The Object-Oriented Thought Process

Chapter 02

How to Think in Terms of Objects

Contents

- Knowing the Difference Between the Interface and the Implementation
- Using Abstract Thinking When Designing Interfaces
- Giving the User the Minimal Interface Possible

How to Think In Terms of Objects

Three important things you can do to develop a good sense of the OO thought process are:

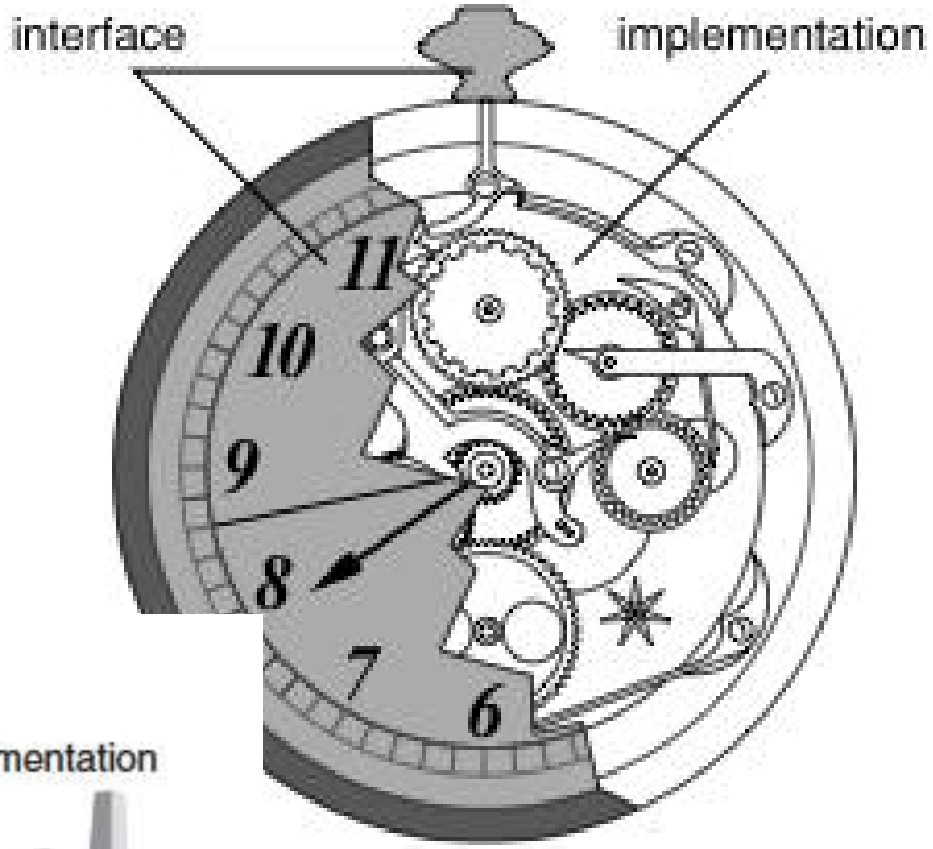
- Knowing the difference between the interface and implementation
- Thinking more abstractly
- Giving the user the minimal interface possible

Knowing the Difference Between the Interface and the Implementation

To understand the difference between the interface and the implementation.

- When designing a class, what the user needs to know and what the user does not need to know are of vital importance.
- The data hiding mechanism inherent with encapsulation is the means by which nonessential data is hidden from the user.

