된다.

StateChart Diagram의 장점들 중에서, “모든 경우”를 추출할 수 있다는 점을 들 수 있다. StateChart Diagram에서는 한눈에 알기 어려울지도 모르겠지만, 표현방법을 바꾼 “상태천이표”라는 표를 작성해 보면 곧바로 알 수 있다.

또 다른 장점으로서, 작성방법에 따라서 StateChart Diagram 그대로 실행하거나 자동적으로 소프트웨어를 생성할 수 있다는 점을 손꼽을 수 있다. 즉, StateChart Diagram은, 일종의 “형식적 사양”이라고 할 수 있다.

그러면, 나머지 개념객체로서, “수장자료”, “대출”, “예약” 등이 있다. 이들 중에서 “대출”, “예약”은 각각 대출과 예약을 나타내는 객체가 된다. 대출 및 예약의 내용은 해당 시점에서 결정되어버리며, 나중에 변경되지 않는다.

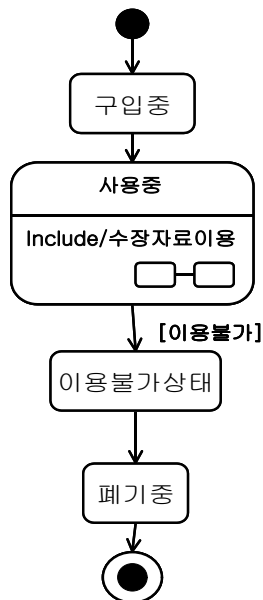
즉, “대출”객체와 “예약”객체는, 시간에 따라서 변경되는 상태를 갖지 않는, “값 객체”가 된다. 따라서, 생명주기를 작성할 필요가 없다.

<문제> “수장자료” 객체의 생명주기를 작성해 보시오.

<해답> 수장자료에 대한 대략적인 생명주기는, 다음과 같다.

1. (이용자로부터의 요청 또는 무엇인가에 의해) 구입하려고 한다.
2. 구입한 후 사용한다.
3. 사용하다가 노후 되거나 훼손되어 사용이 중지된다.
4. 사용중지가 되면 폐기된다.

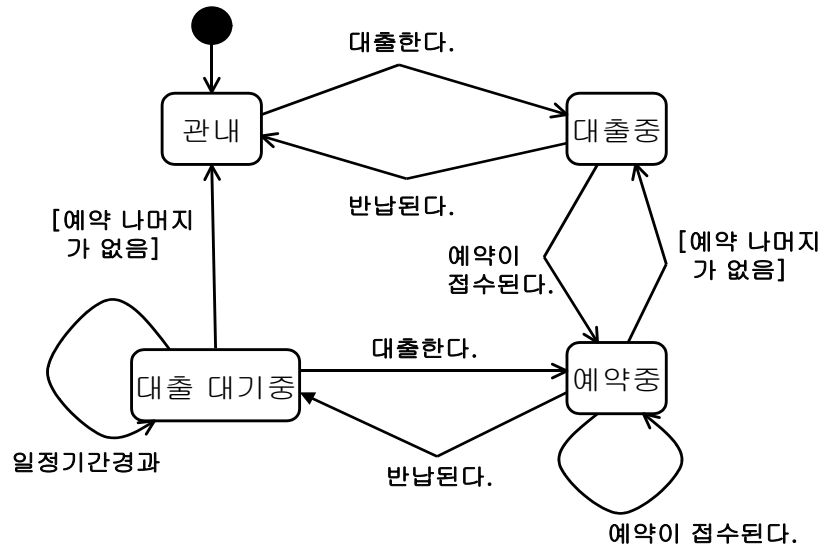
이와 같은 생명주기를 StateChart Diagram으로 표현하면 다음과 같다(<그림3-16>). 단, “사용중”에 대해서는 여러 가지 경우가 있으므로, 별도의 StateChart Diagram에 작성하기로 한다.



<그림 3-16>

수장자료의 생명주기

“사용중”부분에 대해서는 다음과 같다(<그림3-17>).



<그림 3-17> “사용중”인 소장자료의 생명주기

같은 상태천이를 나타내는 경우라고 하더라도, StateChart Diagram에서는 여러 가지 작성방법이 있으므로, 독자여러분들은 상이한 StateChart Diagram을 작성했을지도 모른다. 또한, 무엇을 상태 또는 Event로 하는가에 따라서, 다양한 형태로 표현될 수 있다. 현시점에서는, 같은 상태천이를 일으키는 한, 어느 쪽이 옳고 그른지는 구별하지 않는다.

한편, 위의 StateChart Diagram들을 작성하면서, 예약을 취소할 수 없다면 매우 불편하지 않을까?

<문제> “예약취소”를 추가하려고 한다. 지금까지 작성한 모든 모델들 중에서 수정 및 추가가 필요한 것은 어떤 것들인가?

<해답> 위 문제에 대해서는 여러 가지 해답이 있을 수 있다. 여기서는, 3가지 예를 살펴보기로 한다.

1. “예약을 취소한다”는 것은, Use Case가 될 것이므로 Use Case Diagram에 추가한다.
2. 위의 1에서 Use Case Diagram에 추가한다면, “예약한다” Use Case는 “예약한다” 외에 “통지한다”, “예약한 책을 대출한다”, “예약을 취소한다” 등, 몇 가지로 분해하는 것이 좋다.
3. Use Case Scenario도 추가할 필요가 있다.

## [참고] 상태천이표

상태천이표란, StateChart Diagram과 거의 같은 내용을 표 형태로 나타낸 것이다. 왜, 거의 같은 내용을 다른 형식으로 작성하는 것일까?

상태천이표를 사용하면, 해당 시스템에 어떠한 Event와 어떠한 상태가 있는지를 파악하

게 되어, Event와 상태 사이의 상태천이를 빠짐없이 체크하는데 매우 도움이 된다.

작성방법에는 여러 가지 형태가 있으나, 기본적으로는 거의 같다. 예를 들면, 표의 각 행에는 객체가 수신할 가능성이 있는 Event를, 그리고 각 열에는 객체가 갖게 되는 상태를 기입한다.

상태천이표(1)

이벤트 \ 상태	미대출중	대출중	기한초과
대출한다.			
반환한다.			

행과 열의 교차지점은, “해당상태에 있을 때, 해당 이벤트가 발생하면 어떻게 되는가”를 “조건XXX -> 상태YYY로 천이”와 같은 식으로 기입한다. “-” 표시는, “있을 수 없음”을, 그리고 “X”표시는 “이벤트 거부”를 나타낸다.

상태천이표(2)

이벤트 \ 상태	미대출중	대출중	기한초과
대출한다.	-> 대출중	제한시간이내 -> 대출중	X
반환한다.	-	나머지 == 0 -> 미대출중, Other -> 대출중	나머지 == 0 -> 미대출중, 초과 나머지 > 0 -> 기한초과, Other -> 대출중

이와 같이 하면, 모든 상태에 대하여, 어떠한 이벤트를 수신하더라도 어떠한 대응을 하는지 여부를 확인할 수 있게 된다. 그러나, StateChart Diagram에 비해서 상태천이를 추적하기 어렵다.

상태천이표를 토대로 하는 CASE 툴로서, ZIPC(<http://www.zipc.com/>)등이 있다.

### 3.7 개념객체의 책임

지금까지 개념객체에 대하여 살펴보았으나, 각각의 개념객체가 실제로 무엇을 나타내고 있는 것일까? “수장자료”, “이용자”, “대출”, “예약”이라는 이름이 붙여져 있으므로, 아마도 “이와 같은 사물”을 나타내고 있을 것이지만, 조금 더 명확히 할 수는 없는 것일까?

객체를 사용하여 사물을 만들 경우에, 재료가 되는 객체에는 각각 역할이 맡겨지게 된다. 각 역할이 명확하지 않으면 작성 완료된 사물도 불명확하게 된다. 사물이 명확하게 작성되지 않으면 조금만 기능을 추가하려고 해도 주변의 여러 객체들을 살펴보지 않으면 안된다. 어떤 객체에 대하여 조그만 수정을 하더라도 생각지도 않던 곳까지 영향이 끼쳐지게 된



다.

객체에 부과된 역할을 전문용어로, “책임(Responsibility)”라고 부른다. “책임”이라는 인간본위의 개념이 이와 같은 곳에 등장하는 것이 조금은 이상할지도 모르지만, 실제로는 매우 중요한 개념이다. 어떤 의미에서는 객체를 의인화하여 취급하고 있다고 생각해도 좋을 것이다.

객체를 단순히 데이터 또는 정보가 아닌, 인간과 같이 머리가 좋고 자율적인 존재로서 취급하는 것은, 객체지향 특유의 사고방식이다. 따라서, “제품객체에게 가격을 물어보면, 스스로 계산을 해서 대답해 준다” 등과 같은 의인화된 표현으로 말하는 경우가 많다.

책임은 “책무”라고도 말한다. 객체가 어떠한 서비스를 제공하는지를 규정하는 것이다. 예를 들어, “이용자”개념객체는 어떠한 책임을 가지고 있을까?

특정한 “이용자”개념객체는, 대개 도서관을 이용하는 한사람의 인간과 같은 동작을 수행하면 편리하다. 따라서, “이용자”개념객체는, 당연히 이용자의 이름, 연락처 등과 같은 기본적인 정보를 인지하고 있다. 책을 대출하거나 반납하고, 예약할 수도 있다. 물론, 지금 어떤 책을 대출하고 있는지도 알고 있을 것이다.

#### [위임과 협조상대]

대출하고 있는 책의 상세한 정보에 대해서는, 다른 객체(예를 들면, “수장자료”개념객체)에게 물어보지 않으면 알 수 없을 것이다. 이와 같이, “다른 객체의 힘을 빌리는 것”을 “위임”이라고 부른다. 또한 어떤 책임을 수행하기 위하여 함께 작업을 하는 다른 객체들을 “협조상대”라고 부른다<sup>29)</sup>. 이러한 개념도 인간세계에서 흔히 볼 수 있는 개념들이다.

“책임”과 “협조상대”가 명확해지면, 객체의 역할도 자연스럽게 확실해 진다. 이것을 정적인 모델링과 동적인 모델링을 통해서 명확히 하는 것이 지금 수행하고 있는 작업인 것이다. “이용자”개념객체의 책임을 살펴보기로 하자.

<문제> “이용자”개념객체의 책임을, 지금까지의 모델링을 토대로 열거해 보시오. 또한, 동시에 협조상대가 있는 경우에는, 협조상대도 열거하시오.

<해답>

1. 이름, 연락처 등의 기본정보를 인지하고 있다.
2. 기본정보에 변경이 발생할 경우에는 통보해 준다.
3. 책 등을 대출하거나 반납한다(대출, 수장자료).
4. 스스로 대출한, 또는 대출하고 있는 책 등에 대해서 인지하고 있다(대출, 수장자료).
5. 책 등을 예약하거나, 예약을 취소한다(예약, 수장자료).
6. 스스로 예약한, 또는 예약하고 있는 책 등에 대해서 인지하고 있다(예약, 수장자료).

29) 클래스, 책임, 협조상대 등과 같은 3개 개념을 한데 묶어서, CRC(Class, Responsibility, Collaborator)라고 부르기도 한다.

위의 목록에서 뒷부분의 괄호 속에는 협조상대를 열거하였다. “...한다”, “...할 수 있다”와 같은 책임을 “실행책임”, “...에 대해서 인지하고 있다”와 같은 책임을 “인식책임”이라고 부르는 경우도 있다.

### [책임의 개수]

만약 한 개의 객체에 책임이 20 또는 30개가 있다면, 너무 과도하다. 혼란스럽기도 하고, 원래 분할할 수 있는 것들을 함께 떠안고 있을지도 모른다.

개수 뿐 만아니라, 책임이 분산되어 있어서 부자연스럽기도 하므로, 체크해 보는 것이 좋을 것이다. 여러 가지 형태로 변환시켜보면 각 객체에 부과되는 책임의 개수는 수~수십 개가 되는 것이 일반적이다.

“이용자”개념객체는 인간을 나타내고 있으므로, 앞의 문제에 대한 해답에 열거했던 책임은 당연한 것이라고 생각할 수 있다.

그러나, 인간이 아닌 “수장자료”객체에도 중요한 책임이 부과되어 있다.

1. 제목, 저자 등의 기본정보를 인지하고 있다.
2. 이용자에 의해 대출되거나, 반환된다(이용자, 대출).
3. 지금, 누구에게 언제까지 대출되고 있는지를 인지하고 있다(이용자, 대출).
4. 이용자에 의해 예약되거나 예약이 취소된다(이용자, 예약).
5. 지금 누구에 의해 예약되어 있는지를 인지하고 있다(이용자, 예약).

<문제> “대출”개념객체와 “예약”개념객체의 책임을 각각 열거해 보시오.

#### <해답>

“대출”개념객체는, 한 번의 대출을 나타낸다. 이와 같은 추상적인 객체에 대해서도 다음과 같은 책임이 있다.

1. 언제 무엇이 누구에게 대출되었는지를 인지하고 있다.
2. 언제가 반납기한인지를 인지하고 있다.
3. 기한을 초과하고 있는지 여부를 인지한다.

“예약”개념객체에 대해서도 마찬가지이다. “예약”개념객체는 한 개의 예약을 나타낸다.

1. 언제 무엇을 누가 예약했는지를 인지하고 있다.
2. 예약에 대한 기한이 경과했는지 여부를 인지한다.

<문제> 여러 사람들이 예약하고 있던 책이 반납되었을 경우, 다음에는 누구에게 대출될 것 인지는 어떤 객체에게 문의하면 될까?

#### <해답>

“예약”객체일까? 그러나, “예약”개념객체는 “한 개”의 예약에 대한 정보만 가지고 있다. “여러 개”의 경우에는 대응할 수 없다. “예약”에게 책임을 추가해야 할까? 이렇게 하는 것도 한 가지 방법이다.

그러나, “한개”의 예약이 다른 예약에 대해서 알고 있지 않으면 않된다는 것은 “예약”객체에게 있어서 책임이 과도하다.

이 책을 다음에는 누구에게 대출해 줄 것인가는, 책 자신에게 문의해 보면 어떨까? “책은 그 정도로 똑똑하지 않다”고 생각할지도 모르겠지만, 바로 이런 점이 객체가 갖는 장점인 것이다.

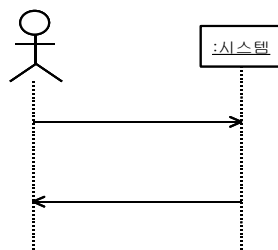
### 3.8 개념객체와 Use Case Scenario

개념객체가, 앞에서 열거했던 책임들을 올바르게 수행해 준다면 시스템이 정상적으로 동작하는지에 대하여 살펴보기로 하자.

현재 작업 중인 “도서관관리시스템”의 개발에 있어서, 맨 처음에 WorkFlow(Activity Diagram)를 작성하였다. 이것은 주로, Actor들 사이의 상호수수작용에 대한 모습을 나타내고 있다. 다음으로는, Use Case Scenario(Sequence Diagram)를 작성함으로써, Actor와 시스템 사이의 상호수수작용을 살펴보았다.

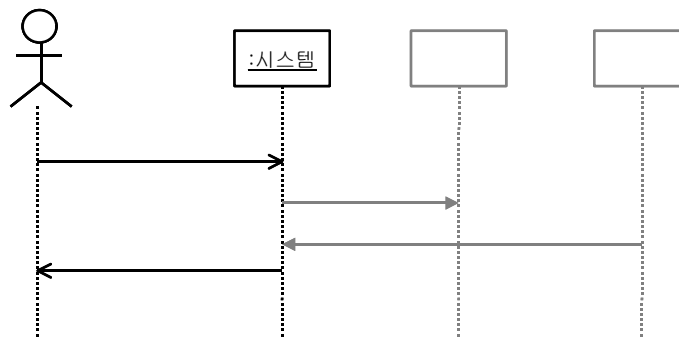
그러면, 다음 단계로서, Sequence Diagram에 개념객체를 추가하여 상호수수작용을 살펴 보도록 한다.

앞서 작성했던 Sequence Diagram은, <그림 3-18>처럼 “시스템”의 내부를 어떤 의미에서 블랙박스로서 취급하였다고 할 수 있다.



<그림 3-18>  
시스템의 내부가  
보이지 않는다.

이에 반하여, 지금은 위의 블랙박스 내부가 조금씩 보이기 시작한다.



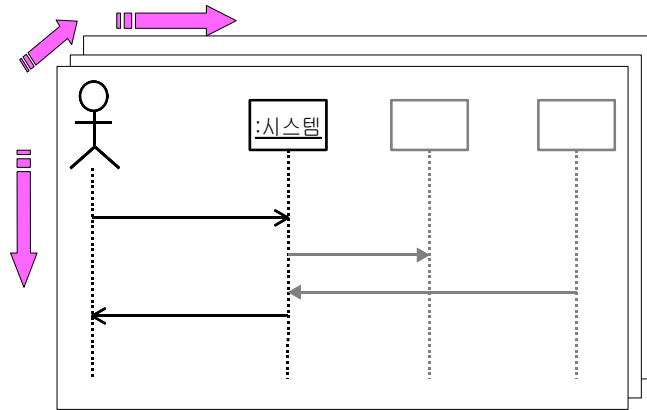
<그림 3-19> 시스템의 내부가 서서히 보이기  
시작한다.

## [“시스템”의 내부에 존재하고 있는 개념객체]

“시스템”의 내부에 존재하고 있는 것은 개념객체들이다. 앞에서는, 각각의 개념객체의 생명주기를 살펴보았지만, 여기서는 개념객체들을 조합하여 시스템 전체가 올바르게 동작하는지를 확인해 보려고 한다.

또한, Fractal한 UML모델링, 즉, 상세화를 조금씩 진척시켜나가는 방법을 Sequence Diagram을 통해 살펴보면 다음과 같다.

- Sequence Diagram이 횡적으로 확장되어 간다. 즉, Sequence Diagram에 참가하는 객체가 점점 상세화/분열/증가되어 간다.
- Sequence Diagram이 종적으로 확장되어 간다. 즉, 객체들 사이의 커뮤니케이션에 대한 표현이 점점 상세화 되어 간다.
- Sequence Diagram자체가 확장되어 간다. 즉, 보다 다양한 상황과 이에 대응하는 시나리오가 고려되어 간다.



<그림 3-20> 시나리오의 상세화

마지막 단계에서는, 객체 및 메시지의 세밀도는 프로그래밍언어가 제공하는 세밀도가 되며, 그 부분까지 모든 시나리오들을 작성할 수는 없다. 따라서, “어느 정도”선에서 중지하지 않으면 안된다. 그러나, 실제로는, 치밀하게 완성도가 높은 모델(특히, 시나리오에 관해서)을 작성하는 것보다는, 어떻게 “어느 정도”선에서 중지해야하는지를 판단하는 것이 보다 더 어려운 문제이다.

여기서는, 참고적으로 몇 가지 시나리오들을 작성해 보자. 이것도 또한 “Use Case Scenario”라고 부르는 것이 조금은 이상하므로, 그다지 일반적인 용어는 아니지만, “개념시나리오”라고 부르기로 한다.

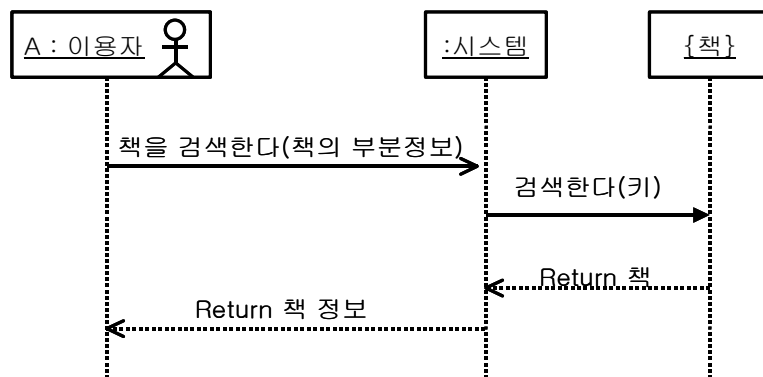
## 3.9 개념시나리오

개념시나리오, 기본적으로 Use Case Scenario를 확장한 것이다. 즉, Actor와 시스템 사이의 상호수수작용부분은 거의 변화가 없다. 그 대신에, 시스템의 내부를 개념객체들의 집합으로 표현한다.

예를 들면, 우선, “책을 검색한다”메시지를 시스템 내부에 있는 개념객체들 사이의 상호 수수작용으로 분해한다. 시스템 내부에서 책을 검색하기 위해서는, 누구(어떤 객체)에게 무엇을 의뢰하면 좋은지를 생각해보자.

책 자체를 나타내는 개념객체는 있었지만(“수장자료”개념객체 및 “책”개념객체), “책들의 집합”을 나타내는 개념객체는 아직 존재하지 않고 있다. 앞으로 “XXX들의 집합”과 같은 것들이 많이 출현할 것이므로, 책들의 집합을 간단히 “{책}”으로 나타내기로 한다<sup>30)</sup>.

위와 같이 표현하면, Sequence Diagram은 다음과 같이 된다.



<그림 3-21> “책”개념객체를 포함하고 있는 Sequence Diagram

시스템의 왼쪽은 “현실세계”에서 일어나고 있는 사항을 나타내고 있으며, 오른쪽은 시스템의 내부, 즉, “가상세계”에서 일어나고 있는 사항들을 나타내고 있다. 이것만으로는, “Use Case Scenario와 크게 다른 점이 없지 않은가!”라고 생각할지도 모르겠다.

### ["대출한다"의 개념시나리오]

계속해서 “대출한다”개념시나리오도 살펴보기로 하자.

책을 실제로 대출하기 위해서는 대출하려는 사람이 대출할 수 있는 자격을 갖추고 있는지 여부와 더불어서, 대출하려는 책이 실제로 대출가능한지 여부를 살펴볼 필요가 있다(“사전 조건”). 따라서, 다음과 같은 사항들을 살펴보기로 한다.

- 우선, 이용자를 찾아서 대출가능한지 여부를 체크한다.
- 다음에는, 책이 대출가능한지 여부를 체크한다<sup>31)</sup>.

만약 두 가지 조건이 모두 OK가 되면, 새로운 대출이 되므로, “대출”개념객체를 1개 생성한다. 이것으로 일단 책을 대출하는 것이 성공된다.

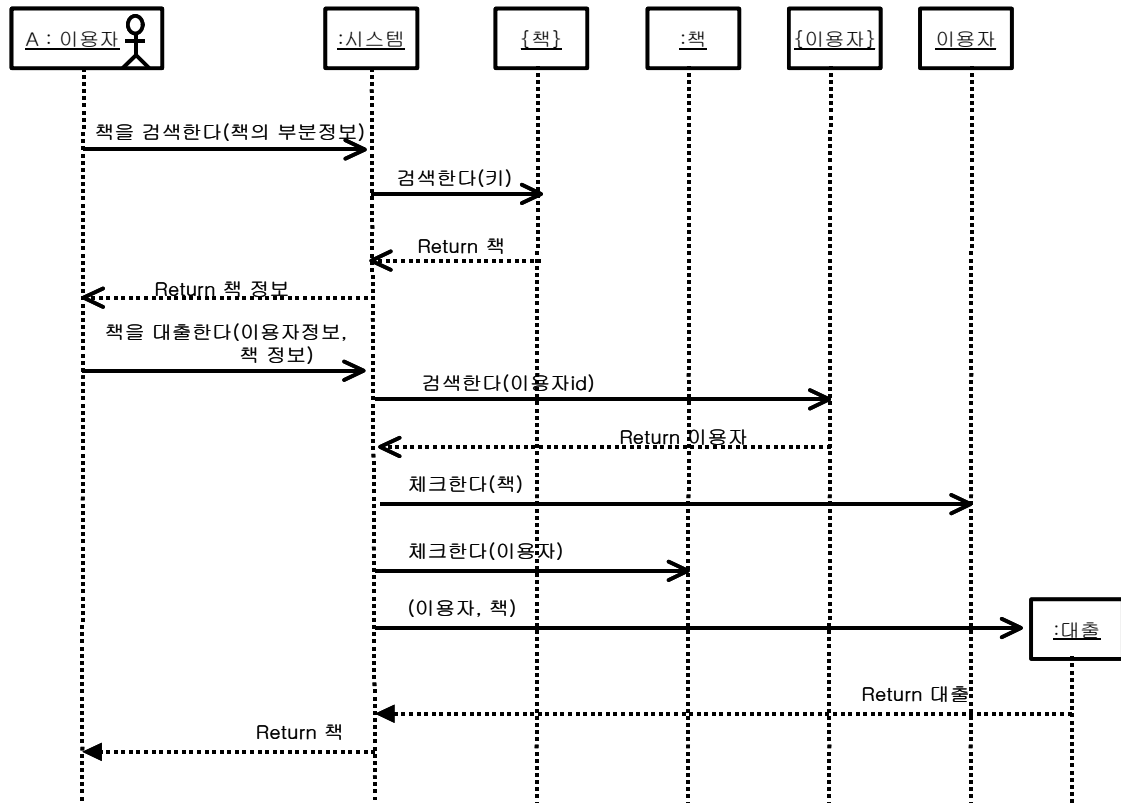
30) { }는 집합을 나타내는 기호이다.

31) 실제로는 이와 같은 순서는 상관없을 듯 하다. 그러나, Sequence Diagram에서는 양자택일하지 않으면 작성할 수 없다. 이것은 Sequence Diagram을 작성할 때의 어려운 점이다. 중요한 경우에는 주석을 붙여 두도록 하자.

<그림3-22>의 오른쪽에는 1개의 “책”개념객체가 등장하고 있다. 이것은, 검색에 의해 찾아낸 책이다. 그리고, 또 한 가지 기억해 둘 것이 있다.

제일 오른쪽 끝에 있는 “대출”에 메시지 화살표가 향하고 있다. 이것은, “객체 생성”을 나타낸다. 또한, “객체의 소멸”은 커다란 X표시로 나타낸다.

이것으로서, 상당히 “시스템”다운 모습이 되었다.



<그림 3-22> “책을 대출한다”에 대한 개념시나리오

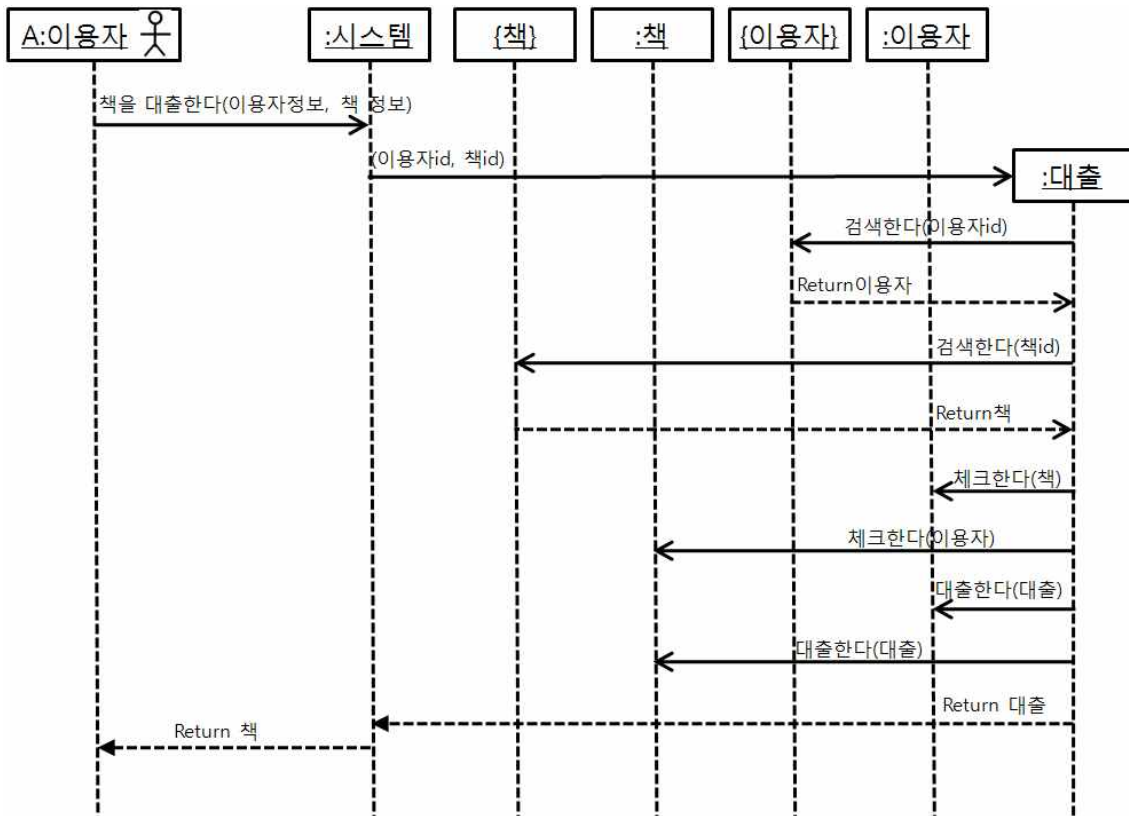
### [책임에 대한 체크]

한편, 위의 개념시나리오에서 각 개념객체들의 책임이 올바르게 수행되고 있는지 여부를 체크해 보도록 하자. “책을 대출한다”에서는, 시스템이 책 객체를 검색하여 무엇인가를 수행하고, 이용자 객체를 찾아서 무엇인가를 수행하는 등, 많은 사항들이 시스템에 집중되어 있다. 각 개념객체가 수행하고 있는 것이라고는, 체크하는 것뿐이다. 이것은, “대출”개념객체가 단순히 데이터와 마찬가지로 사용되고 있음을 의미한다.

“대출”개념객체의 책임은, 대출을 수행하고 반납을 처리하는 것이다. 따라서, <그림 3-22>에 나타난 시스템이 수행하고 있는 업무들은 “대출”개념객체가 수행해야 한다<sup>32)</sup>.

32) 바꾸어 표현하면, “대출에 관한 지식이 시스템의 여기저기에 산재해 있다”고 할 수 있다. “대출이란 무엇을 수행하는 것인가”라는 지식은, “대출”개념객체 내부에 패키징해 두지 않으면 안된다. 객체란, 지식의 모듈인 것이다.

따라서, 이와 같은 사항들을 토대로 <그림3-22>를 수정하도록 하자.



<그림3-23> 그림3-22를 수정한 개념시나리오

### ["반납한다"에 대한 개념시나리오]

다음으로, “반납한다”, “예약한다”에 대한 개념시나리오를 작성해 보자.

우선, “반납한다”개념시나리오부터 살펴보자.

이것은 “대출한다”와 거의 같다. 우선은, “대출”개념객체는 어디에 있는가? “대출”개념객체를 찾기 위해서는, 누가 무엇을 대출했는가(이용자의 정보 및 책에 대한 정보)가 필요하다. <그림3-22> 및 <그림3-23>의 “대출”개념객체는, 새로이 작성한 것이므로 찾을 필요가 없으나, 이번에는 어떻게 찾는 것이 좋을까?

해결방법은 다음과 같은 3가지가 있다.

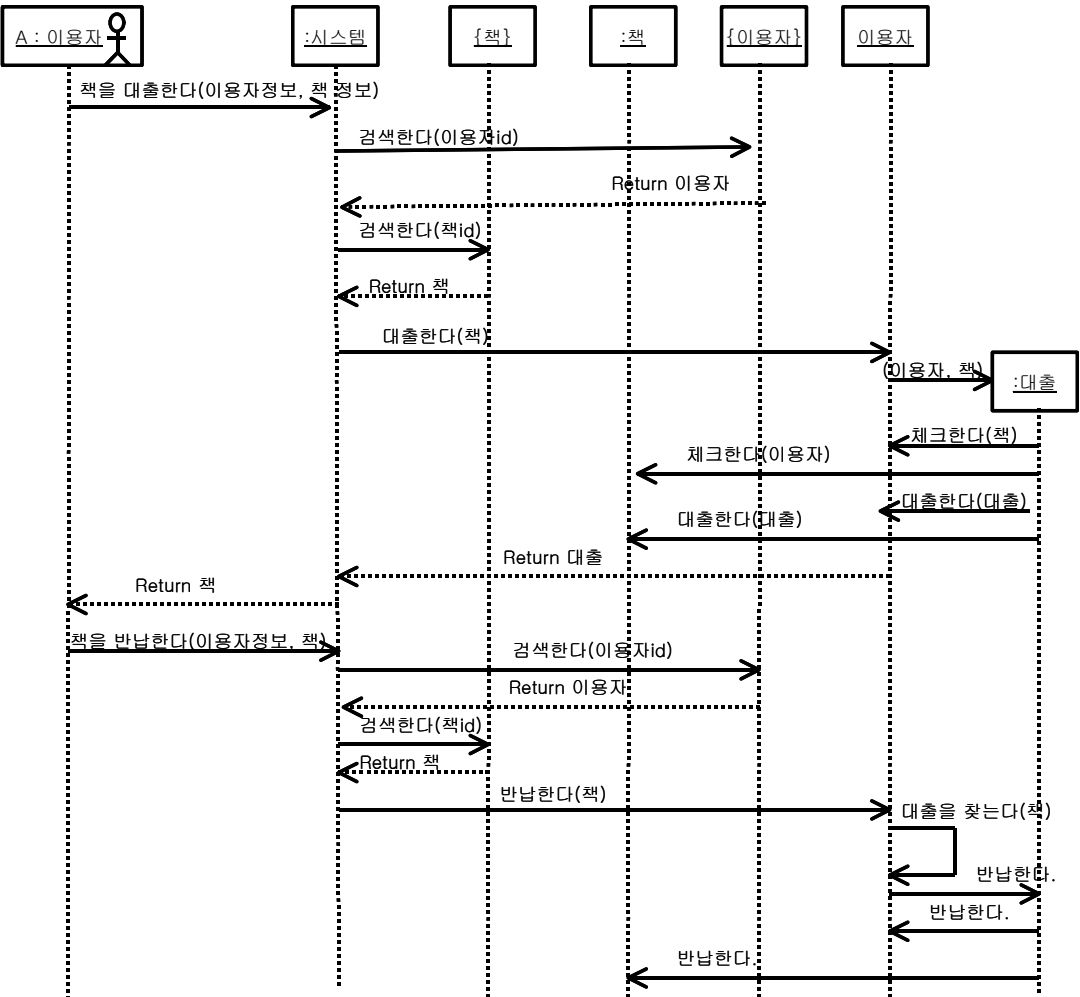
- 이용자로부터 시작하여 추적하면서 찾는 방법
- 책으로부터 시작하여 추적하면서 찾는 방법
- 대출을 직접 찾는 방법

어떤 방법도 좋아 보이지만, 모두 장단점을 가지고 있다. 이용자 또는 책으로부터 시작하여 찾는 방법은, “대출한다”개념시나리오와 정합성이 없어지게 된다. 왜냐하면, 앞에서는 직접 “대출”개념객체를 작성했기 때문이다. 이용자와 책의 대칭성이 붕괴된다.

대출을 직접 찾게 되면, “{대출}”에 해당하는 것(대출 전체)을 가정하지 않으면 안된다.

<문제> 이용자로부터 시작하는 “반납한다”개념시나리오를 작성하시오. 필요하다면, “대출한다”개념시나리오도 이에 맞추어서 변경해도 좋다.

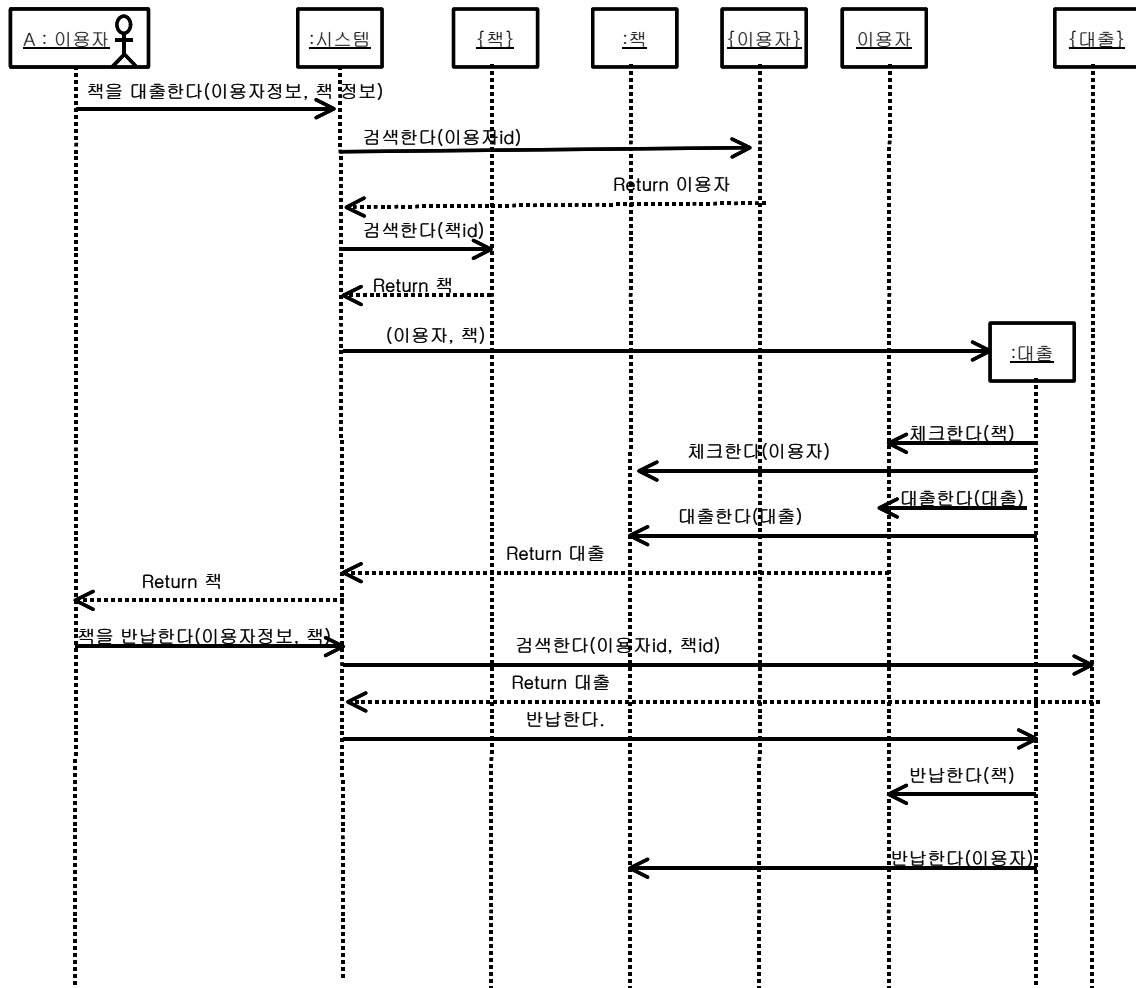
<해답>



<그림 3-24> “반납한다” 개념시나리오의 해답 예(1)

앞서 살펴보았던 3가지 방법들 중에서, 3번째 방법을 사용하여 작성하면 다음과 같이 된다.





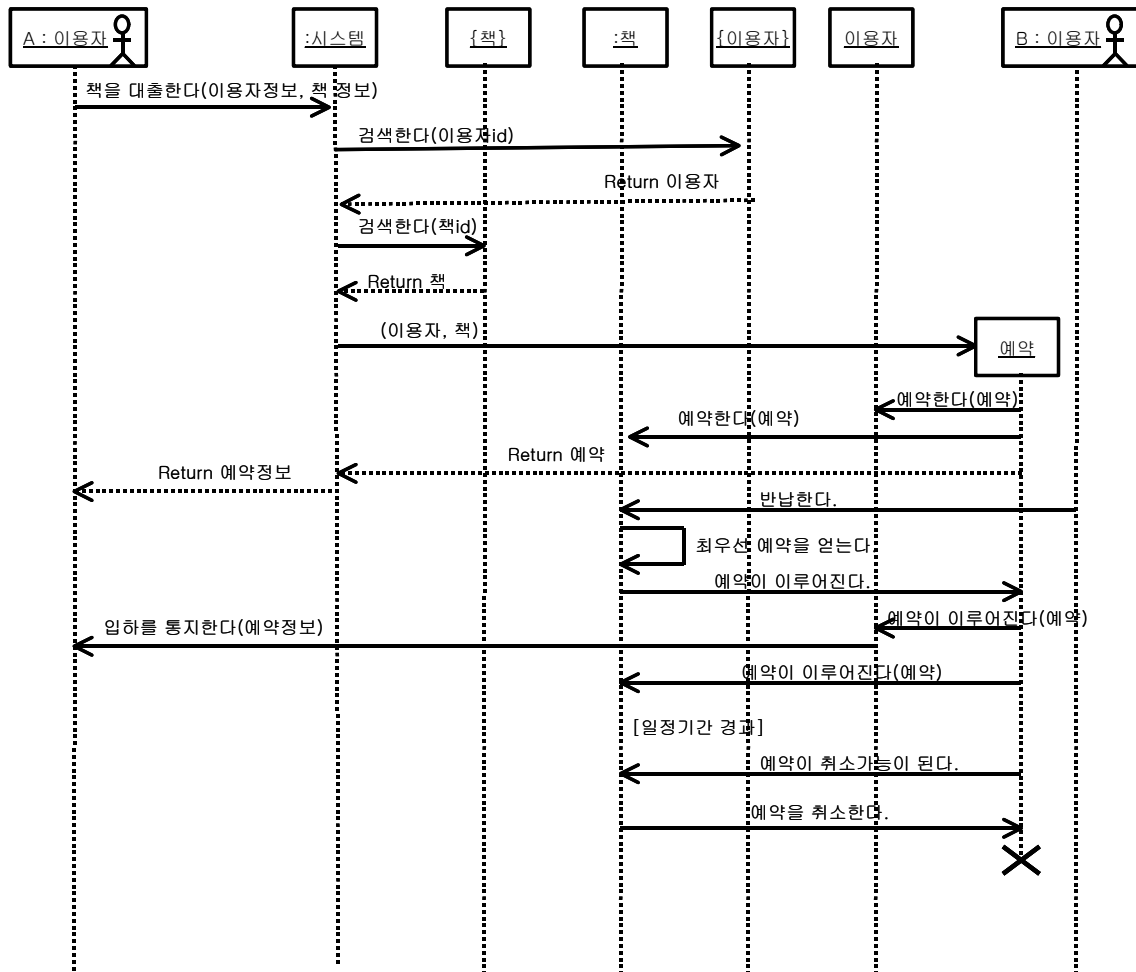
<그림 3-25> “반납한다” 개념시나리오의 해답 예(2)

#### ["예약한다"에 대한 개념시나리오]

“예약한다” 개념시나리오도 마찬가지로이다. “예약한다”의 경우,

- 예약순서가 되어, 일정기간 내에 대출하러 온 경우
- 예약순서가 되었지만, 일정기간 내에 대출하러 오지 않아서 취소되어버린 경우
- 예약순서가 되어, 도서관에서 보유하고 있는 사이에 다른 사람이 대출하러 온 경우

등과 같이 여러 가지 시나리오를 생각할 수 있다. 여기서는, 2번째 경우에 대해서 검토해보기로 한다.