- Interface vs. Implementation



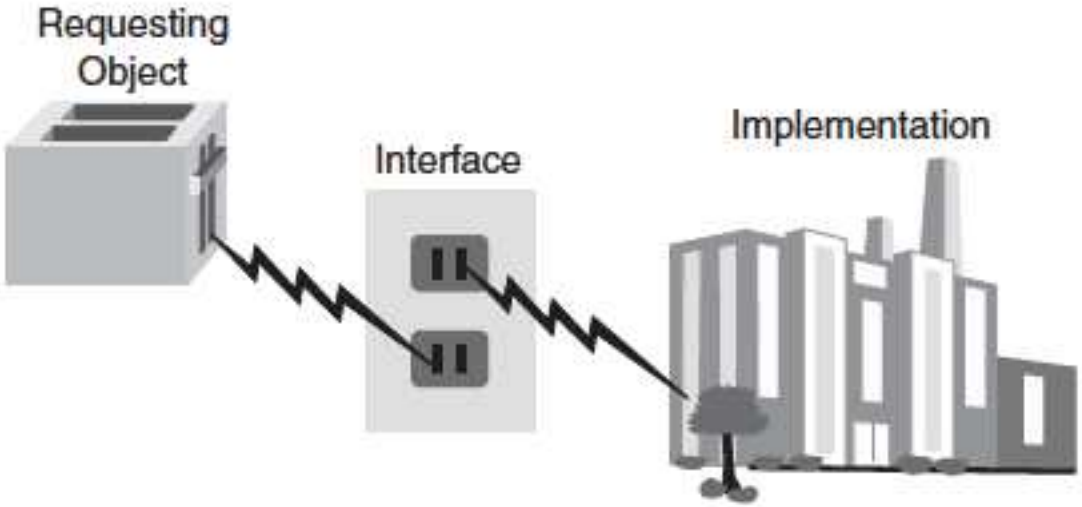
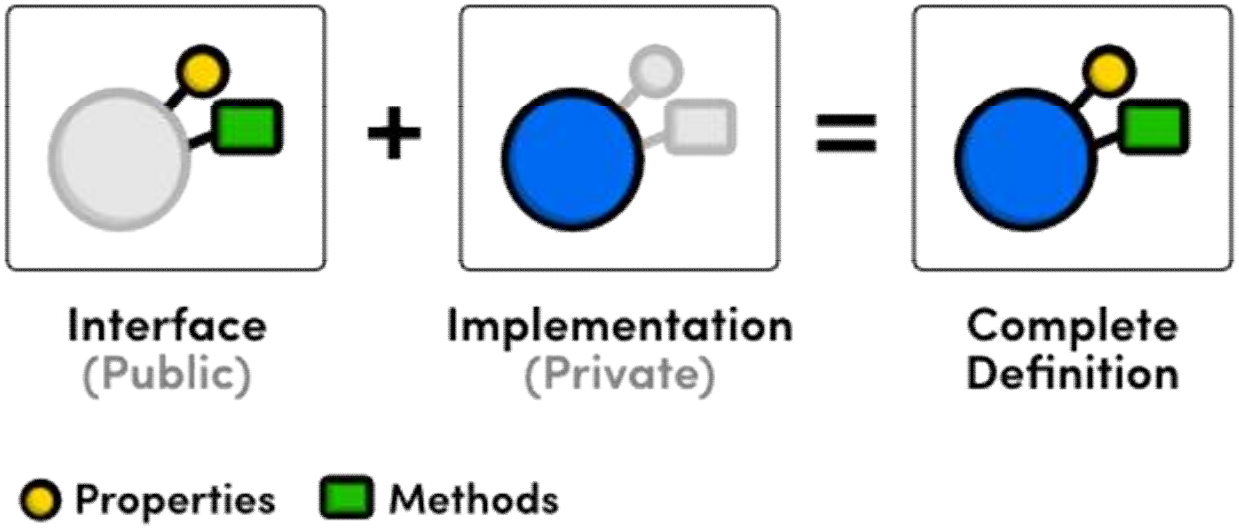


Figure 2.1 Power plant revisited.

Properly constructed classes are designed in two parts

- the interface and the implementation.



The Interface

The services presented to an end user compose the interface.

- In the best case, only the services the end user needs are presented.
- As a general rule, the interface to a class should contain only what the user needs to know.

The Implementation

The implementation details are hidden from the user.

- A change to the implementation should not require a change to the user's code.
- If the interface does not change, the user does not care whether or not the implementation is changed.

An Interface/Implementation Example

Requirements we might want to use for the database reader:

- We must be able to
  - open a connection to the database.
  - close the connection to the database.
  - position the cursor on the first record in the database.
  - position the cursor on the last record in the database.
  - find the number of records in the database.
  - determine whether there are more records in the database (that is, if we are at the end).
  - position the cursor at a specific record by supplying the key.
  - retrieve a record by supplying a key.
  - get the next record, based on the position of the cursor.

- Class diagram representing a possible interface to the `DataBaseReader` class

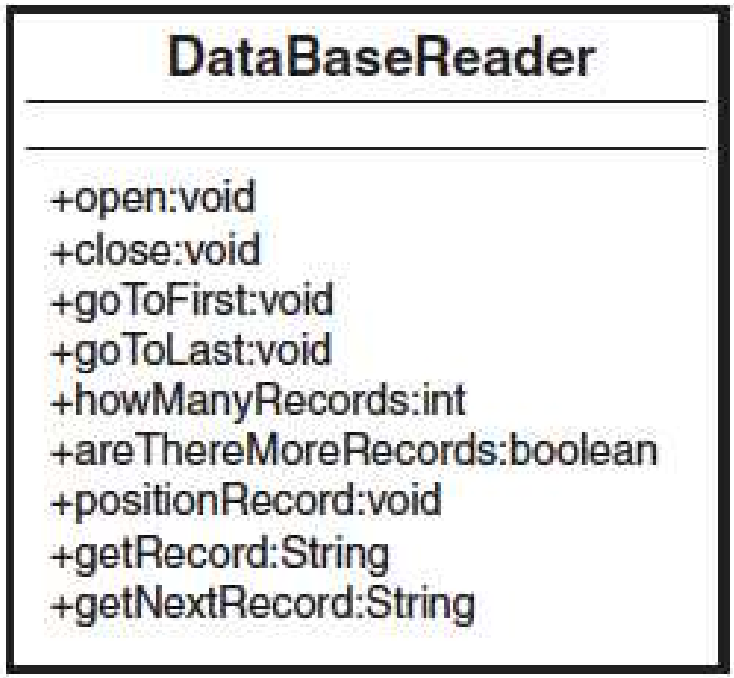


Figure 2.2 A Unified Modeling Language class diagram for the **`DataBaseReader`** class.