<그림 3-26> “예약한다”에 대한 개념시나리오

FUM에서는, 개념시나리오를 작성하는 주요 목적이 시물레이션과 체크를 하기 위해서이다. 종이 위에서, 개념객체가 어떻게 동작하는가를 시물레이션해 보는 것이다. 이러한 시물레이션에 의해서, 개념객체의 책임이 올바르게 부과되었는지를 체크할 수 있게 된다. 프로그램을 작성하기에 앞서서, 시물레이션과 체크를 수행하여 시행착오를 겪어보는 것이 모델링의 좋은 점이라고 할 수 있다. 시행착오를 많이 거칠수록 좋다고 생각한다.

단, FUM에서는, 개념시나리오를 너무 상세하게 작성하거나, 있는 모든 경우를 찾아내서 살펴보는 것을 목적으로 하지 않는다. 노력에 비해 얻는 이득이 많지 않기 때문이다<sup>33)</sup>.

### 3.10 개념객체의 정적 구조

이즈음에서 개념객체의 정적 모델링에 대하여 살펴보기로 하자.

개념시나리오를 작성하기 전에, 개념객체의 책임에 대해서 살펴보았다. “책임”이라는 자체는, UML에 포함되어 있지 않다. 그러나, UML이 사용되기 훨씬 전부터, 특히, 객체지

33) 단, “시나리오”라는 사고방식 자체는, 대단히 유용한 개념이다. 작성하려는 시스템이 본래 어떤 것인가를 이해하기 위해서는 납득될 때까지 시나리오를 철저히 작성해 보는 것이 좋다.

향 세계에서는 널리 사용되어 왔던 개념이다. UML에서는, 객체의 책임이라는 개념을 주로 “정적구조도<sup>34)</sup>”라는 도면에 표현한다.

정적구조에는 크게 나누어서 2종류가 있다. 한가지는, 각 객체의 내부구조, 또 다른 한 가지는 여러 객체들 사이의 관계구조이다. 이들 중에서 특히, 객체의 내부구조가 “책임”에 해당하고, 객체들 사이의 관계구조(의 일부)가 “협조상대”에 대응한다.

또한, 내부구조는 속성과 동작으로 나누어진다.

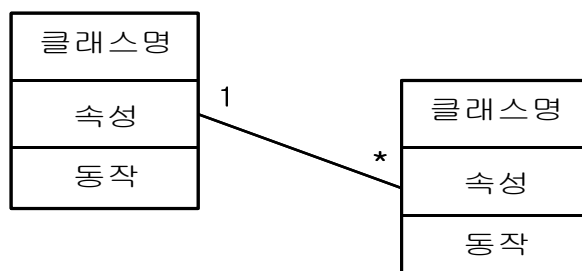
UML 정적구조	CRC
클래스	클래스
내부구조	책임
속성	인지책임
동작	실행책임
관계구조	
관련	협조상대
기타	특별히 없음.

<표 3-1> UML의 정적구조와 CRC

“관련”에 대해서는, Actor와 Use Case 사이의 관련관계와 유사한 의미를 갖는다. 즉, 클래스들 사이의 관련관계는 해당 인스턴스들 사이의 다양한 수수작용을 의미한다. “속성”은, 객체지향 프로그래밍언어의 경우, 인스턴스변수 및 데이터 멤버 등에 대응한다. 즉, 객체가 갖고 있는 내부데이터 및 내부상태라고 표현해도 좋다.

한편, “동작”은, 객체지향프로그래밍언어에서의 메소드 및 멤버함수에 해당한다. 바꾸어 표현하면, 객체가 수신할 수 있는 메시지, 또는 수신한 메시지에 반응하여 실행되는 동작이다.

정적구조도는 다음과 같이 표현한다.



<그림 3-27> 정적구조도

사각형을 3개영역으로 분할하여, 제일 윗부분에는 해당 객체의 이름을, 가운데 부분에는

34) “정적구조도”는, 이른바 “Class Diagram”을 의미한다. 어찌하여, “정적구조도”라는 표현을 사용했는가하면, 클래스 뿐만아니라, 다양한 “사물”의 내부구조, 상호관련 관계 등을 표현하려고 하기 때문이다. 정적구조도에 한정하지 않고, 앞으로의 UML은 도면의 종류에 대한 경계가 점차 없어지게 되어, 결국 1개의 도면에 다양한 정보들을 표현하도록 변화해 갈 것으로 생각된다.

속성들을 나열하고, 제일 아랫부분에는 동작들을 열거한다. 속성 및 동작은 필요 없는 경우에는 표기하지 않아도 좋다. UML 도구들 중에는, 속성 및 동작의 표시/비표시 기능을 제공하는 것도 있다.

지금부터 차례대로, “도서관관리시스템”에 등장한 개념객체들의 정적구조를 살펴보기로 한다.

### 3.11 개념객체의 속성

개념객체의 속성이란, 해당 개념객체가 무엇에 대해서 인지해야 하는가를 나타낸 것이다. 개념객체의 속성을 인지하기 위해서는, 예를 들면, 개념객체의 지식책임 및 개념객체의 생명주기가 갖는 상태를 힌트로 삼는다. 우선은, 해당 개념객체가 “무엇에 대해서 인지해야 하는가”를 살펴보기로 하자.

예를 들면, “이용자”개념객체의 속성을 인지하기 위해서는, “이용자”개념객체가 가져야만 되는 정보를 생각해 보는 것으로부터 시작한다.

- 이름, 연락처 등의 기본적인 정보

(우선은 최소한의 정보만 기입해 둔다. 나중에 추가해도 문제없다)

- 지금 대출하고 있는 책 등에 관한 정보

(단, 이것은 “대출”개념객체에 해당한다. 속성이라고 하지 않고 나중에 객체들 사이의 관계구조로서 취급한다)

- 현재 예약하고 있는 책 등에 관한 정보

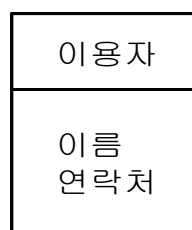
(상동)

- 과거에 대출했던 책에 관한 정보

(상동)

데이터베이스의 스키마설계를 수행하고 있는 것이 아니므로, 나중에까지 고려한 완전무결한 정보를 도출할 필요는 없다. 지금까지 작성한 생명주기 및 시나리오를 참조하면서 누락된 것이 없다면 일단 완료해도 좋다.

이것을 도면으로 작성해 보면 다음과 같다.



<그림 3-28>

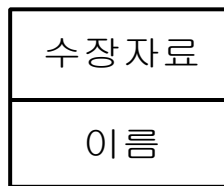
“이용자”개념객체의 속성

개념 시나리오에서는, “이용자의 id”를 사용했었다(<그림3-22>). 이것을 명시하려는 경우

에는, id라는 속성을 추가해도 좋다. 그러나, 객체는 기본적으로 식별가능(즉, 아무 일도 하지 않더라도 식별자, id를 갖는다)하다는 전제사항이 있다. 따라서, 여기서는 id를 추가하지 않도록 한다.

#### [기타 개념객체들의 속성]

위와 동일한 방법을 사용하여, 수장자료에 대해서도 살펴보자. 책 또는 잡지, CD 등에 따라서, 실제 내용은 상당히 상이하며, 각각의 상세한 내용까지 현시점에서 고려하게 되면 매우 복잡해 질 것이므로, “일단은 최소한”의 정보만을 도면에 나타내기로 한다.



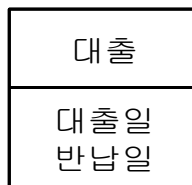
<그림 3-29>  
“수장자료”  
개념객체의 속성

한편, “대출”개념객체는 어떠한가? 적어도, “누가, 무엇을, 언제 대출했는가, 언제 반납했는가”에 대한 정보가 필요할 것이다.

여기서, “누가”는 이용자 객체에 해당하며, “무엇을”은 수장자료객체를 가리키므로, 나중에 미루기로 하고 나머지 “언제”에 대하여 살펴보자.

프로그래밍언어에서는, 프로그램을 작성할 경우, “언제”는 어떻게 나타내는가 라는 문제로 고민하게 된다. 1970년 1월1일부터의 날짜수로 나타낼 것인가? 아니면, 문자열로 나타낼 것인가? 그렇다면 문자열의 형식은? 라이브러리에 날짜를 나타내는 클래스가 있으므로 그것을 이용할까? 등등, 생각해야 할 사항이 너무 많아진다.

단, 이와 같은 부분에 대해서는 편리한 모델링언어를 이용하는 경우, 단순히 “날짜”로 기입해 버려도 좋다.



<그림 3-30>  
“대출”개념객체  
의 속성

<문제> “예약”개념객체의 속성을 도면으로 나타내시오.

<해답> 필자가 생각한 “예약”개념객체의 속성은 다음과 같다.

예약
예약일 입하일 취소완료

<그림 3-31>  
“예약”개념객체의  
속성

### 3.12 개념객체의 동작

개념객체의 동작이란, 해당 개념객체가 어떠한 메시지를 수신할 수 있는가에 대한 정보를 나타낸다. 이것은, 일반적으로 “동작”이라는 용어가 갖는 의미와는 약간 상이하므로 주의하기 바란다.

개념객체의 동작을 파악하기 위해서는, 예를 들면, 개념객체의 실행책임 및 생명주기의 이벤트, 시나리오의 메시지 등을 고려해야 한다. 일단은, 해당 개념객체가 “무엇을 하지 않으면 안되는가”를 생각해 보기로 하자.

예를 들면, “이용자”개념객체가 수행해야하는 것으로서는,

- 대출자격을 체크한다.
- 대출한다.
- 반납한다.
- 예약한다.

등이 있다. UML로 표기하면 다음과 같다.

이용자
이름 연락처
예약한다. 대출가능한가? 대출한다. 반납한다.

<그림 3-32>  
“이용자”개념객체  
의 동작

## [동작의 매개변수]

동작은, 객체지향프로그래밍언어에 있어서, 메소드 및 멤버함수에 대응한다. 따라서, 실제로는, 매개변수 및 매개변수의 데이터 형, 반환 값 및 반환 값의 데이터 형 등 여러 가지를 고려하지 않으면 안된다.

그러나, 맨 처음에는 이와 같은 상세한 부분에 대해서는 무시해도 좋다. 만약, 현시점에서 파악된 것이 있을 경우에는, 기입해 두어도 좋다. 예를 들면, 개념시나리오를 살펴보면, 대략적으로 어떠한 매개변수가 어떠한 메시지에 필요할 것인지 알 수 있으므로, 다음과 같은 도면을 작성할 수 있다.

이용자
이름 연락처
예약한다(예약) 대출가능한가?(수장자료) 대출한다(대출) 반납한다(대출)

<그림 3-33>

“이용자”개념객체의  
동작에 매개변수를  
추가한 예

“수장자료”개념객체에 대해서도 마찬가지로 다음과 같이 작성할 수 있다.

수장자료
이름
예약된다(예약) 대출한다(대출) 대출가능한가?(이용자) 반납된다(대출)

<그림 3-34>

“수장자료”개념객체의  
동작

개념시나리오를 살펴보고, 어떠한 동작과 어떤 메시지가 대응하고 있는가를 체크해 보기 바란다. 한편, “대출”개념객체의 경우, 다음과 같이 표현할 수 있다.

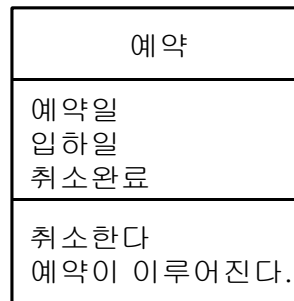
대출
대출일 반납일
반납한다

<그림 3-35>

“대출”개념객  
체의 동작

<문제> “예약”개념객체의 동작을 도면으로 나타내시오.

<해답>



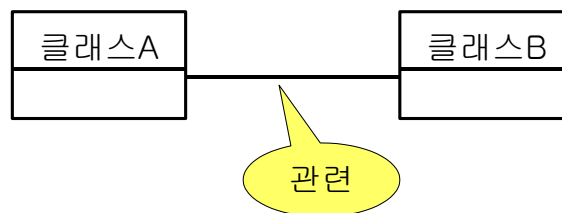
<그림 3-36>

“예약”개념객체의 동작

### 3.13 관련

UML의 각 도면에 있어서, 요소들 사이를 연결하는 선분에는 다양한 종류가 존재한다. 가장 중요한 것이, 클래스와 클래스 사이의 “관련관계”이다<sup>35)</sup>.

관련관계는, 클래스와 클래스가 “서로 인지하고 있다”는 사실을 나타내고 있다. 인지하고 있는 상대에게는 직접 메시지를 보낼 수 있다<sup>36)</sup>. 객체지향 세계에서는, 객체들 사이의 상호수수작용이 메시지를 보내는 것과 같은 의미를 갖게 되므로, 관련관계가 중요한 역할을 수행하게 된다. 관련관계는 2개의 클래스를 연결하는 실선으로 나타낸다.



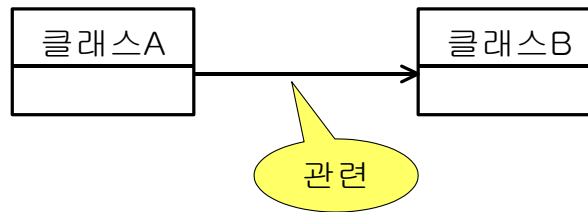
<그림 3-37> 관련

특히, A는 B를 인지하고 있지만, B는 A를 인지하지 못하는 경우, 즉 일방통행 상태인 경우에는 B로 향하는 화살표 선을 사용한다.

35) 정확하게 표현하자면, 관련은 클래스들 사이의 관계 뿐 만아니라, “클래스와 유사한 것(분류자, Classifier)”들 사이의 관계이다. Actor와 Use Case도 분류자의 일종이다. 따라서, Use Case View를 고려할 때, Actor와 Use Case 사이에도 관련관계를 나타내는 선분을 긋는 것이 가능하다. 그밖에도 분류자들은 많이 존재한다. 관련을 그을 수 있는 것 이외에도, 분류자는 많은 공통적인 성질이 있다.

36) 최근에는 객체를 사용하여 실제로 사물을 만들고 있는 사람들 사이에도, “메시지를 보낸다”는 표현이 통용되지 못하는 경우가 있는 듯하다. “메시지를 보낸다”는 표현은, “메소드를 호출한다” 또는 “멤버함수를 호출한다”는 표현과 거의 같은 의미를 갖는다. 객체들 사이에는 메시지를 서로 주고받음으로써 상호수수작용을 하는 것이 객체 계산모델의 대략적인 모습이다.



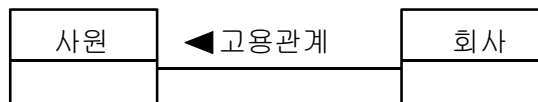


<그림 3-38> 일방통행 형 관련

### [관련에 이름을 붙이는 방법]

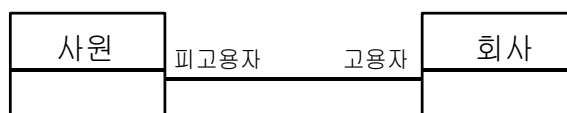
선분위에 이름을 붙여서 나타낸다. 관련의 경우, 2가지 방법이 있다.

첫 번째 방법은, 관련자체에 이름을 붙이는 경우이다. 예를 들면, 회사와 그 회사의 직원 사이의 관련관계(이런 경우, 상호 인지하고 있음)는 “고용관계”이다. 방향성이 중요한 경우에는, 방향을 나타내는 조그만 삼각형을 표기하도록 한다.



<그림 3-39> 관련자체에 이름을 붙인 예

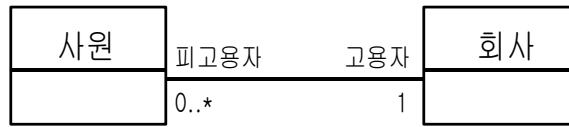
또 다른 방법으로서, 관련되어 있는 각 클래스에 역할의 이름을 붙이는 방법이 있다. 클래스A의 입장에서는 클래스B가 어떻게 인지되는가를, 관련을 나타내는 선분의 클래스B쪽에 표기하며, 반대로, 클래스B의 입장에서 클래스A를 바라보았을 때의 역할을 클래스A쪽에 기입한다. 예를 들면, 회사의 입장에서 직원을 바라볼 경우는 “피고용자”, 직원의 입장에서 회사를 바라볼 경우에는 “고용자”라는 역할명을 붙일 수 있다.



<그림 3-40> 관련되어 있는 클래스에  
역할명을 붙인 예

### [다중도]

관련에는, “다중도”라는 것을 기입하는 경우도 있다. 다중도는, 해당 관련이 몇 대 몇의 관계인가를 나타낸다. 관련의 양쪽 끝부분에 숫자를 기입하여, 몇 대 몇의 관계인지를 나타낸다. 나타내려는 다중도 값에 폭이 존재하는 경우에는, “..”을 2개 숫자 사이에 끼워 넣어서 표현한다. 예를 들면, “0..\*”인 경우에는, “0부터 무한대”라는 의미가 된다. 이와 관련된 예를 살펴보기로 하자.



<그림 3-41> 다중도를 나타낸 예

회사에는 일반적으로 여러 명의 사원이 있으며(사원이 없는 경우도 있음), 특히 사원의 수를 몇 명까지로 제한하는 경우는 거의 없으므로, 사원 쪽에는 “0..\*”으로 기입한다. 사원은 1개 회사에만 소속되어 있는 것으로 한다면, 회사 쪽에는 “1”로 기입한다.

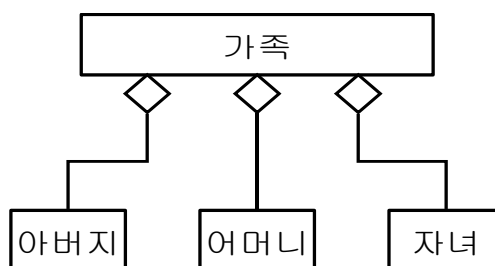
모델링에 있어서, 우선은 어떠한 요소와 관계가 있는가를 파악하는 것이 중요하다.

화살표 선 또는 이름, 다중도와 같은 장식물도 중요하지만, 부수적인 정보를 나타내므로, “서서히” 명확하게 표현하면 된다. 따라서, 이와 같은 부수적인 정보들에 너무 집착하게 되면, 결국 본질적인 모델링 작업이 진행되기 어려운 경우가 발생한다. 맨 처음에는, 기입하지 않더라도 컴파일하는 것이 아니므로, 누구도 불만을 이야기하지 않을 것이다. 이러한 점은 프로그래밍언어와 다른 모델링언어만의 장점이기도 하다.

### 3.14 집약과 합성

관련관계에는 몇 가지 종류가 있다. 우선, 여기서는 “집약(Aggregation)”이라는 관계에 대해서 살펴보기로 한다.

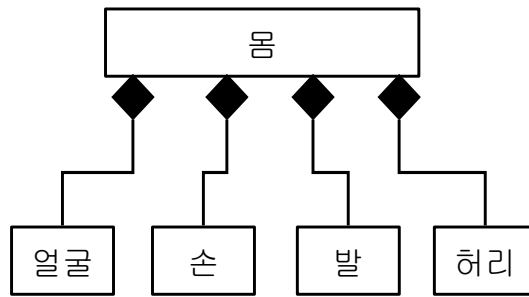
관련은, “서로 인지하고 있는” 관계를 나타내고 있으며, “집약”은 조금 더 강력한 관계를 나타낸다. “집약”은, 한쪽이 다른 쪽의 구성요소가 되어 있는 경우에 사용한다. 예를 들면, 아버지, 어머니, 자녀가 모여서 가족을 구성하고 있는 경우에 대하여, 아버지, 어머니, 자녀와 가족은 집약관계에 있다고 표현한다.



<그림 3-42> 집약의 예

한편, “합성(Composition)”은, 더 강력한 관계를 나타내며, “일심동체”와 같은 의미를 갖는다. 엄밀하게 정의하면, 생명주기가 거의 같은 경우 또는 분리되면 의미가 없는 경우, 그리고 한쪽이 다른 쪽의 생성/소멸을 제어하고 있는 경우 등에 사용한다.

표기방법은 다음과 같이 검은 색 다이아몬드를 붙여서 나타낸다.



<그림 3-43> 합성의 예

관련, 집약, 합성을 각각 구별하여 사용하는 경우, 어떤 관계가 적당할지는 곧바로 판단할 수 있는 경우도 있지만, 그렇지 못한 경우도 있다. 왜냐하면, 의미적인 측면이 강하기 때문이다.

단, 너무 고민하게 되면 오히려 복잡하게 되므로, 적당한 선에서 판단해 두도록 한다. 맨 처음부터, 이와 같은 구별이 치명적으로 작용하는 경우는 없다.

단지, 주의해야할 점으로서는, 한 개의 관련 양쪽 끝에 다이아몬드가 붙는 경우는 있을 수 없다는 점이다. 어느 쪽이든 부품이 되며, 다른 한쪽은 전체가 되어야 한다. 특히 합성의 경우, 부품이 동시에 여러 개의 구성체에 속할 수 없다.

집약도 합성도, 모두 1대다 관계가 되는 경우가 많지만, 반드시 그렇다고는 말할 수 없다. 집약 및 합성과 관련의 방향을 나타내는 화살표시를 조합하여 사용하는 경우도 있다.

### 3.15 개념객체의 관련

개념객체의 관련이란, 자신의 책임을 수행하기 위하여 어떤 객체와 상호수수작용을 하는 것을 의미한다.

개념객체의 관련을 파악하기 위해서는, 예를 들면, 개념객체의 실행책임 및 시나리오의 메시지 매개변수 등을 조사한다. 우선은, 개념객체의 책임을 수행하기 위하여, “어떤 객체와 상호수수작용을 하는가”, “어떤 객체와 서로 인지하는 사이인가” 등을 살펴보기로 한다.

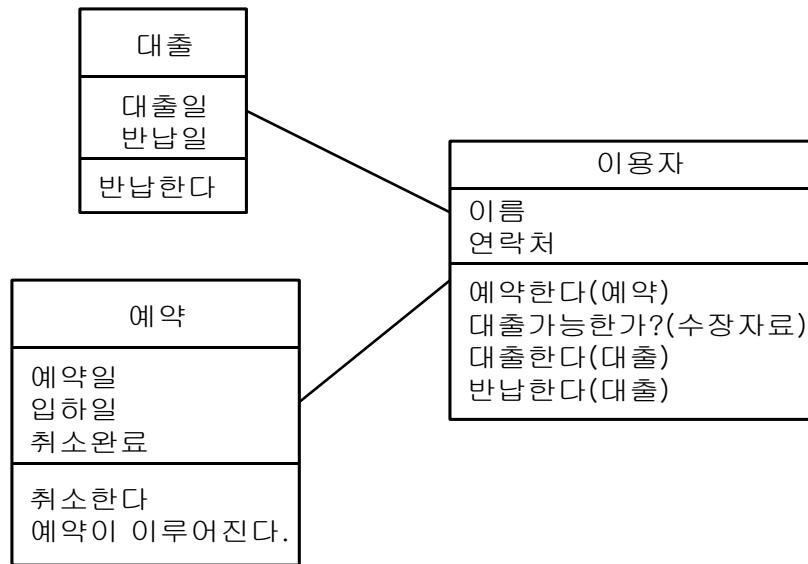
또한, 개념객체의 인지책임 및 개념객체의 생명주기가 갖는 상태들 등으로부터도 찾아낼 수 있다. 앞에서는 개념객체의 속성을 살펴보았으나, 각 속성들 중에서 다른 개념객체로 표현된 것들은, 해당 개념객체와 관련되어 있게 된다.

예를 들면, “이용자”개념객체에서,

- 이름, 연락처 등의 기본적인 정보(이것들은 앞서 속성으로서 살펴보았다).
- 지금 대출하고 있는 책 등에 관한 정보(이것은 “이용자”로부터 “대출”로의 관련)
- 지금 예약하고 있는 책 등에 관한 정보(이것은 “이용자”로부터 “예약”으로의 관련)
- 과거에 대출했던 책에 관한 정보(이것은 “이용자”로부터 “대출”로의 관련)

등을 열거할 수 있다.

우선, 위에서 파악한 관련관계를 도면에 표시해 보도록 하자.



<그림 3-44> 개념객체의 관련

#### [관련에 대하여 고려해야할 사항들]

다음과 같이 관련에는 몇 가지 고려해야 할 사항들이 있다.

- 관련명 또는 역할명(관련은 무엇을 나타내고 있는가?)
- 방향성(어느 쪽에서 어느 쪽을 인지하고 있는가?)
- 다중도(몇 대 몇의 관계인가?)

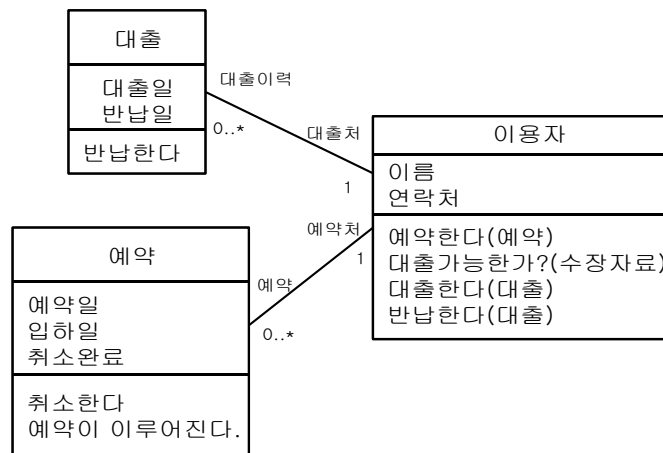
우선, 각각의 관련들이 무엇을 나타내고 있는 것일까? 이용자에게 있어서 “대출”은 자신이 지금까지(지금도) 대출했던 수장자료를 나타내고 있다. 따라서, 이용자 입장에서 살펴보았을 때 “대출이력”이라는 역할명을 붙여두기로 하자. 만약, 도서관의 정책이, 현재 대출하고 있는 것들만 관리하는 것으로도 충분하다고 한다면<sup>37)</sup>, “대출”이라는 역할명을 붙이도록 한다. 반대로, “대출”의 입장에서 살펴보았을 때, 상대방은 “대출처”라는 역할명을 붙이도록 한다.

예약에 있어서, 이력은 필요 없을 것 같아 보인다. 지금 예약하고 있는 책들의 일람표와 같은 것으로 충분할 것이다. 따라서, 그대로 “예약”이라는 역할명을 붙이도록 하자.

방향성은, 양방향이다. 이용자도 무엇을 대출했는지, 그리고 무엇을 예약했는지 알 수 없으면 곤란할 것이고, 대출 및 예약에 있어서도, 누가 대출 및 예약을 했는지 알 수 없으면 곤란해 질 것이다. 양방향성 관련관계를 표기할 경우, 화살표시는 필요없다.

다중도는 어떻게 될까? 일반적으로, 이용자의 입장에서 보면, 대출이력 및 예약은 “0..\*”이며, 대출 및 예약의 입장에서 이용자를 살펴보면, “누군가가 대출했다(또는 예약했다)”가 되므로, 반드시 “1”이 되어야 한다.

37) 도서관에 따라서는, 개인정보보호 측면에서 이용자의 대출이력을 보존하지 않는 경우도 있다. 또한, 과거의 대출이력에서 이용자별로 “이번 달의 추천도서”와 같은 것을 통보하려고 한다면 이용자별로 과거의 대출이력이 중요하게 될 것이다.

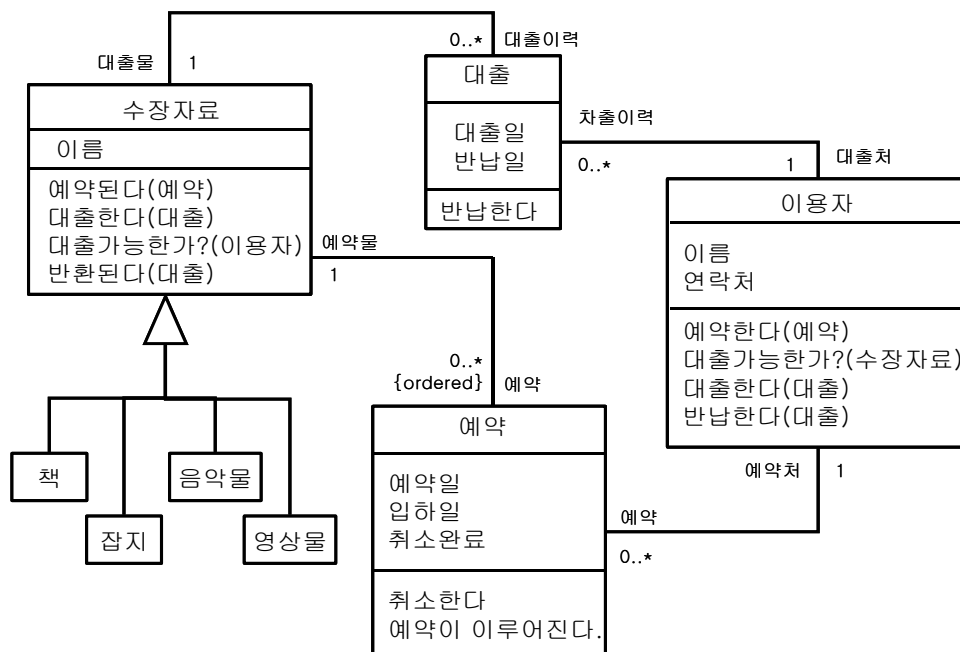


<그림 3-45> 그림3-44를 상세화한 도면

여기서, “관련이 있다”라는 것과, 개념시나리오에서 “상호수수작용이 있다”라는 것 사이에는, 대응관계가 존재함에 주의하기 바란다. 기본적으로 상호수수작용을 하기 위해서는, 상대방을 인지하고 있지 않으면 안되는, 즉, 관련관계가 없으면 안된다.

<문제> “수장자료”개념객체, “대출”, “예약”개념객체들에 대해서 관련관계를 조사하여 1장의 정적구조도를 작성하시오(주의: <그림3-45>에 “수장자료”개념객체를 추가하는 형태로 작성할 것).

<해답> 필자가 작성한 도면은 다음과 같다.<sup>38)</sup>



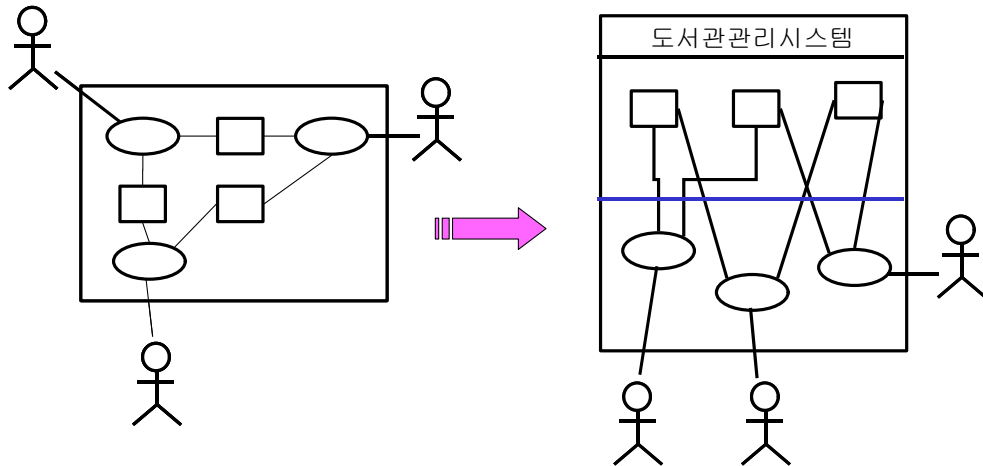
<그림 3-46> “수장자료”, “대출”, “예약” 등의 개념객체들 사이의 관련

38) 이외에도, “관련클래스”를 사용하는 방법이 있다. 이와 같은 경우, 대출과 예약이, 이용자와 수장자료의 관련 클래스가 된다. “관련클래스”에 대해서는 여러 참고도서 등을 통해서 익히기 바란다.

### 3.16 시스템 전체는 1개의 객체

지금까지 개발하려는 시스템을 한 개의 커다란 사각형으로 생각하면서 모델링 해 왔다. 이러한 사각형 외부에는 Actor가 존재하고, 내부에는 Use Case 및 개념객체, WorkFlow, Scenario, StateChart 등이 기입되었다.

이러한 사각형을 다음과 같이 실제로 한 개의 커다란 “객체”로서 생각할 수도 있다.



<그림 3-47> Use Case View로부터 시스템 객체로의 변환

이와 같은 객체를 “시스템전체를 나타내는 객체”라는 의미로, “시스템 객체”라고 부르기로 한다. 매우 개략적인 표현이지만, 시스템 객체에서는 개념객체가 속성에 해당하고, Use Case가 동작에 대응한다. 각 Actor들은 또다른 별개의 시스템 객체들이 된다.

지금까지 일관되게 객체를 소재로 하여, “Fractal하게” “도서관관리시스템”을 개발해 왔다. 따라서, 이러한 시스템 전체를 나타내는 한 개의 커다란 객체가 개발의 출발점이 된다.

이것으로 드디어 출발점에 도달하였다. 이후에는, 이와 같은 작업을 Fractal하게 반복하여 궁극적으로 “동작하는” 객체들의 집합으로 변형시켜가게 된다.

### 3.17 개념모델링의 종료

지금까지 수행한 작업들은, 현재 작성하려는 시스템을 외부에서 살펴보면서 바깥쪽의 형태를 만드는 작업이었다. 여기까지의 작업을 “개념모델링”이라고 부른다.

지금까지의 작업에서는 어떻게 시스템을 만들지, 그리고 시스템의 내부를 어떠한 구조로 만들 것인가에 대해서는 아직 고려해 보지 않았다. 이에 대해서는 제4장에서 살펴보기로 한다.

또한, 한 가지 더 주의해야 할 사항으로서, 개념모델링이 이것으로 완성된 것이 아니라는 사실이다. 개념모델링 뿐만 아니라, 어떠한 모델링에서도 완성은 있을 수 없으며, 모델링이란 완성을 목적으로 하지 않는다.

우수한 모델작성자는, 완성상태로 모델을 만들 수 있는 사람이 아니라, “모델링을 어디쯤에서 종료하면 좋을지를 알고 있는 사람”이라고 할 수 있다.

특히, 개념모델은 사용자 및 고객의 희망과 요구에 의해, 제한 없이 진화한다. 또한, 시간이 경과하면 점점 변화해 갈 가능성이 있다. 따라서, “좋았어! 일단 여기까지 되었으면 그 다음으로 진행!”이라고 파악하는 것이 매우 중요하다.

그러나, 개중에는 불안해하는 사람이 있을 수 있다. “상세하게 하지 않으면 나중에 문제가 되어 비용과 시간이 2배 또는 3배가 되어 버리지 않을까?”라고 우려할 수도 있다. 틀림 없이 객체지향 이전의 방식, 또는 객체지향을 사용하더라도 잘 활용하지 못하게 되면 이와 같은 현상은 충분히 일어날 수 있다. 앞서 설명한 바와 같이, 객체지향을 사용하여 이와 같은 사태를 회피하기 위한 지침으로서 다음과 같은 2가지가 있다.

한가지는, 현실세계를 토대로 하여 시스템을 작성하므로, 나중에 기능을 추가하거나 상세 사항을 첨가하더라도 좀처럼 이상한 현상은 발생하지 않는다는 점이다.

또 다른 한 가지는, 종래의 기법과 비교해서 모듈성이 높은 객체라는 소재를 사용하여 조립함으로써, 일부분의 변경이 전체에 걸쳐 영향을 끼치는 현상은 발생하지 않는다는 점을 들 수 있다.



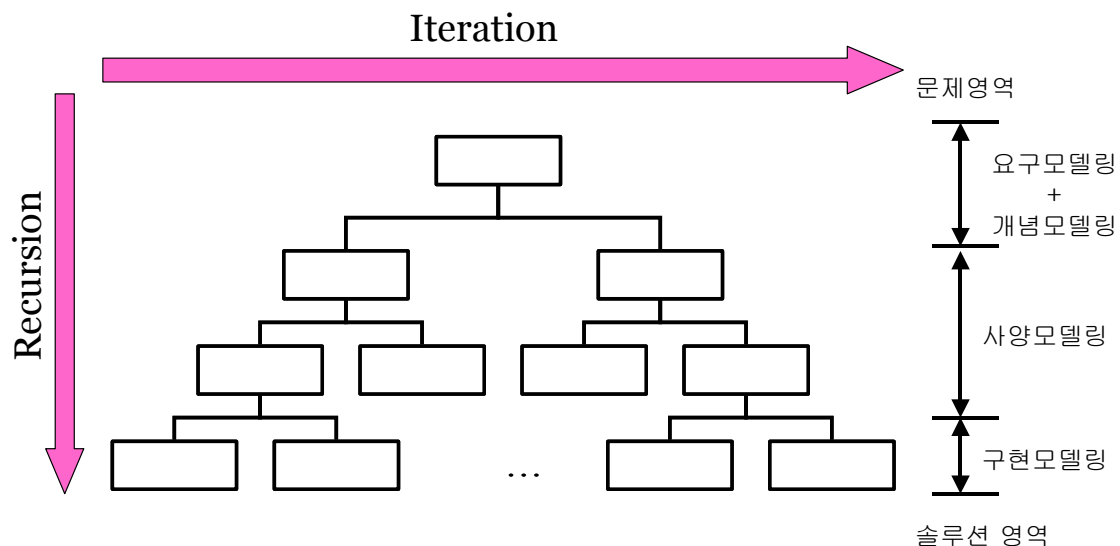


## 제4장 사양모델링

### 4.1 사양모델링

지금까지의 작업을 개념모델링이라고 부른다면, 여기서부터의 작업은 “사양모델링”이라고 부른다. 개념모델링은 개념을 명확히 하는 작업이며, 사양모델링은 사양을 결정하는 작업에 해당한다. 개념모델링까지는, 현재 개발 중인 시스템을 외부에서 살펴보았지만, 지금부터는 내부를 살펴보기로 한다.

사양모델링의 첫 작업은, 개념모델링 과정에서 만든 커다란 한 개의 사각형을 조금 더 작은 객체들로 분해해 가는 것이다. 분해하면서 상세한 부분을 채워가기로 한다. 이와 같은 작업을 반복하는 것이 FUM(Fractal UML Modeling)의 기본적인 사고방식이다.

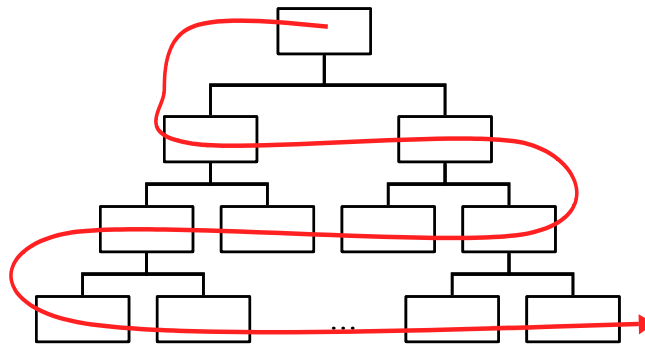


<그림 4-1> Fractal UML Modeling

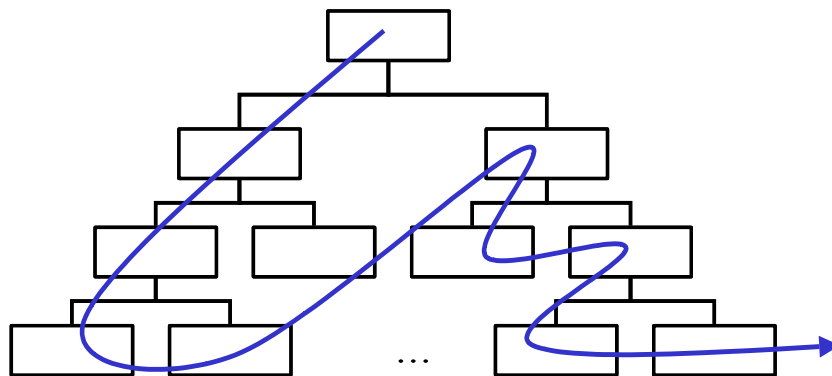
<그림4-1>에서, 점점 레벨을 세분화해 가는 것이 Recursion(세로방향)이며, 같은 레벨의 객체들을 차례대로 작성해 가는 것이 Iteration(가로방향)이다.

현재 단계는, 제일 첫 Recursion이 종료되어 2번째 Recursion에 돌입하려는 단계에 있다. 맨 첫 Recursion은 한 개의 객체(시스템)뿐이었으므로, Iteration은 한번뿐이었다. 그러나, 다음 Recursion부터는 여러 개의 Iteration이 반복될 것이다.

단, 반드시 <그림4-2>와 같은 식으로 작업을 진행하는 것은 아니다(이것은, 오래전부터 사용되었던 단계적 상세화기법과 크게 다르지 않다). 오히려, <그림4-3>과 같은 순서로 작업을 진행하는 경우도 있다.



<그림 4-2>종래의 작업진행방법



<그림 4-3> FUM에서의 작업진행방법

## 4.2 아키텍처

객체를 보다 작은 단위의 객체로 분해해 가는 작업을, “세련화 작업”이라고 부른다. 객체의 세련화는, 결국 복잡한 작업이지만 작업실시후에 얻는 것이 많다.