For each of the requirements we listed, we need a corresponding **method** that provides the functionality we want.:

- To effectively use this class, do you, as a programmer, need to know anything else about it?
- Do you need to know how the internal database code actually opens the database?
- Do you need to know how the internal database code physically positions itself over a specific record?
- Do you need to know how the internal database code determines whether there are any more records left?

- The application programmer will most likely be at least one more abstract level away from the implementation.

- open() method

```
public void open(String Name){
  /* Some application-specific processing */
  /* call the Oracle API to open the database */
  /* Some more application-specific processing */
};
```

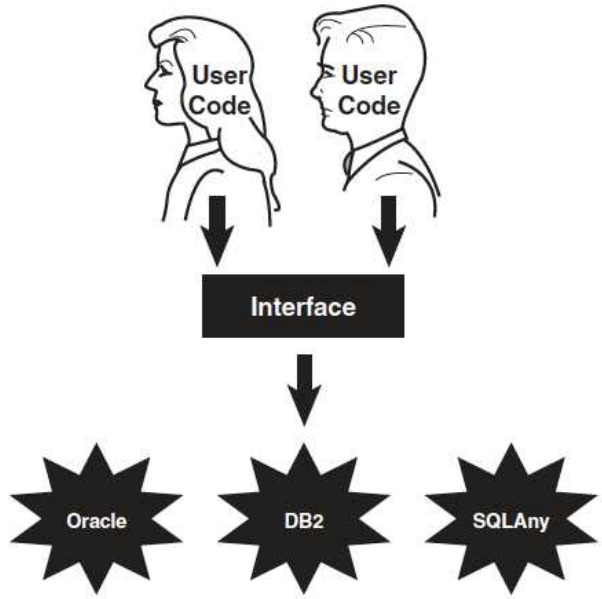


Figure 2.3 The interface.

– To change the implementation

```
public void open(String Name){
  /* Some application-specific processing
  /* call the SQLAnywhere API to open the database*/
  /* Some more application-specific processing */
};
```

Using Abstract Thinking When Designing Interfaces

- Our goal is to design abstract, highly reusable classes—and to do this we will design highly abstract user interfaces