세련화라는 작업을 수행할 때, 일방적으로 객체를 분해해 가는 것이 아니라, 분해한 객체를 어떻게 조립할 것인가에 대해서도 고려하지 않으면 안된다. 어떤 식으로 객체들로 분해하고, 객체들을 다시 어떻게 조립하는가를 “(소프트웨어) 아키텍처”라고 부른다.

아키텍처란 용어는, 건축물을 지칭하기도 하지만, 구조, 골격 등을 의미하기도 한다. 시스템을 어떠한 아키텍처로 구현할 것인가, 선택의 폭은 다수 존재한다. 또한, 시스템 전체를 대상으로 하는 마크로 아키텍처에서부터 소형 객체 내부의 마이크로 아키텍처에 이르기까지, 아키텍처 자체에도 다양한 레벨이 있다.

양질의 아키텍처 구축 여부는, 시스템의 비기능적 특징에 크게 영향을 받는다고 할 수 있다. “비기능적 특징”이란, 보수용이성, 확장가능성, 이식가능성 등이라고 할 수 있다. 특히, 비기능적 특징은 정량적/표면적으로 계측하기 어려운 면이 있다.

아키텍처구축은 지침서도 없으며, 다양한 경험과 센스에 따라 좌우된다.

## [패턴]

자주 사용되는 아키텍처를 모아서 정리한 것을 “패턴(Pattern)”이라고 부른다<sup>39)</sup>. 자주 사용되는 아키텍처의 카탈로그와 같은 것이다. 이와 같은 패턴을 적용하여 사용하는 것이다. 패턴도 아키텍처 레벨에 대응하며 여러 가지 레벨이 있다. 예를 들면,

Analysis Pattern	요구/개념모델링에서 사용되는 아키텍처
Architecture Pattern	사양모델링에서 사용되는 아키텍처
Design Pattern	구현모델링에서 사용되는 아키텍처
Idiom	코딩에서 사용되는 아키텍처

이밖에도,

Organization/Process Pattern	소프트웨어개발조직 및 개발프로세스의 아키텍처
------------------------------	--------------------------

등과 같은 것도 있다.

흔히 볼 수 있는 아키텍처 패턴의 예로서, “클라이언트/서버”라고 불리우는 패턴이 있다. 클라이언트(고객)가 무엇인가 하고자 하는 것이 있을 때, 스스로 수행하지 않고 서버에게 의뢰하는 아키텍처이다.

이와 같이 아키텍처를 비유 및 메타포에 의해 나타내는 경우도 있다. 여러분들도 자신의 프로젝트에 걸맞는 비유법을 사용하여 시스템의 구조를 표현하는 경우가 자주 있지 않은가? 만약 그와 같은 비유적 표현이 다른 사람에게 있어서 이해하는 데 도움이 되거나, 다시 재사용하게 된다면, “패턴”이라는 형식으로 정리해 두면 도움이 된다.<sup>40)</sup>

패턴은, 최근에 각광받고 있으나, 관련 서적에 기재되어 있는 패턴을 열심히 암기하는 것만으로는 아무런 의미가 없다. 패턴을 사용함과 동시에, 스스로 패턴을 발견해서 기록할 수 있는 능력을 기르는 것이 매우 중요하다.

39) 최근 소프트웨어 산업계에서 주목받고 있는 패턴이라는 개념의 원조는, 건축가 알렉산더에서 시작된다. 소프트웨어는 건축분야로부터 많은 것들을 배우고 전수받고 있다.

40) 패턴 작성방법에는 일정한 형식이 있다. 패턴과 관련된 서적을 살펴보면, 대체적으로,

- 패턴의 이름,
- 어떤 문제가 있는가?
- 어떤 상황에서 문제가 발생하는가?
- 패턴을 적용함으로써 어떤 결과가 얻어지는가?
- 관련된 다른 패턴

등을 고려한다.

### 4.3 객체의 세련화

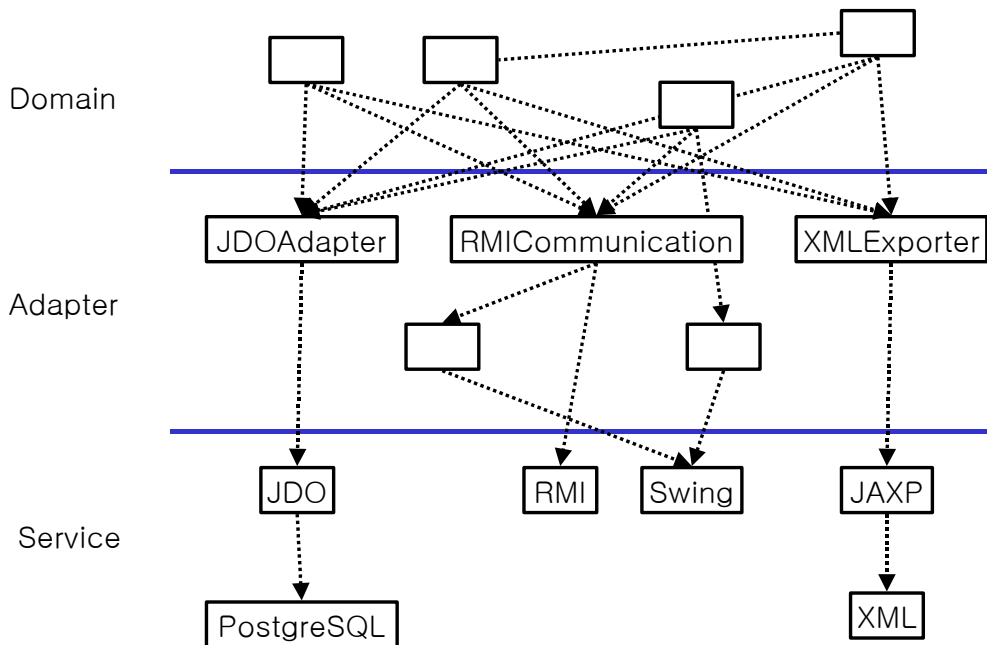
객체의 세련화 작업에 대하여 살펴보기로 하자. 세련화에는 크게 나누어 2가지 방향이 존재한다.

첫 번째는, 수직방향의 세련화, 즉, 계층화이다. 어떤 시스템이더라도 몇 개의 계층으로 구성된다. 현재 구축중인 시스템의 아랫부분에는, 미들웨어 및 클래스 라이브러리 계층이 존재하며, 그 아래 부분에는 운영체제 계층이 존재하게 된다. 여기서는, 다음과 같은 3층 구조를 생각해 보기로 한다.

- 도메인 : 현재 구축중인 시스템 고유의 객체들로 구성되는 계층. 어떤 플랫폼 및 미들웨어를 사용하고 있는지에 대해서는 기본적으로 의존하지 않는다.

- Adaptor : 도메인계층과 서비스계층을 연결해 주는 계층. 대부분의 경우, 스스로 작성하게 된다.

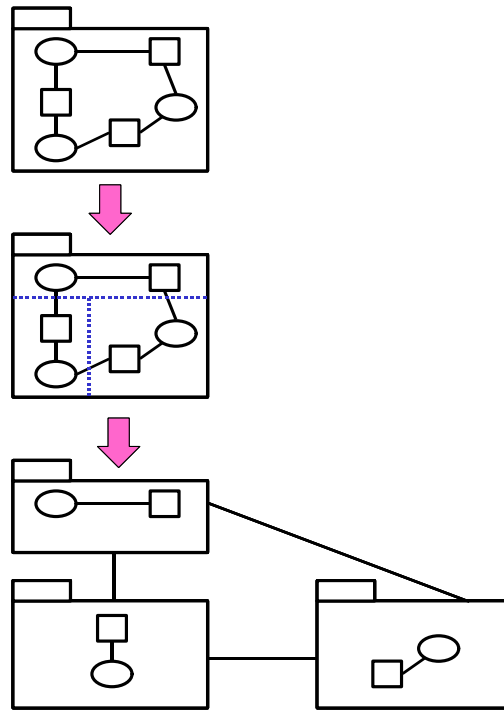
- 서비스 : GUI, DB, 통신 등과 같은 각종 서비스를 구현하는 계층. 미들웨어 및 프레임워크, 라이브러리로서 제공되는 경우도 있으며, 스스로 작성하는 경우도 있다.



<그림 4-4> 수직적인 세련화

도메인 계층에 속하는 객체들은, 개념모델링에서 살펴보았던 객체들을 그대로 세련화한 것들이다. 서비스 계층에 속하는 객체들은, 건축분야의 예를 들어 설명하면, 전기, 가스, 수도, 통신 등의 인프라에 해당하는 부분이다. 어댑터계층은, 수도 및 전기의 배선에 해당하는다. 인프라에 의한 여러 가지 유형들을 어댑터 계층에서 흡수할 수 있다.

또한, 수평적인 세련화를 살펴보자. 개념모델링에서 만든 시스템 객체를 같은 계층 내에서 여러 개의 작은 객체들의 조합으로 변형시켜가는 방법이다.



<그림 4-5> 수평적인 세련화

이 책에서는, 도메인 계층에 초점을 맞추어서 모델링을 진행하도록 하겠다.

#### 4.4 시스템 객체의 세련화

##### [객체의 분해]

우선, 개념모델링에서 작성한 커다란 한 개의 객체, 시스템 객체를 세련해 보도록 하자.

객체를 분해할 때의 기준들 중에서 한가지로서, “결합도는 낮게, 응집도는 높게”가 있다.

결합도란, 분해한 객체들 사이의 의존정도를 나타낸다. 신중하게 분해했음에도 불구하고, 객체들 사이에 복잡한 관계가 존재한다면 무의미해 진다. 결합도는 낮게 하는 것이 좋다.

한편, 응집도는, 한 개의 객체를 구성하는 작은 객체들 사이의 공통성이다. 여러 개의 객체를 모아서 한 개의 객체에 종합할 경우, 공통성이 없이 객체들을 모아놓게 되면 의미가 없다. 응집도는 높게 하는 것이 좋다.

##### [실제로 분해해 보자]

“도서관관리시스템”객체는, 어떠한 객체들로 분해할 수 있을까?

우선은, 이용자와 관련된 부분을 한 개의 덩어리로 생각할 수 있다.

- “이용자”개념객체
- “이용자를 등록한다” Use Case
- “이용자를 삭제한다” Use Case
- “이용자정보를 변경한다” Use Case

이것을 “이용자관리서비스시스템”<sup>41)</sup>으로 부르기로 하자. 그러면, 수장자료와 관련된 부분도 다음과 같이 정리할 수 있다.

- “수장자료” 개념객체
- “책”, “잡지” 등, 서브클래스의 개념객체
- “수장자료를 등록한다” Use Case
- “수장자료를 삭제한다” Use Case
- “수장자료를 검색한다” Use Case

이것들도 마찬가지로, “수장자료관리서비스시스템”이라고 부르기로 한다.

여기서 한 가지 의문이 생긴다. “수장자료를 대출한다”Use Case는 수장자료에 관계하고 있으므로 수장자료관리서비스시스템에 분할하여 추가해야하는가?

“수장자료를 대출한다”Use Case는, “대출”개념객체와 강하게 결합하고 있다. 앞에서 살펴본 “결합도와 응집도”원칙에 따르면, “수장자료를 대출한다”Use Case를 “수장자료관리서비스시스템”에 넣으면, “대출”개념객체, 나머지 Use Case 및 “예약”개념객체도 넣지 않으면 안된다.

“대출”개념객체 및 “예약”개념객체는, “수장자료”개념객체와 마찬가지로, “이용자”와도 관련되어 있다. 그러므로, 결국 모두 한 개의 객체가 되어 버린다.

이렇게 되면 곤란하므로, 여기서 선을 그어두는 것이 좋을 것이다.

나머지는 다음과 같다.

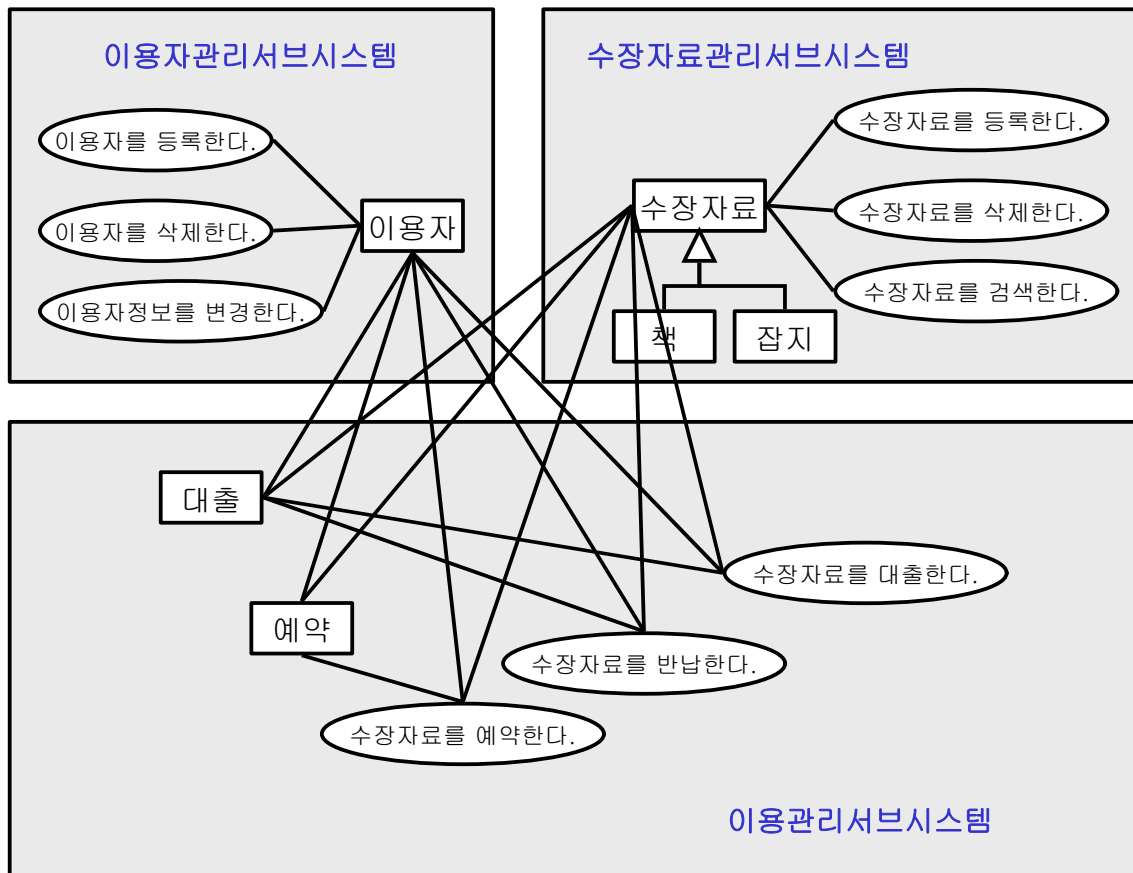
- “대출”개념객체
- “예약”개념객체
- “수장자료를 대출한다” Use Case
- “수장자료를 반납한다” Use Case
- “수장자료를 예약한다” Use Case

위의 모든 것들은 이용과 관계있는 개념객체 및 Use Case들이다. 또한, “이용자관리서비스시스템” 및 “수장자료관리서비스시스템”이 대등하게 관여하고 있다.

전체를 도면으로 나타내면 다음과 같다.

---

41) 여기서, “시스템”을 분할한 것이므로 “서비스시스템”이라고 부르기로 한다. 사람에 따라서는 “컴포넌트” 또는 “모듈”이라는 용어를 사용하는 사람도 있으나, 의미적으로 큰 차이가 없다.



<그림 4-6> 서브시스템으로 분할한 도면

기본적으로는, 서브시스템의 경계를 횡단하는 선의 개수가 적을수록 좋은 세련화라고 할 수 있다. 단, 현실적인 모델링에서는 반드시 횡단선의 개수만이 세련화의 기준이 되는 것은 아니다.

## 4.5 패키지

앞에서 살펴보았던 세련화를, UML에서는 패키지라는 개념을 사용하여 표현할 수도 있다. “패키지(Package)”란, 모델에 출현하는 요소들을 적당한 기준에 의해 모은 것이다. 파일 시스템의 디렉토리 및 폴더와 같은 것이라고 생각해도 좋다. UML 도면에 있어서 패키지를 나타내는 아이콘도, 폴더와 같은 형태를 하고 있다. 모으는 기준은, 어느 정도 명확하게 되어 있다면 어떤 기준을 사용해도 상관없다.

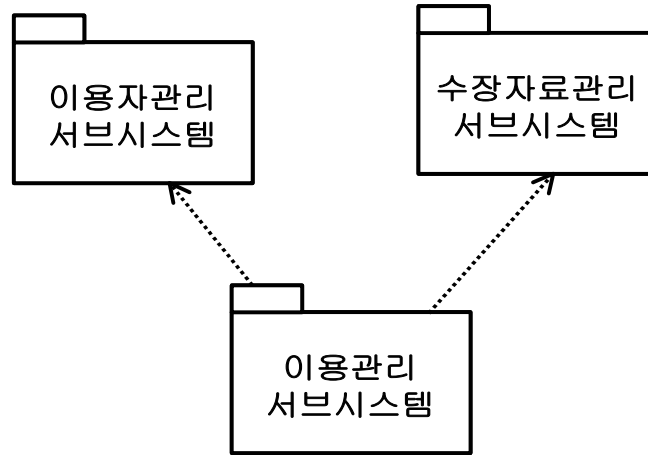
<그림4-7>은 서브시스템들을 패키지로 나타내고 있다.

점선화살표 선은 “의존관계”를 나타낸다. 예를 들면, 이용관리서브시스템은 이용자관리서브시스템에 의존하고 있음을 나타내고 있다. <그림4-6>에서, 서브시스템을 나타내는 사각형들 사이에 걸쳐있었던 선분들을 정리한 것에 해당한다.

화살표의 방향은, 예를 들면, 이용자관리서브시스템은 이용관리서브시스템이 없더라도 운용할 수 있지만, 반대로, 이용관리서브시스템은 이용자관리서브시스템 없이는 운영할 수 없

음을 의미한다.

의존관계에 대하여, 현시점에서는 완전하게 분석할 필요는 없다.



<그림 4-7> 패키지의 예

#### 4.6 Concurrent Engineering<sup>42)</sup>

이와 같이 작성해 보면, 이용관리서브시스템에 앞서서, 이용자관리서브시스템과 수장자료관리서브시스템을 병행해서 개발 가능함을 알 수 있다.

만약, 프로젝트에 충분한 개발인력이 있다면, 2개 팀으로 나누어서 각 팀에 이용자관리서브시스템과 수장자료관리서브시스템을 각각 담당하도록 할 수 있다.

오래전부터 사용되었던 소프트웨어 개발기법에서는, 3개의 서브시스템을 분석하여 그 결과를 토대로 설계하고, 다시 설계결과를 토대로 구현하는 방법을 사용하였다. 그러나, 객체지향 소프트웨어의 표준적인 개발방법은 이와는 다르다. 가능한 곳부터 가능한 만큼 빨리 실제로 동작하는 소프트웨어를 작성하고, 이것에 의해 가능한 만큼 많은 피드백을 얻어서 이후의 개발에 반영시키는 방법으로 진행된다.

도서관관리시스템의 경우, 우선 이용자관리서브시스템과 수장자료관리서브시스템의 분석, 설계, 구현을 먼저 진행한다. 병행 가능한 작업은 병행하여 처리하도록 한다.

단, 이용관리서브시스템을 작성하기 전에, 이용자관리서브시스템과 수장자료관리서브시스템의 사양을 명확히 결정하지 않으면 안된다. 실제로는, 이미 동작하는 서브시스템이 있을 것이다. 이용관리서브시스템은, 이용자관리서브시스템과 수장자료관리서브시스템의 사양을 토대로 만들어지기 때문이다.<sup>43)</sup>

42) Concurrent Engineering이란, 여러 개의 공정을 병행 처리하는 것을 가리킨다.

43) 이때, 이용자관리서브시스템과 수장자료관리서브시스템에도, 틀림없이 약간의 수정 및 기능추가가 필요하게 될 것이다. 이와 같은 수정 및 기능추가는, 오버헤드가 되며, 성과물을 확실하게 구성관리하지 않을 경우 혼란을 야기 시키게 된다. 그러나, 요즘은, 이와 같은 오버헤드보다는, 위와 같은 방법에 의해 얻을 수 있는 잇점이 큰 것으로 여겨진다. 또한, 구성관리도 충분히 적은 비용으로 구현할 수 있게 되고 있다.



우선은, 이용자인터페이스시스템의 사양모델링을 계속해 가기로 하자.

## 4.7 “이용자인터페이스시스템”의 사양모델링

“이용자인터페이스시스템”의 사양모델링을 시작해 보자. 우선, 맨 처음으로 해야 할 것은 이름을 결정하는 것이다.

개념모델링에서는, 가능한 한 국어를 사용하여 모델을 표기하였다. 국어는 가장 자연스러운 언어이며, 현실세계에도 잘 대응하며 사용자 및 고객이 이해하기 쉽기 때문이다.

그러나, 이쯤에서 표기를 영어로 전환해 보자. 반드시 영어로 표기해야만 하는 것은 아니지만, 필자의 습관일 뿐이다.

최종적으로는, 영어를 사용하여 다양한 개념들을 표기하지 않으면 안된다. 왜냐하면, 현재, 한글로 (주석문 이외의) 프로그램을 작성할 수 있는 프로그래밍언어는 거의 없기 때문이다. 예를 들어 표기할 수 있다고 하더라도 읽기 쉽지 않다.

더불어서, 현 시점에서 영어로 전환하게 되면, “현실세계에 있어서 XXX라는 개념”(개념 모델은 현실세계와의 인터페이스를 다루고 있다)과 “소프트웨어 세계의 XXX라는 개념”(사양모델에서는 소프트웨어 세계를 다룬다)들 사이의 미묘한 차이점을 한글과 영어라는 형태로 구별할 수 있는 이점이 있다.

이용자인터페이스시스템은 일단 UserManager라는 이름을 사용하기로 한다. 내부의 내용을 패키지 속에 작성해 가기로 하자.



<그림 4-8>

UserManager

패키지

## 4.8 서비스시스템의 인터페이스

서비스시스템을 세련화하는 방법은, 개념모델링에서 시스템을 세련화하는 방법과 기본적으로 같다.

개념모델링에서 맨 처음 생각한 것은 Actor, 즉 시스템과 상호수수작용하는 외부요소들이었다. 현재 Actor에 대응하는 것은, 다른 2개의 서비스시스템이 된다. Actor의 Workflow는, 개념모델링에서 생각했던 개념시나리오와 유사하다.

다음으로 살펴보았던 것은 Use Case, 즉 시스템과 Actor의 상호수수작용이었다. Use Case란, “시스템이 Actor에게 어떠한 서비스(기능)를 제공하는가”였다. 서브시스템의 경우도 마찬가지로, “서브시스템이 다른 서브시스템에게 어떠한 서비스를 제공하는가”를 살펴보기로 하자. 단, 개념모델링과는 달리, 사양모델링에서는 “서비스”를 Use Case로서가 아니라, 더욱 일반적인 객체의 용어인 “동작”으로 생각한다.

동작이란, 서브시스템이 수신할 수 있는 메시지를 가리킨다. 동작의 내부내용, 즉, 메시지에 반응하여 무엇을 수행하는가에 대해서는 현시점에서는 아직 생각하지 않는다. 메시지를 수신할 수 있는지 여부만이 중요하기 때문이다.

어떤 서브시스템이 수신할 수 있는 메시지들을 모아놓은 것을 “인터페이스”라고 부른다. 문자 그대로, 해당 서브시스템과 다른 서브시스템과의 접속점이 된다.

동작을 생각할 때에 필요한 것으로서,

- 동작의 이름
- 동작의 매개변수와 필요하다면 각 매개변수의 데이터형
- 필요한 경우, 동작의 반환값과 반환값의 데이터형
- 필요한 경우, 동작이 일으키는 예외처리

등이 있다.<sup>44)</sup>

인터페이스는 동작을 모아서 정리한 것이므로, 서브시스템이 복수개의 인터페이스를 갖는 경우도 있다. 어떤 사항에 관한 동작을 정리하여 1개의 인터페이스로, 그리고 다른 사항에 관한 동작들을 모아서 별개의 인터페이스로 만드는 것이 이해하기 용이하게 되는 경우가 있기 때문이다.

## 4.9 UserManager의 인터페이스

### [동작명]

UserManager서브시스템의 인터페이스를 살펴보자. 서브시스템이 제공하는 서비스는, 서브시스템에 포함되는 Use Case에 거의 상응한다. 단, Use Case는 어디까지나 “시스템”이 Actor에게 제공하는 서비스이므로, 서브시스템이 다른 서브시스템에게 제공하는 서비스는 약간 차이점이 있거나, 여분으로 필요한 것이 있을지도 모른다.

우선, “이용자를 등록한다”는 것이 불가능하게 되면 어떤 일도 시작할 수 없게 된다.

이름은 무엇으로 할까? “이용자”는 서브시스템의 이름에 맞추어서 “user”라고 하자. “등록한다”는 “register”라고 해도 좋겠지만, 가능한 한 단순하고 범용적인 이름을 사용하도록 한다. “등록한다”는, 이용자를 신규추가하는 것이므로 “create”라고 명명하기로 한다<sup>45)</sup>.

이름은, “XXX를 YYY한다”와 같은 형식으로 하면 대개 분별할 수 있을 것이다.

44) 이것은 프로그래밍언어에서, 함수 및 메소드의 “시그니처(Signature)”와 같은 의미를 갖는다.

45) 마찬가지로, 삭제는 delete, 변경은 update, 검색은 retrieve 등으로 결정해 두면, 고민하지 않아도 되고 통일성도 갖추게 된다.

```
이름 createUser46)           // 이용자를 등록한다.
```

### [매개변수]

다음으로 살펴볼 것은 매개변수이다. 모든 매개변수를 여기서 결정해야 하는 것은 아니다. 개념모델링을 할 때 보다는 명확하게, “어떤 정보가 중요한가”를 결정하면 된다.

예를 들면, “이름”은 아무튼 필요하게 된다. 데이터형은 문자열형이 될까? 성과 이름을 분리해야 하는가? 이것에 대해서는 현시점에서는 고려하지 않기로 한다. 그러나, 현시점에서 반드시 명확히 해야 한다면, 가장 범용성이 높은 문자열을 사용하거나, “이름”이라는 데이터형을 나중에 적당히 만들어서 사용하도록 하고 일단 “이름”형으로 해 둔다.<sup>47)</sup>

연락처에 대해서도 마찬가지이다.

```
이름 createUser           // 이용자를 등록한다.
매개변수 name             // 이용자의 이름
매개변수 contact          // 이용자의 연락처
```

### [반환값]

다음은 반환값에 대하여 살펴보자. 객체를 생성하는 경우의 반환값은 생성된 객체가 되는 것이 일반적이다. 이와 같은 경우에는, “이용자”개념객체에 상응하는 객체가 되므로 User라는 데이터형으로 지정해 두기로 한다.

```
이름 createUser           // 이용자를 등록한다.
매개변수 name             // 이용자의 이름
매개변수 contact          // 이용자의 연락처
반환값 User               // 등록된 이용자
```

예외에 대해서도 반환값과 마찬가지로 너무 상세한 부분까지 결정할 필요는 없지만, 필요하다면 대략적으로 결정해 두기로 한다. 생성에 대한 예외는, “생성에 실패했다” 정도가 되므로, 여기서는 특별히 고려할 필요는 없다.

### [기타 동작]

---

46) 여기서는, 여러 개의 단어로 이름을 짓는 경우, 단어의 선두문자를 대문자로 하여 연결하는 방법을 사용한다. 약어의 경우(예를 들면, IO)에는, 관습에 따라서 전부 대문자를 사용한다. 선두를 대문자로 할 것인지 소문자로 할 것인지는, 이름이 광역적인 경우에는 대문자를, 지역적인 경우에는 소문자를 사용한다. 예를들면, 동작명은 소문자로 시작한다. 구현에 사용하는 프로그래밍언어가 정해지면, 해당 언어의 규약에 따르도록 해도 좋다.

47) 소프트웨어개발에서는, “결단을 가능한 한 뒤로 미루도록 한다”는 것은, (상식에 반하는 것일지도 모르지만) 실제로는 중요한 개념이다. 너무 빨리 결단을 내리는 것은, 막대한 비용과 위험을 요구하게 된다. 데이터형을 제공하지 않는 프로그래밍언어를 사람들이 선호하는 이유가 바로 이런 점 때문이다.

이용자를 등록하는 동작(서비스), createUser에 대하여 살펴보면, 이번에는 정말로 이용자가 등록됨을 확인하는 동작도 살펴보아야 할 필요가 있다. 이것을 “관측가능성”이라고 하며, 확인을 위한 동작을 “Observer”라고 부르기도 한다. Observer가 필요한지 여부는 경우에 따라 존재여부가 결정되므로, 그때그때 생각해 보기로 한다.

현시점에서는, 예를 들어 다음과 같은 동작들을 살펴보면 이용자가 실제로 등록되었는지 여부를 확인할 수 있다.

```
이름  hasUser      // 사용자등록을 확인한다.
                        (createUser의 Observer)
매개변수 id        // 이용자의 식별자
반환값 Boolean     // id를 갖는 이용자가 등록되어 있으면 true, 아니면 false
```

hasUser가 있으면, 이용자가 실제로 등록되었음을 확인할 수 있지만, 내부내용까지 올바른지 여부는 알 수 없다. 이를 확인하기 위해서는 또 다른 동작이 필요하게 된다.

```
이름  retrieveUser  // 특정한 사용자를 얻어낸다.
                        (createUser의 Observer)
매개변수 id        // 이용자의 식별자
반환값 User        // id를 갖는 이용자. 만약 등록되어 있지 않을 경우에는 null
```

retrieveUser가 있으면, 등록된 이용자의 이름 및 연락처가 지정되어 있는 데로 되어있는지 여부를 알 수 있다.

## [역동작]

다음으로 동작을 고려할 때 힌트가 되는 것이 “역동작”이다. 위의 예에서는, “등록하다”의 역동작으로서, “등록을 취소한다”를 생각할 수 있다. 물론, 모든 조작들에 대하여 역조작이 만들어져야하는 것은 아니며, 여기서는 필요한 경우에만 생각해 보기로 한다.

```
이름  deleteUser   // 이용자의 등록을 삭제한다(createUser의 역동작)
매개변수 user : User // 삭제될 이용자
```

매개변수는 user보다는 id로 해야 한다고 생각하는 사람도 있을 것이다. 물론, 여러 가지 유형의 조작을 생각하는 것도 좋지만, 일단 여기서는 가능한 한 일반적인 형태의 조작을 생각해 보기로 한다.

역동작에 관하여 흥미 있는 점은, 동작과 역동작을 연결했을 때 원래대로 되지 않으면 안 된다는 점이다. 예를 들면, 위의 예에서

`deleteUser(createUser(XXX, YYY))`

를 실행했을 때, 등록되어 있는 이용자전체는 이와 같은 조작을 수행하기 전과 동일해야만 한다.

지금까지 살펴보았던 것 이외의 서비스로서 무엇이 필요할 것인가? Use Case에는 명확하게 “이용자정보를 변경한다”가 있었다.

```
이름 updateUser      // 이용자의 정보를 갱신한다.
매개변수 id          // 갱신될 이용자의 식별자
매개변수 name        // 이용자의 이름
매개변수 contact     // 이용자의 연락처
반환값 User          // 갱신된 이용자
```

반환값은 없어도 상관없다. updateUser가 정확하게 수행되었는지 확인할 수 있는지도 점검해 두도록 하자. 이와 같은 경우에는 retrieveUser가 정의되어있다면 문제없을 것이다.

<문제> 수장자료관리서브시스템의 인터페이스를 작성하시오. 여기서 수장자료관리서브시스템을 StuffManager라고 부르기로 한다.

<해답>

```
이름 createStuff     //수장자료를 등록한다.
매개변수 ...
```

그러나, 이 단계에서 몇 가지 문제점이 있다. 이용자는 특별히 종류가 있는 것이 아니지만, 수장자료의 경우에는 책, 잡지, CD, DVD 등 여러 가지 종류가 있다<sup>48)</sup>. 또한, 이와 같은 종류는 앞으로도 계속 늘어날 가능성이 있다. 이와 같은 문제점을 어떻게 해결해야 할까?

해결방법으로서는, 다음과 같은 여러 가지가 있다.

1. createBook, createMagazine, ... 등과 같은 인터페이스를 많이 준비해 둔다.

그러나, 수장자료의 종류가 늘어날때마다 수장자료관리서브시스템의 인터페이스를 변경해야 되므로 별로 바람직하지 않은 방법이다. 수장자료관리서브시스템의 책임은, “수장자료를 관리하는”것이므로, 어떠한 종류의 수장자료가 있을지는 관계가 없다.

2. 수장자료객체를 만드는 책임을 분리하여 새로운 클래스(예를들면, StuffFactory)를 만든다. createStuff가 수행하는 작업은 새로운 수장자료를 등록하는 것으로만 한정시킨다<sup>49)</sup>.

48) “여러 가지 종류”를 객체지향적 관점에서는 “서브클래스(SubClass)”라고 표현한다. “여러 가지 종류”의 토대가 되는 것(위의 경우에는, 책, 잡지, CD, DVD, ...에 대하여 “수장자료”)을 “추상 슈퍼클래스(Abstract SuperClass)”라고 부른다. 책, 잡지, CD, DVD, ...들 중에서 어느 것도 아닌 “수장자료”는 실제로 존재하지 않으므로 “추상”이라는 용어가 붙게 되는 것이다.

위의 2번 방법이 더 바람직해 보인다. 이것이 바로 “Factory 패턴”이라고 부르는 것으로서, 자주 사용되는 방법이다. Factory패턴을 사용할 경우, 다음과 같이 된다.

```
이름 createStuff      // 수장자료를 등록한다.
매개변수 stuff : Stuff // 수장자료
반환값 Stuff          // 등록된 수장자료
```

한편, StuffFactory클래스는 한 가지 동작만을 갖는다.

```
이름 createStuff      // 수장자료를 작성한다.
매개변수 kind         // 수장자료의 종류
반환값 Stuff          // 작성된 수장자료
```

kind에는, 예를 들면 문자열로서 “Book” 또는 “CD” 등을 지정한다. 이와 같이 하므로써, StuffFactory는 적당한 서브클래스를 찾아내어, 상응하는 인터페이스를 만들어서 반환하게 된다. 어떠한 종류의 서브클래스에 대하여 어떤 인터페이스를 어떻게 만들 것인가와 같은 “지식”은 StuffFactory클래스 내부에 은폐시킬 수 있게 된다.

그 밖의 동작은 이용자관리서비스시스템의 경우와 거의 같다.

```
이름 hasStuff         // 수장자료등록을 확인한다.
매개변수 id           // 수장자료의 식별자
반환값 Boolean        // id를 갖는 수장자료가 등록되어 있으면 true, 아니면 false
```

```
이름 retrieveStuff    // 특정한 수장자료를 얻어낸다.
매개변수 id           // 수장자료의 식별자
반환값 Stuff          // id를 갖는 수장자료
```

```
이름 deleteStuff      // 수장자료의 등록을 제거한다.
매개변수 stuff : Stuff // 제거될 수장자료
```

updateUser에 대응하는 updateStuff는 존재하지 않는다. 왜냐하면, 수장자료의 내부내용, 즉, 기재항목이 변경되는 일은 없기 때문이다.

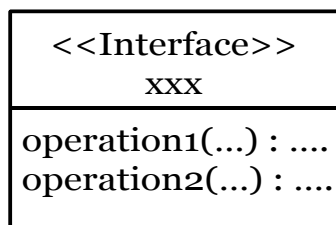
## 4.10 UserManager의 인터페이스(2)

[UML에 있어서 인터페이스]

UML에서는, 인터페이스를 클래스와 거의 같은 방식으로 표현한다. 상이한 점으로써는, 속성을 나타내는 영역이 없다는 점과, 이름을 기입하는 영역 내에 <<Interface>>라는 표식

49) 이와 같은 경우에는 createStuff라는 동작명이 어울리지 않을지도 모른다.

을 붙인다는 점이다. 여기서 « 과 »<sup>50)</sup>으로 둘러 쌓여있는 이름을 “스테레오타입 (Stereotype)” 이라고 부른다.



<그림 4-9> 인터페이스

스테레오타입이란, “비슷하지만 조금 다른 것”을 나타내는 표시이다. “클래스와 비슷하지만 인터페이스이다”라는 것을 나타내고 있다. “<<Interface>>” 대신에 아이콘으로 ○을 사용해도 상관없다.

동작은,

**동작명(매개변수, ...) : 반환값의 데이터형**

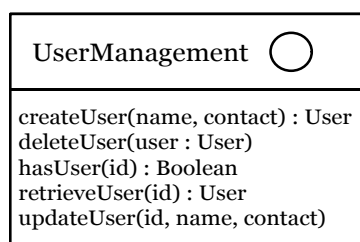
과 같이 표기한다. 반환값이 없는 경우, 또는 아직 고려하지 않고 있는 경우에는, “:” 표기 이후를 생략해도 좋다.

매개변수도 마찬가지로,

**매개변수명 : 데이터형**

과 같이 표기한다. 여기서도 “:”이후를 생략해도 좋다.

지금까지 살펴보았던 인터페이스를, UserManagement라는 이름을 붙여서 도면으로 나타내보도록 하자. UserManagement란, 이용자인터페이스에 관한 서비스들을 모아놓은 인터페이스라는 의미이다.

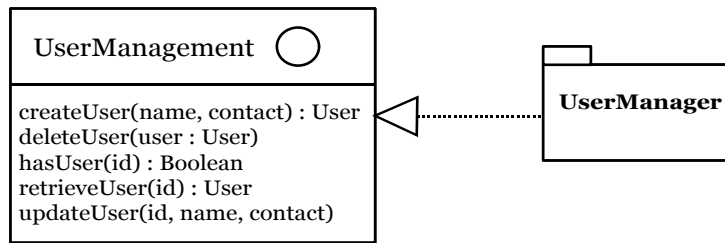


<그림 4-10>

UserManagement  
인터페이스

UserManager서비스시스템이, 인터페이스 UserManagement를 제공하고 있는 사실, 즉, UserManagement의 각 동작에 대응하는 메시지를 전송하면 올바르게 응답하는 것을 UML에서는 다음과 같이 표현한다.

50) « »는 부등호 기호를 나열하여 표기하는 경우도 있지만, 실제로는 1개 문자로서 “guillemet”라고 부른다.

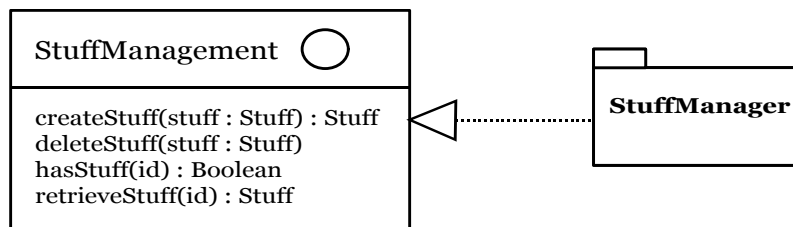


<그림 4-11> UserManager서브시스템이, 인터페이스 UserManagement를 제공한다.

점선으로 된 흰색 화살표는, “구현관계”를 나타낸다. 즉, “UserManager는 UserManagement 인터페이스를 구현하고 있다”는 것을 나타낸다.

<문제> 수장자료관리서브시스템과 해당 인터페이스를 UML을 사용하여 도면으로 작성하시오.

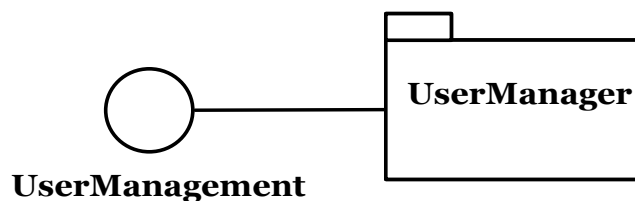
<해답>



<그림 4-12> 수장자료관리서브시스템과 해당 인터페이스

### [막대 사탕]

인터페이스 UserManagement의 내용을 알고 있는 경우, 다음과 같이 표기하는 경우도 있다. 이와 같은 표현방법을 “막대사탕”이라고 부르기도 한다. 막대사탕처럼 보이지 않는가??



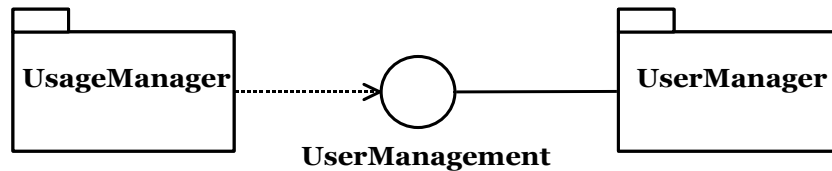
<그림 4-13> 막대사탕으로 표현한 인터페이스

수작업으로 화이트보드에 모델링 하는 경우에는 위의 막대사탕 표기법이 보다 작성하기 수월할 수도 있다.

이에 대하여, 예를 들면 이용관리서브시스템이 UserManagement 인터페이스를 사용하는



쪽이라고 한다면, 다음과 같이 표현된다.



<그림 4-14> UsageManager(이용관리) 서브시스템이  
UserManagement 인터페이스를 사용한다.

이것으로서, 어떤 인터페이스를 매개로 하여 서비스를 제공하는 쪽 및 서비스를 이용하는 쪽을 알기 쉽게 표현할 수 있다. 앞에서도 설명한 바와 같이, 점선화살표시는 “의존관계”를 나타낸다.

#### 4.11 서브시스템의 계약

인터페이스가 올바르게 결정되기만 하면 서브시스템이 어떤 서비스를 제공해 줄 것인가를 파악할 수 있게 된다.

서브시스템은, 반드시 별개의 서브시스템 등에서 이용된다. 즉, 서브시스템을 만든 사람 외에 다른 사람들도 해당 서브시스템이 무엇을 수행하는지를 명확하게 이해하지 않으면 곤란해진다. 인터페이스를 정의하는 것은 이에 대한 해결책들 중에 하나에 해당한다.