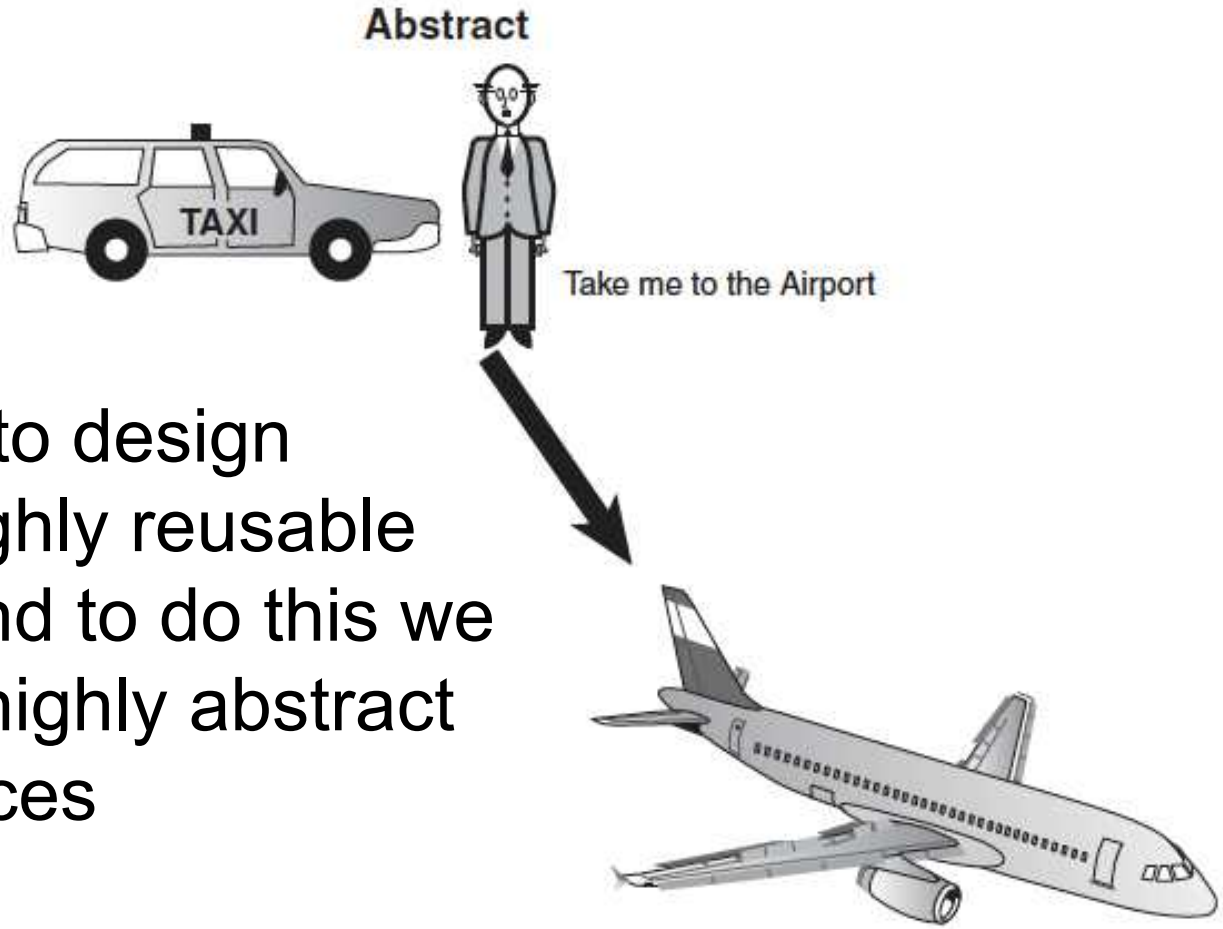


Figure 2.4 An abstract interface.

Abstract Thinking

One of the main advantages of OO programming is that classes can be reused.

- Reusable classes tend to have interfaces that are more abstract than concrete.
- Concrete interfaces tend to be very specific.
- Abstract interfaces are more general.

- interface vs. implementation
  - What vs. how

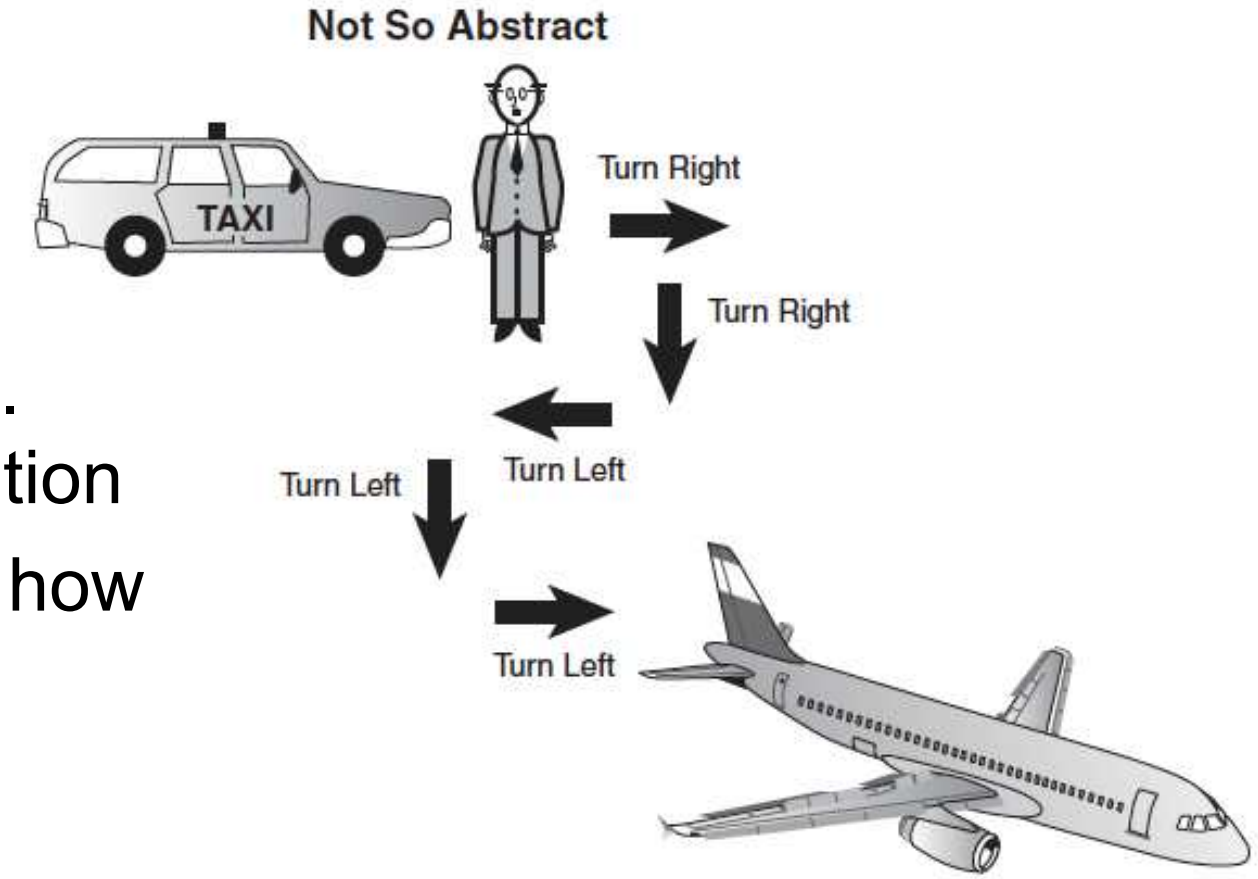


Figure 2.5 A not-so-abstract interface.