그러나, 인터페이스만으로는 실제로 정보가 조금 부족하다. 서브시스템이 인터페이스를 통해서 실제로 무엇을 해 줄 것인가는, 동작의 이름과 매개변수 등을 살펴보고 추측할 수 있는 정보밖에 안된다. 더욱이, 이러한 추측에는 오류가 포함될지도 모른다. 물론, 한글 및 영어로 주석문을 붙일 수도 있겠지만, 주석문을 신뢰할 수 없는 경우도 있다.

따라서, “계약”이 필요하다. 계약이란, 동작마다 “이러이러한 조건이 갖추어졌을 때에 한해서, 해당 동작이 올바르게 작동한다”, “동작이 올바르게 작동한다면, 이러이러한 조건이 성립했다는 것을 의미한다”와 같은 약속을 서비스를 제공하는 쪽과 서비스를 이용하는 쪽 사이에 연결시키는 것이다.

센스가 둔한 독자라면 알아차리지 못했을지도 모르겠지만, 이와 같은 계약에 관해서는 앞서 설명한 Use Case를 표현할 때 사용했던 사전조건/사후조건에 해당한다<sup>51)</sup>.

즉, 사전조건/사후조건과 마찬가지로, 서브시스템의 인터페이스를 명확하게 표현함에 있어서, 계약을 이용할 수 있다. 단, Use Case의 경우에는 자연언어를 사용하여 적당히 표현했지만, 여기서는 더욱 엄밀하게 작성해야 한다.

계약의 작성방법에는 몇 가지 종류가 있다. 첫 번째 방법으로서, Java와 같이 구현할 때

51) 계약에는, 여기에 소개한 것 이외에도 “불변조건”이라는 것이 있다. 불변조건은 각 동작마다 정의되는 것이 아니라, 각 서브시스템마다 정의하며, 어떤 동작을 작동시키기 전과 후에 모두 성립하는 조건을 의미한다.

사용하는 프로그래밍언어로 작성하는 방법이다. 이 방법을 사용할 경우, 새로운 언어를 학습할 필요가 없으므로 편리하지만, 알고리즘을 작성하기 위해 만들어진 언어를 사용하여 조건을 표현하기에는 어려움이 많다.

또 다른 방법으로써, 조건을 기술하기 위해 만들어진 언어를 사용하는 방법이 있다. UML에서는 OCL(Object Constraint Language, 객체제약언어)라는 언어가 정의되어 있다. 새로운 언어를 익히는 것은 다소 시간과 노력이 필요하지만(OCL자체는 그다지 복잡한 언어가 아니다), Java 등으로 작성하는 것 보다는 훨씬 간단하게 표현할 수 있다<sup>52)</sup>.

## 4.12 UserManager의 계약

### [createUser의 사전조건/사후조건]

계약에 관하여, 사전조건/사후조건은 각 동작별로, 불변조건은 각 서브시스템 별로 파악한다. 추출 완료된 Use Case의 사전조건/사후조건을 참고로 하여, createUser동작의 조건을 살펴보기로 하자.

동작“createUser(name, contact)”의 사전조건/사후조건을 기술하기 전에, 이에 대응하는 Use Case”이용자를 등록한다“의 사전조건/사후조건을 다시 한번 살펴보자.

사전조건은 다음과 같다.

1. 이용자는 아직 등록되어 있지 않다.
2. 이용자는 이용자가 되는 자격을 갖추고 있다.

위의 1과 2는 Use Case의 사전조건이므로, “현실세계”내에서만 성립된다. 현재, 고려하고 있는 것은 “객체의 세계”이므로, 현실세계로부터 객체의 세계로 용어를 전환시켜야 된다.

1의 조건은, 이용자가 지금까지 등록되었던 사람인지 여부와 관련 있다. 객체의 세계에서는 어떻게 표현하는 것이 좋을까? 이 문제는 상당히 난해하다. 이름 및 연락처가 동일한 경우는 있을 수 없을 것이다. 지문감정 및 DNA조사를 할 수도 없다.

2의 조건은, “A시에 살고 있거나 근무하고 있다”는 것이 이용자 등록 조건이었다. 즉, 2의 조건을 체크하기 위해서는, 주소 또는 근무지를 파악할 필요가 있다는 것이다. 그러나, 최종적으로 체크하는 것은 등록하는 사서직원이지, “도서관관리시스템”이 아닐 것이다.

한편, 사후조건은 다음과 같다.

1. 이용자는 등록되어 있다.

이것은 실제로 이용자가 등록되었는지 여부를 확인해 보면 될 것이다. 이를 위하여, createUser()를 실행한 결과, 새롭게 만들어진 이용자 객체가 반환되어 오도록 하자. 이러

---

52) 현재, OCL을 직접 해석해서, 사전조건/사후조건이 만족되는지를 체크해 주는 도구는 거의 없다. 그러나, 아무튼 궁극적으로는, OCL을 Java 등으로 변환시킬 필요가 있다.

한 기능을 사용하여, 이용자가 정말로 등록되어 있는지 여부를 확인할 수 있게 된다.

편의상, “등록되어 있는 모든 이용자”를 allUsers로 나타내기로 하자. UML에서 정의하고 있는 OCL의 문법에 맞추어 작성해 보면 다음과 같다.

```
context UserManager::createUser(name, contact)
post:
allUsers = allUsers@pre->including(result) and
name = result.name and contact = result.contact
```

"context"는, 사후조건의 성립대상을 나타내고 있다. "post:"는 사후조건임을 나타내고 있으며, 반대로, "pre:"는 사전조건임을 나타낸다.

“result”는, 위 동작의 반환값, 즉, 등록되었을 이용자가 된다.

“allUsers@pre”는, 위 동작이 호출되기 전의 allUsers를 나타내고 있으며, 이것은 사후조건에만 사용할 수 있다.

“->including(result)”는, 집합에 대한 조작으로서, "...에 result를 추가한 집합“을 나타낸다.

즉, 위의 작성 예에서, 전반부는 “이용자를 등록하면 등록하기 전의 전체 이용자에, 새롭게 등록한 이용자를 추가한 것이 새로운 전체이용자가 된다”라는 의미가 된다.

#### [deleteUser의 사전조건/사후조건]

deleteUser의 사전조건/사후조건도 살펴보자. “이용자를 삭제한다”Use Case의 사전조건/사후조건은 다음과 같다.

사전조건:

1. 이용자가 등록되어 있다.

사후조건:

1. 이용자가 등록되어 있지 않다.

deleteUser의 사전조건/사후조건을 OCL로 작성하면 다음과 같다.

```
context UserManager::deleteUser(user : User)
pre:
allUsers->includes(user)
post:
allUsers = allUsers@pre->excluding(user)
```

기타, 다른 서비스에 대한 계약도 살펴보기로 하자.

```
context UserManager::retrieveUser(id)
post:
allUsers->includes(result) and result.id = id
```

```
context UserManager::hasUser(id)
post:
result = allUsers->includes(retrieveUser(id))
```

```
context UserManager::updateUser(id, name, contact)
post:
if not hasUser(id) then
  allUsers = allUsers@pre
else
  let updated : User = retrieveUser(id) in
  allUsers->excluding(updated) =
    allUsers@pre->excluding(updated) and
  name = updated.name and contact = updated.contact
end
```

update와 관련된 부분은 다소 복잡하다. “if then else end”는, 프로그래밍언어와 거의 같다. “let XXX in YYY”는, XXX를 YYY의 지역변수로 정의한다는 의미를 갖는다.

<문제> 수장자료관리서비스시스템의 인터페이스에 대한 계약을 OCL로 작성하시오.

<해답>

```
context StuffManager::createStuff(stuff : Stuff)
pre:
allStuffs->excludes(stuff)
post:
allStuffs = allStuffs@pre->including(result)
and stuff = result
```

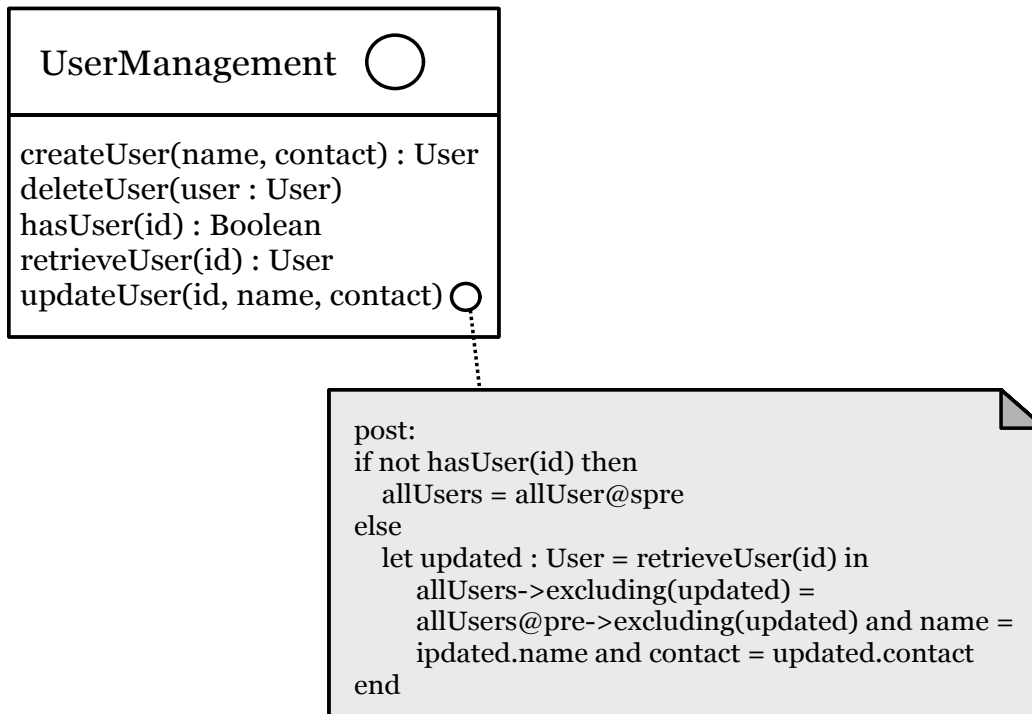
```
context StuffManager::deleteStuff(stuff : Stuff)
pre:
allStuffs->includes(stuff)
post:
allStuffs = allStuffs@pre->excluding(result)
```

```
context StuffManager::retrieveStuff(id)
post:
allStuffs->includes(result) and result.id = id
```

```
context StuffManager::hasStuff(id)
result = allStuffs->includes(retrieveStuff(id))
```

## [UML 도면에 계약을 기입한다]

다음과 같이 작성된 계약을 UML의 도면에 기입한다<sup>53)</sup>.



<그림 4-15> UML을 사용한 계약의 표현 예

수작업에 의해 화이트보드를 사용하는 경우, 계약은 별도로 어딘가에 기록하도록 한다.

## 4.13 서브시스템의 내부구조

서브시스템의 인터페이스와 계약을 결정함으로써, 서브시스템의 외부가 완성되었다. 이번에는 서브시스템의 내부를 살펴보기로 하자. 개념모델링의 경우, Use Case View를 살펴본 후에 개념객체들을 살펴보았다. 즉, 이 부분은 시스템의 내부구조를 살펴본 것이다.

서브시스템의 내부구조는 어떻게 살펴보면 될까? 이에 대한 해답은, 서브시스템을 구성하고 있는 객체는 무엇인지를 조사하는 것과 거의 같은 작업으로 진행된다.

대부분의 경우, 서브시스템을 구성하고 있는 객체들의 대부분은, 서브시스템의 인터페이스를 검토할 때 발견했을 것이다. 현 시점에서 발견해 낸 서브시스템의 구성요소는, 주로 다른 서브시스템에서 사용되는 객체들인 경우가 많다.

이와 같은 구성요소들을 “사양요소”라고 부르는 경우도 있다. 사양요소들은 가능한 한 클래스 형태가 아닌 인터페이스 형태로 작성하는 것이 나중에 편리하다<sup>54)</sup>.

53) 우측 끝이 안쪽으로 굽혀진 사각형을 사용하여 각종 주석을 기입한다. 이와 같은 사각형을 Note라고 부른다. Note에는 무엇을 기입해도 좋으며, 보충설명 등에 사용할 수 있다. 모델링에는 영향을 주지 않는다. 위와 같이 점선으로 UML 도면의 요소들과 연결시킬 수도 있다.

54) 클래스와 인터페이스의 차이점은, 객체지향언어의 경우와 마찬가지로이다. 인터페이스는 “어떤 동작이 있는지”

한편, 다른 서브시스템과는 거의 관계가 없지만, 대상이 되는 서브시스템을 기동시키기 위해서 필요한 구성요소들은, 지금부터 서브시스템 내부의 가장 깊은 곳으로 가면서 밝혀질 것이다.

이와 같은 구성요소들을 “구현요소”라고 부른다. 구현요소들은, 외부에 보이기 위한 것이 아니므로 클래스 형태로 작성해도 좋다.

#### 4.14 UserManager의 내부구조/사양요소

UserManager 서브시스템을 구성하는 최대 구성요소는, 역시 “이용자”개념객체에 대응하는 것이다. 이용자가 이용자관리서브시스템의 대상이기 때문이다. UserManager서브시스템의 인터페이스에서도, 이와 같은 정보를 얻어낼 수 있었으며, 잠정적으로 “User”객체라고 명명하였으므로, 그대로 사용하기로 하자.

User객체는, 도서관 이용자를 나타내는 객체이다. “이용자”개념객체가 “진화”한 것이라고 생각해도 좋다.

이와 같이 “진화”하기 전의 요소와 “진화”한 후의 요소 사이의 관계는, 개념객체와 서브시스템의 구성요소 이외에도 여러 가지가 있다. 예를 들면, Use Case로부터, 서브시스템의 인터페이스로 “진화”한 예를 앞서 살펴보았다. 이와 같이 “진화전”의 것과 “진화후”의 것 사이의 대응관계를 “추적가능성”이라고 부른다. 진화 과정을 추적할 수 있다는 의미이다.

User객체는 다른 서브시스템과의 사이의 상호 수수작용시에도 교환된다. 따라서, 우선은 사양요소로서 생각하도록 한다. 즉, 인터페이스부터 정의하기로 한다. 이를 위해서는, “이용자”개념객체의 책임과 동작을 상기해 볼 필요가 있다. 단, 이번에는 이용자관리서브시스템에 관한 책임만을 대상으로 하고 있으므로, “책을 대출한다” 등은 고려하지 않는다.

“이용자”개념객체의 책임들 중에서, 이용자관리에 관계있는 것으로서 다음의 2가지를 손꼽을 수 있다.

1. 이름, 연락처 등의 기본정보를 인지하고 있다.
2. 기본정보에 변경이 발생하면 알려준다.

이름을 인지하고 있다는 등의 지식책임은, 다음과 같은 동작으로 변환시켜서 생각한다.

```
이름 getName          // 이름을 인지한다.  
반환값                // 이름
```

“XXX에 대하여 인지하고 있다”에 대응하는 동작의 이름을 getName이라는 동작에 대응시키도록 한다. 반환값의 데이터형은 그대로 두기로 한다. 연락처에 대해서도 동일하게 적용하면 다음과 같다.

```
이름 getContact       // 연락처를 인지한다.
```

---

만을 결정하지만, 클래스는 “해당 동작을 어떻게 구현하는가”와 “구현하기 위해서는 어떤 데이터를 어떻게 보유하는가”까지를 결정해야 한다. 이러한 경우에도 “결단은 가능한 한 나중에 미룬다”라는 원칙을 적용한다.

반환값                      // 연락처

또한, “기본정보에 변경이 발생하면 알려준다”에 대하여, 사양모델링 세계에서는 “이름 등을 변경한다”로 변환시키도록 한다.

이름 setName              // 이름을 변경한다.  
매개변수 name            // 이름

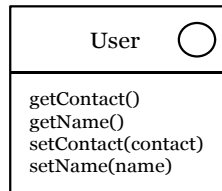
“이름을 인지한다”가 getName이었으므로, “이름을 변경한다”는 setName으로 한다. 연락처도 마찬가지로,

이름 setContact          // 연락처를 변경한다.  
매개변수 contact        // 연락처

와같이 된다.

이와 같이 getXXX 및 setXXX와 같은 동작은 객체가 갖고 있는 정보에 액세스하기 위한 동작들로서, “Accessor”라고 부르기도 한다. 특히, getXXX를 “getter”, setXXX를 “setter”라고 부른다.

서브시스템의 인터페이스와 마찬가지로 방법으로, UML을 사용하여 표기하면 다음과 같다.

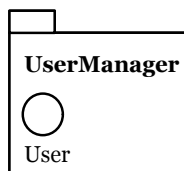


<그림 4-16>  
User의 동작

User의 동작에는 Accessor들만 있으므로, 현 단계에서는 User가 단순히 데이터구조임을 알 수 있다.

#### [객체의 세련화]

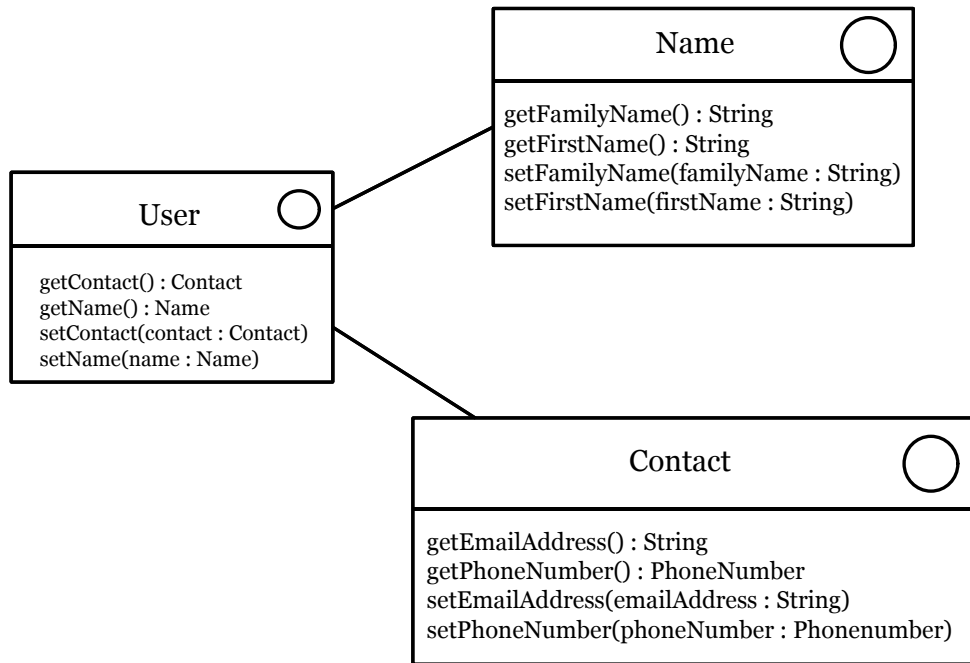
User는 UserManager의 구성요소라는 사실을 UML을 사용하여 표현하면 다음과 같다.



<그림 4-17>  
User는  
UserManager  
의 구성요소

위 그림에서 알 수 있듯이, User를 인터페이스 아이콘만을 사용하여 표현하였으나, <그림 4-16>과 같이 동작들 전체를 포함하여 표기하여도 좋다.

또한, 다음과 같이 User를 더 세련화하여 Name 및 Contact와 같은 객체들로 분해할 수도 있다.



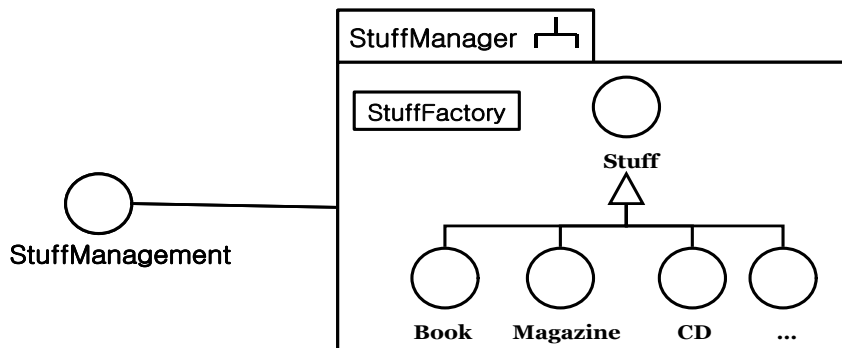
<그림 4-18> 세련화된 User

그러나, 위와 같은 수준의 도면은 현 단계에서는 작성할 필요가 없다. <그림 4-16>으로 충분하다. 착실하게 한 단계씩 진행해 가기로 하자.


<문제> 수장자료관리서비스시스템(StuffManager)의 사양요소를 세련화하여 모델링하시오.

<해답>

필자가 생각한 UML 도면은 다음과 같다<sup>55)</sup>.



<그림 4-19> 수장자료관리서비스시스템의 사양요소

55)  표시는 UML에서 서브시스템을 나타내는 스테레오타입 아이콘이다.



## [참고] 추적가능성

추적가능성이라는 개념은, 소프트웨어개발에 있어서 오래전부터 매우 중요한 개념들 중의 하나로 알려져 있다. 본문에서 설명한 바와 같이, “추적가능성”이란 “개념의 진화과정을 추적할 수 있는 성질”을 의미한다.