Giving the User the Minimal Interface Possible

- Determining the Users
- Object Behavior
- Environmental Constraints
- Identifying the Public Interfaces
- Identifying the Implementation

The Minimal User Interface

Give the users only what they absolutely need.

- It is better to have to add interfaces because users really need it than to give the users more interfaces than they need.
- Public interfaces define what the users can access.
- It is vital to design classes from a user's perspective and not from an information systems viewpoint.
- Make sure when you are designing a class that you go over the requirements and the design with the people who will actually use it—not just developers.

The Minimal User Interface

- Minimal Interface
  - One way to determine the minimalist interface is to initially provide the user no public interfaces.
  - add interfaces only when it is requested.
  - Never assume that the user needs something.

Determining the Users

Who are the users?

- In the Taxi example, the first impulse is to say the *customers.*
  - *This is only about half right*

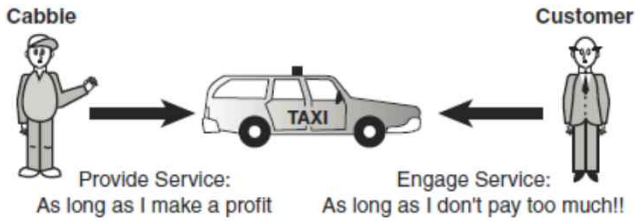


Figure 2.6 Providing services.

- *Although the customers* are certainly users, the developers must be a partner with the users.
  - In short, users can have unrealistic expectations.

Object Behavior

After the users are identified:

- You must determine the behaviors of the objects.
- From the viewpoint of all the users, begin identifying the purpose of each object and what it must do to perform properly.
- Note that many of the initial choices will not survive the final cut of the public interface.

Environmental Constraints

Environmental constraints are almost always a factor.

- Computer hardware might limit software functionality.
- A system might not be connected to a network, or a company might use a specific type of printer.

The Public Interfaces

Think about how the object is used and not how it is built.

- You might discover that the object needs more interfaces
- Nailing down the final interface is an iterative process
- It is often recommended that each interface model only one behavior.

What do you need to do to use the taxi object?

- Have a place to go.
- Hail a taxi.
- Pay the cabbie money.

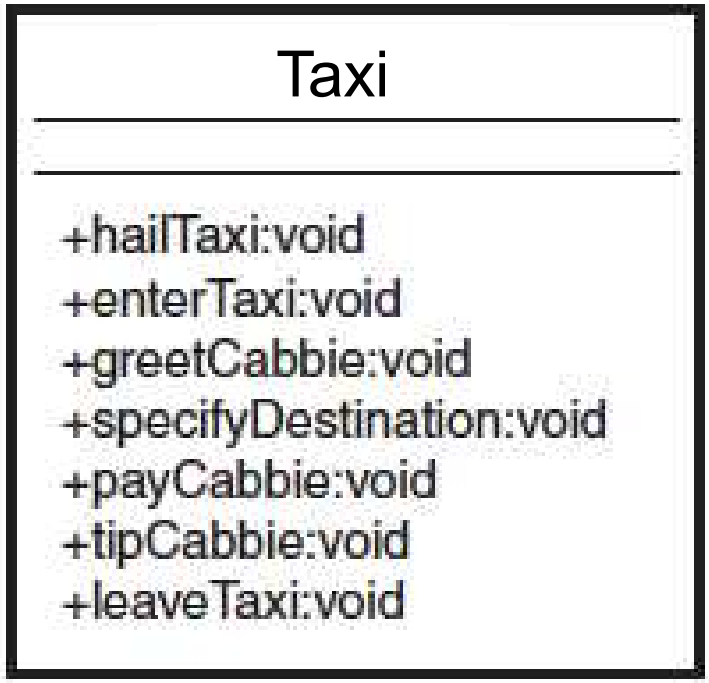


Figure 2.7 The methods in a Taxi class.

Identifying The Implementation

Technically, anything that is not a public interface can be considered the implementation.

- This means that the user will never see any of the methods that are considered part of the implementation.
  - Including the method's signature (which includes the name of the method and the parameter list), as well as the actual code inside the method.

[설계] 1 장 UML의 소개

(교재: J. Schmuller 저/곽용재 역, 초보자를 위한 UML 객체지향설계, 제3판, 정보문화사, 2006)

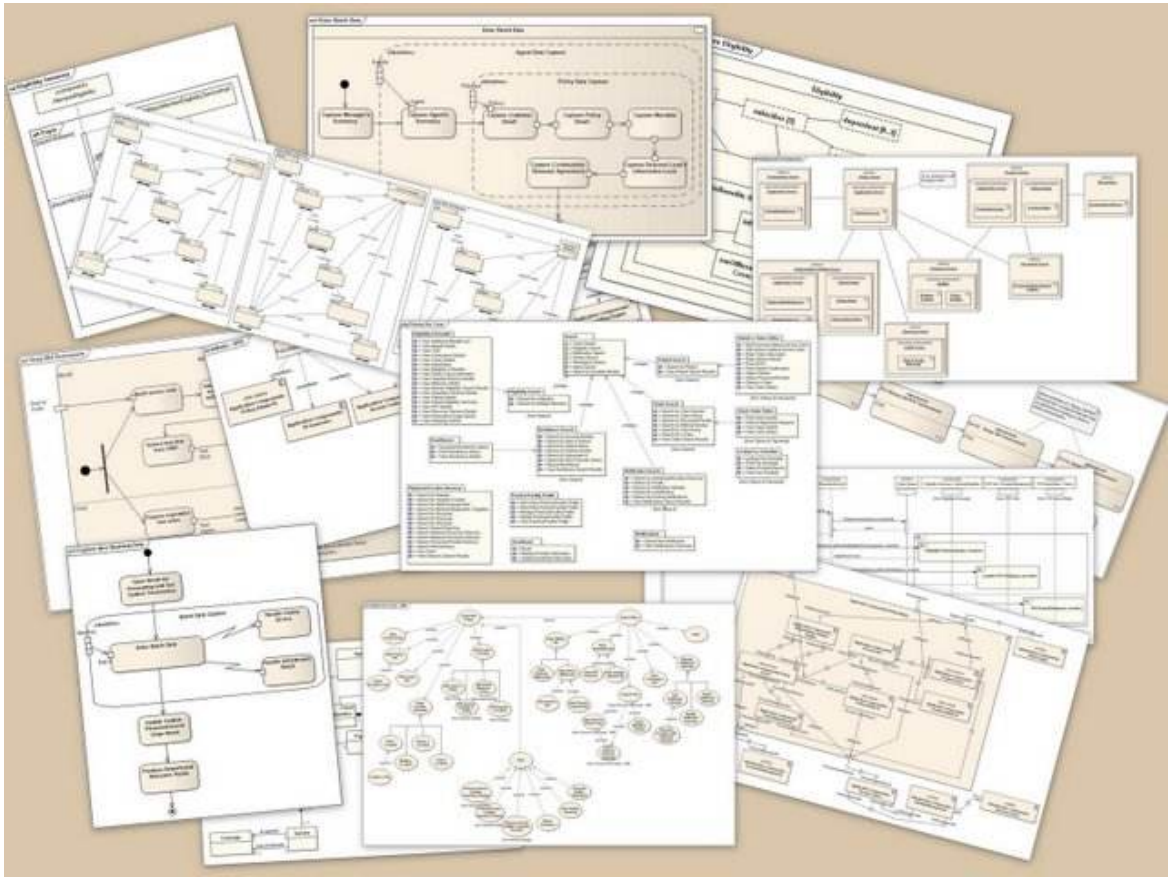
차례

1.1 복잡한 세상 바로잡기

- 왜 UML이 필요할까?
  - UML(Unified Modeling Language)은 오늘날의 객체지향 시스템 개발 분야에서 가장 각광받는 도구 중 하나
  - UML은 시스템 개발자가 자신의 비전(vision)을 구축하고 반영하는데 있어서 표준적이고 이해하기 쉬운 방법으로 할 수 있도록 지원
  - 자신의 설계 결과물을 다른 사람과 효과적으로 주고받으며 공유할 수 있는 메커니즘 제공

1.1 복잡한 세상 바로잡기

- UML Diagrams



(출처: http://www.wikipedia.org, Unified Modeling Language)