대부분의 경우, 사용자 및 고객으로부터의 요구가 어떻게 진화하여 최종적으로 소스코드까지 변환되는가를 추적하는 경우에 사용된다. 한 개의 요구가 여러 개의 컴포넌트에 의해 구현되거나, 어떤 컴포넌트가 여러 개의 요구사항들과 관계할 수 있으므로, 추적가능성은 기본적으로 다대다의 관계이다.

추적가능성이 중요하다는 점에는 몇 가지 의미가 있다. 예를 들어, 어떤 클래스가 사용자 및 고객의 어떤 요구사항을 구현하기 위하여 만들어진 것인지를 알기 위해서는 클래스로부터 역추적하여 요구사항을 파악해야만 한다. 이를 위하여, 추적가능성이 보장되어야 할 필요가 있다.

반대로, 사용자 및 고객의 요구사항으로부터 추적하는 도중에 단절되어, 최종적으로 해당 요구사항을 구현하고 있는 클래스까지 추적할 수 없는 경우에는, 해당 요구사항은 구현되지 못할 가능성이 있다.

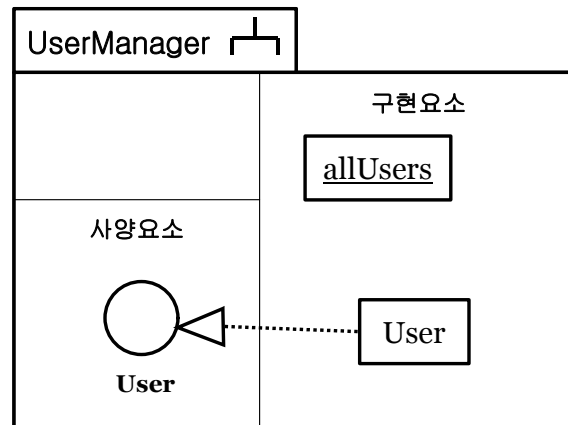
추적가능성의 또 다른 사용방법으로서, 요구사항 변경의 영향을 조사할 수도 있다. 어떤 요구사항 변경이 관계하고 있는 요구사항들로부터, 추적가능성 경로를 따라서 추적해 가면서 영향의 크기를 평가할 수도 있다. 일반적으로 추적가능성이 보장되지 않으면, 요구변경에 대한 영향을 조사할 수 없게 된다.

UML에서, <<trace>>(추적관계)라는 관계가 정의되어 있다. 그러나, 현시점에서 어떤 UML 도구에서도 원활하게 제공되지 못하고 있어서 이용하기 어려운 것이 현실이다.

## 4.15 UserManager의 내부구조/구현요소

한편, UserManager는 사실은 1개 더 구성요소를 갖고 있다. 즉, 개념시나리오에서 출현했던 {이용자}로서(<그림3-22>, p69 참조), “이용자전체”를 나타내는 개념객체였다. 또한, UserManager의 계약을 살펴보았을 때에도 “등록되어 있는 이용자전체”를 나타내는 allUsers로서 표현되었다. 여기서도 allUsers라는 이름을 사용하도록 하자. allUsers는 User들의 집합이다.

단, allUsers는 User와는 다르며, UserManager서브시스템의 외부에는 나타나지 않는 객체, 즉, 구현요소가 된다. 따라서, 사양요소와는 별도로 정중앙에 선분으로 분리된 오른쪽 영역에 표기하는 경우도 있다.

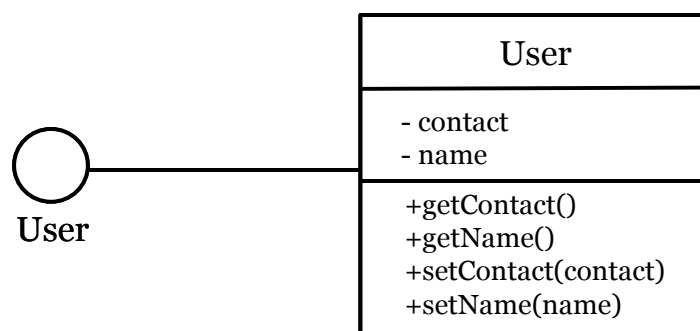


<그림 4-20> allUsers와 User

왼쪽 아래 부분이 사양요소, 왼쪽 윗부분에는 해당 서브시스템이 제공하는 동작들(즉, UserManager 인터페이스의 내부내용)이 나열된다. 이 부분에 대한 표기방법 등에 대해서는 아직 확실하게 정해져 있다고 말하기 어려운 상태이다. 더욱이 서브시스템이 점점 복잡하게 되면, 위와 같이 간단명료하게 표기하기 어렵게 된다. 따라서, 현시점에서는 이정도로 마무리하도록 한다.

allUsers의 아래에 있는 “User”는 사양요소에 있는 User인터페이스를 구현한 클래스이다. 서브시스템의 외부에는, 인터페이스만을 보여주지만 하면 되므로, 누군가가 해당 인터페이스를 구현하지 않으면 안된다. 이와 같은 작업은 UserManager의 책임이므로, 여기서 살펴보기로 한다.

User클래스의 내부를 좀 더 확대해서 살펴보면 다음과 같다.



<그림 4-21> User클래스

User가 제공하는 동작에 대한 계약을 살펴보는 것도 물론 가능하다. 그러나, 이와 같은 경우는, 자명(속성에 대한 accessor들뿐이다)하므로 상세히 작성하지는 않겠다.

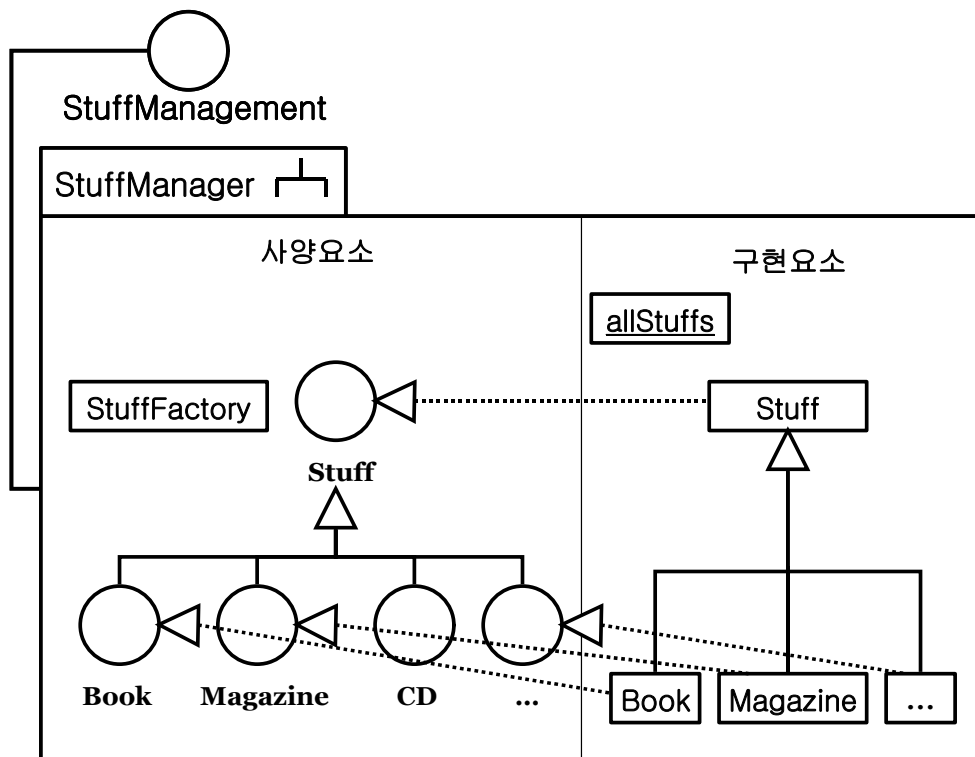
여기서, 각 속성 및 동작명 앞에 붙여진 - 또는 +기호는, 속성 및 동작의 가시성을 나타

낸다. “-”는 private(다른 클래스에서 액세스 불가능), “+”는 public(어떤 클래스로부터도 액세스가능)을 나타낸다. 이밖에도 “#”은 protected를, “~”는 package를 나타낸다.

본서에서는, 현 단계에 이르기까지 가시성에 대하여 설명하지 않았다. 속성들은 모두 “-”로, 동작은 모두 “+”로 가정하기로 한다.

<문제> 저장자료관리서비스시스템(StuffManager)의 구현요소를 세련화하여 모델링하시오.

<해답>



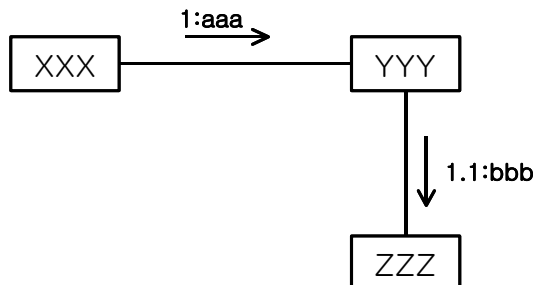
<그림 4-22> 저장자료관리서비스시스템의 구현요소

#### 4.16 서브시스템내의 협조

UserManager서비스시스템의 구성요소가 모두 갖추어졌으므로, 각 구성요소들을 사용하여 올바르게 UserManager의 인터페이스를 구현할 수 있는지 여부를 점검해 보기로 하자.

앞서, Use Case View에서는 Use Case Scenario를, 개념객체에서는 개념시나리오를 사용하여 각 사항들이 올바르게 진행되는지 여부를 확인하였다. 여기서도 기본적으로 앞의 방법들과 같다. 인터페이스의 각 동작들에 대하여, 동작의 내부 움직임을 실제로 시험해 보기로 한다(일단은, 도면위에서 움직임을 시험한다). 구성요소들 사이에서 정보의 수수작용이 원활히 진행되고, 구성요소 각각의 책임에 따라서 각 사항들이 적절히 진행됨을 확인하는 것이다.

지금까지는, 동작을 UML의 Sequence Diagram을 사용하여 표기하였으나, 여기서는 Collaboration diagram을 사용하여 표현해 보기로 한다. Collaboration diagram 또한 기본적인 의미에서 Sequence Diagram과 거의 같고, 표현방법 몇 가지가 다를 뿐이다.



<그림 4-23> Collaboration Diagram

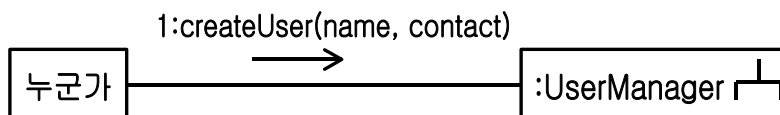
Sequence Diagram은, 가로축에 객체들을 나열하고, 위에서 아래로 시간이 흐르는 형상으로 작성되었다. Collaboration Diagram에서는, 객체들은 보기 쉽도록 자유롭게 배치되며, 시간의 흐름은 메시지를 나타내는 화살표에 번호를 부여하여 나타낸다.

Sequence Diagram을 사용할지 Collaboration Diagram을 사용할지는, 무엇을 주로 나타내려는가에 따라 선택하도록 한다. 시간의 흐름이 중요한 경우에는 Sequence Diagram을 사용하고, 객체들 사이의 협조관계가 중요한 경우에는 Collaboration Diagram을 사용한다. 그 이외에는 설계자의 습관 및 기호에 맞게 선택하도록 한다.

## 4.17 협조(Collaboration)

“협조(Collaboration)”란, 어떤 서비스를 구현하는데 있어서 어떤 객체들이 참가하여 어떻게 협력해서 동작하는가를 표현하는 것이다. UserManager의 협조를 살펴보기로 하자.

우선, createUser를 살펴보자. 다른 서브시스템 또는 누군가가 UserManager에게 “createUser”의 실행을 의뢰한다. 즉, “이용자를 한사람 등록해 주십시오”라는 의뢰를 하는 것이다.



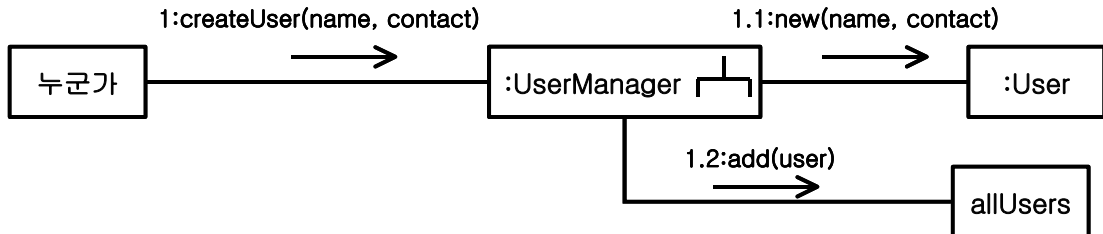
<그림 4-24> createUser의 실행의뢰

UserManager는 이와 같은 실행을 의뢰받아서 어떻게 하면 될까? 우선은, User객체를 1개 생성해야 된다.

“new”는 새로운 인스턴스를 생성하는 일반적인 조작이름을 나타낸다. 그리고, 새롭게 생성된 User객체를 allUsers에 추가한다.

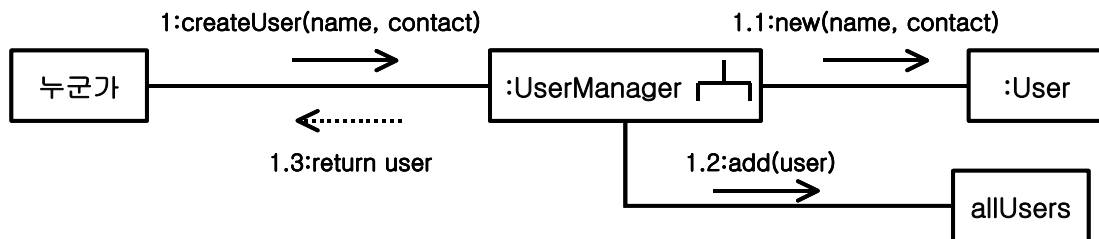


<그림 4-25> User객체를 생성한다.



<그림 4-26> allUsers에 User객체를 추가

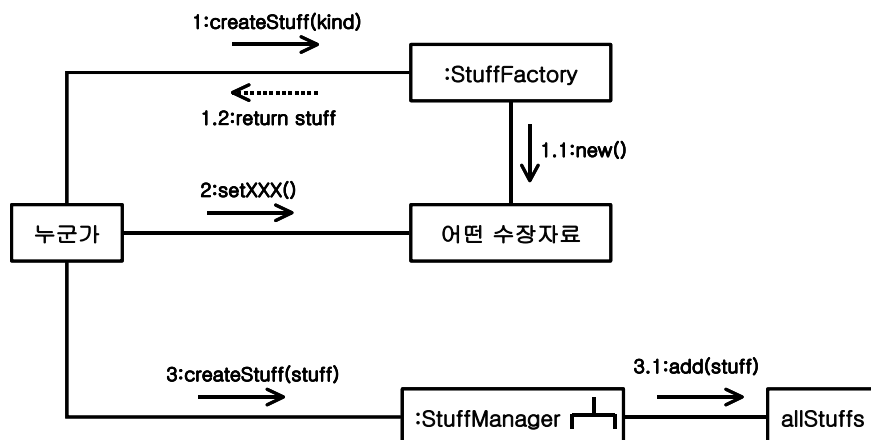
“add”는 집합allUsers에 요소를 추가하는 일반적인 조작의 이름이다. 추가한 후에, 앞서 생성한 user객체를 이용자등록 작업을 의뢰한 누군가에게 반환하도록 한다.



<그림 4-27> User객체의 추가 후, 추가한 User객체를 작업의뢰한 쪽으로 반환한다.

<문제> 수장자료관리서브시스템(StuffManager)의 Collaboration Diagram을 작성하시오.

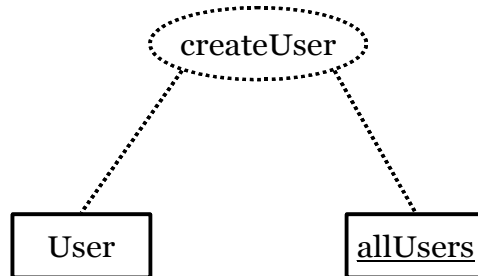
<해답>



<그림 4-28> 수장자료관리서브시스템의 Collaboration Diagram

[createUser 협조작업에 User클래스와 allUsers객체가 참가]

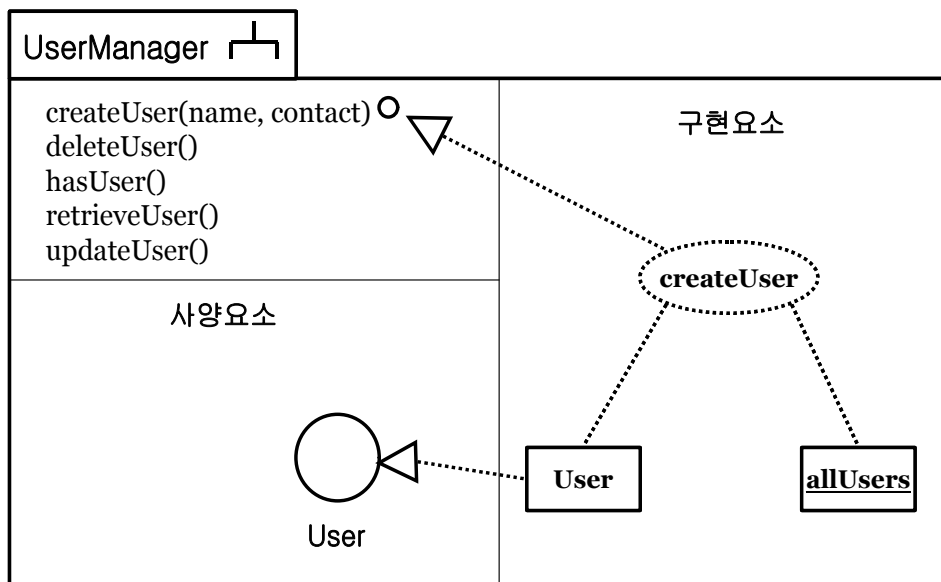
이와 같은 createUser 협조 작업에, User클래스와 allUsers객체가 참가하고 있음을 UML을 사용하여 다음의 그림과 같이 나타낸다.



<그림 4-29> createUser협조작업에,  
User클래스와 allUsers객체가 참가

여기서 점선으로 둘러싸인 createUser가 createUser협조작업을 나타낸다. createUser로부터 User 및 allUsers로 향하는 점선은 “참가하고 있다”라는 의미이다. 협조의 내부내용은, 앞서 작성한 Collaboration Diagram이라고 생각해도 좋다.

또한, UserManager 서브시스템의 인터페이스들 중의 하나인, createUser를 createUser협조작업으로 구현하고 있음을 다음과 같이 표현할 수도 있다<sup>56)</sup>.



<그림 4-30>createUser를 createUser협조작업으로 구현

지금까지의 작업을 살펴보면, createUser이외의 동작에 대해서도 협조를 파악하는 것은 그렇게 어렵지 않을 것임을 알 수 있다.

56) 여기서는, UserManager인터페이스를 왼쪽 윗부분에 기입하였다. 이와같이 작성해도 좋다.

## 4.18 사양모델링의 종료

UserManager 서브시스템의 사양모델링은, 이 정도에서 종료하기로 한다. 사양모델링의 목적은, 해당 서브시스템의 사양을 명확히 하는 것이다.

“사양”의 절반정도는 외부사양, 즉, 해당 서브시스템을 이용하는 시스템 및 다른 서브시스템에게 제공하는 서비스를 명확히 하는 것이다. 따라서, 서브시스템 레벨에서의 인터페이스를 결정하고, 해당 계약도 명시하였다. 나머지 절반은 내부사양, 즉, 서브시스템 내부의 구성요소들을 명확히 하는 것이다. 이를 위하여, 구성요소들을 열거하고, 해당 인터페이스 및 협조를 살펴보았다.

이번에는 매우 단순한 서브시스템을 대상으로 하였으므로 1번의 반복 작업만 수행하였다. 그러나, 실제로는, 사양모델링에서는 여러 번 Recursion하여 여러 단계의 상세화를 수행하는 경우가 많다. 본문에서도(“내부구조/사양요소-UserManager”), 맨 마지막에 잠시 상세화를 진행하는 예를 살펴보았다.

사양모델링에 있어서, 실제로 어느 정도까지 상세화를 진행해야하는지는 프로젝트의 상황 및 정책에 따라 다르다.

아무튼, 어느 정도 규모의 서브시스템인가가 중요하다. 앞으로의 작업도 서브시스템을 단위로 하여 진행할 것이다. 대략적으로는, 1~2주 정도로 움직일 수 있는 규모의 서브시스템을 대상으로 하는 것이 좋을 것이다. 각각의 서브시스템(1개가 아니더라도 좋다)을 1명 내지는 여러 명으로 구성된 소규모 팀이 담당하도록 하자.

단, “사양모델링의 종료”라고 하더라도 시스템의 모든 서브시스템들의 사양모델링이 종료되는 것을 기다린 다음에 다음 작업으로 진행하는 것은 아니다. 서브시스템들 사이의 의존성을 고려하여, 사양모델링이 종료된 서브시스템부터 다음 작업을 시작하면 된다. 또한, 일단 종료되었다고 하더라도 다음번 Iteration에서 같은 서브시스템이 배정되어 작업해야하는 경우도 있다.