1.2 UML의 탄생 배경

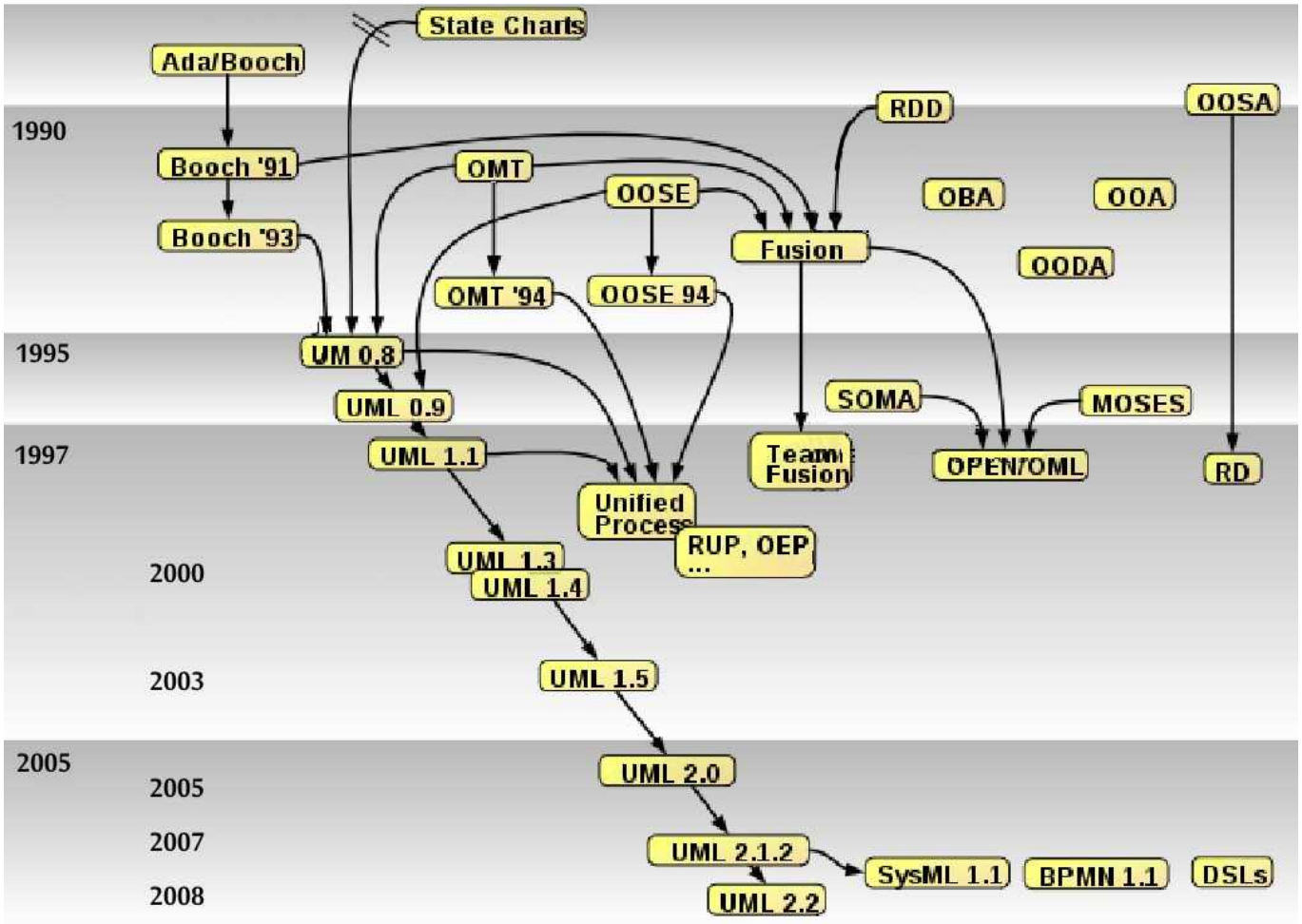
- UML 1.0 (1997) 은 세 친구(Three Amigos)에 의해 만들어졌다.
  - Grady Booch - Booch Notation
  - James Rumbaugh - Object Modeling Technique (OMT)
  - Ivar Jacobson - Objectory methodology

- UML 은 소프트웨어 업계의 명실 상부한 표준이 되었으며, 계속 수정 보완되고 있다.

- UML 1.3과 UML 1.4 (2000)그리고 UML 1.5 (2003)가 나와 있고, 최근에는 UML 2.0 (2005) 이 OMG에 의해 승인

(참고: http://www.omg.org, Object Management Group)

1.2 UML의 탄생 배경



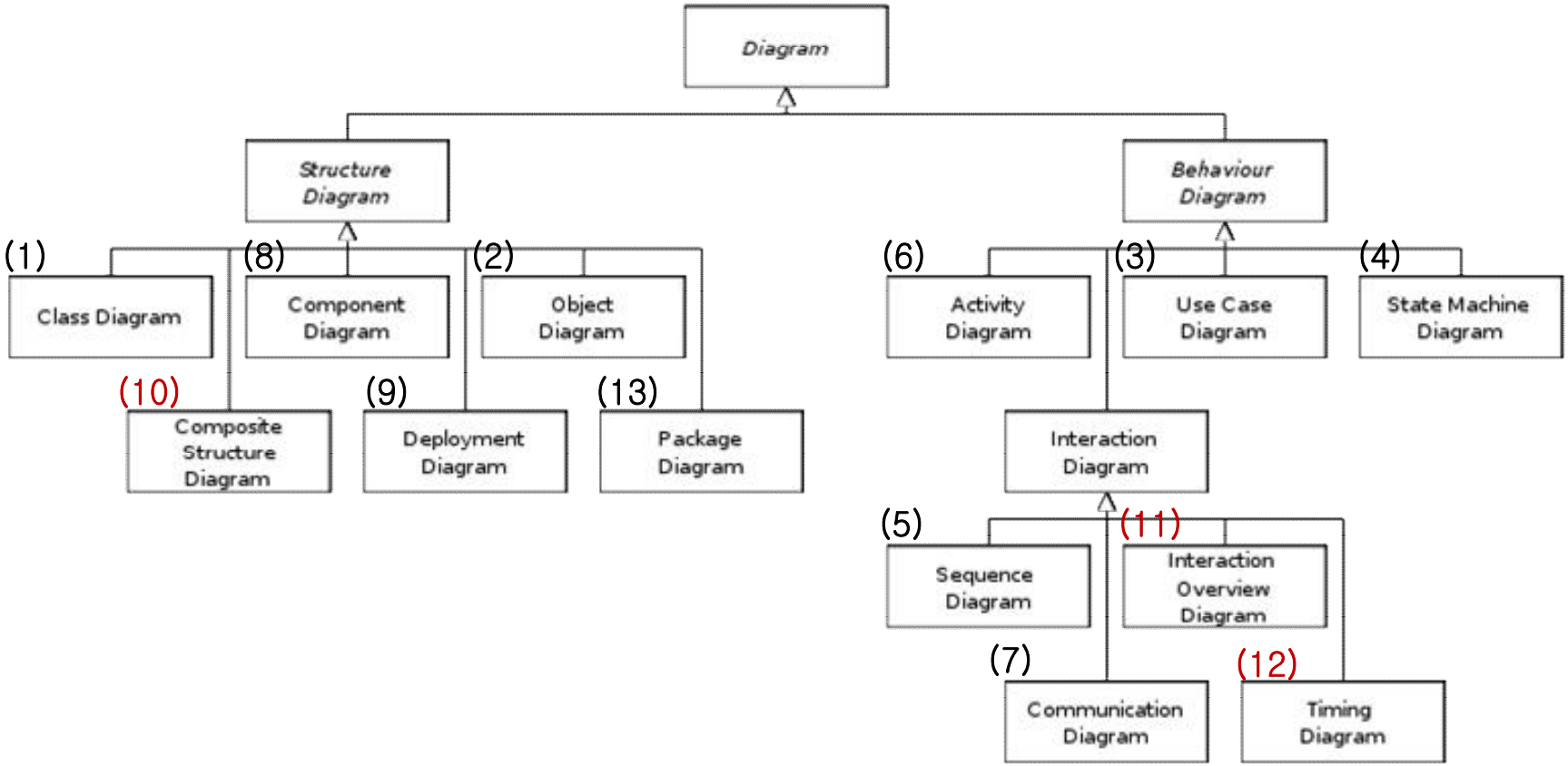
(출처: http://www.wikipedia.org, Unified Modeling Language)

1.3 UML의 구성요소

- UML 다이어그램
  - UML의 여러 가지 그래픽 요소는 하나의 큰 그림, 즉 다이어그램을 그리는데 사용
  - 다이어그램의 목적은 시스템을 여러 가지 관점에서 볼 수 있는 뷰(View)를 제공하는 것이며, 이러한 뷰의 집합을 모델(Model)이라 함
  - UML 모델은 시스템 자체의 “목적 행동”을 설명하는 언어

1.3 UML의 구성요소

- UML diagram



(출처: http://www.wikipedia.org, Unified Modeling Language)

1.3 UML의 구성요소

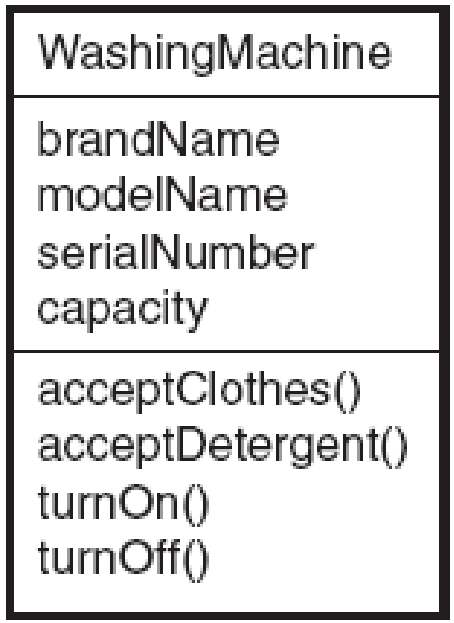
(1) 클래스 다이어그램(Class Diagram)

- 클래스(class) - 비슷한 속성과 공통적인 행동 수단을 지닌 것들의 범주 혹은 그룹

- 예: 세탁기 클래스
  - 속성 : 브랜드 이름, 모델, 일련 번호, 용량 등
  - 행동 :
    - “옷을 넣는다(accept clothes)”
    - “세제를 뿌린다(accept detergent)”
    - “전원을 켜다(turn on)”
    - “전원을 끄다(turn off)” 등

1.3 UML의 구성요소

- UML 클래스 기호
  - 예: 세탁기 클래스



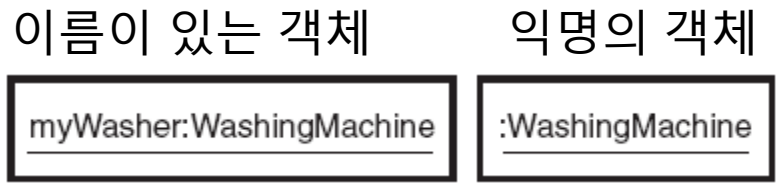
(출처: J. Schmuller, Teach Yourself UML in 24 Hours, 3rd Ed., Sams, 2004)

1.3 UML의 구성요소

(2) 객체 다이어그램 (Object Diagram)

- 객체(Object)
  - 클래스의 인스턴스
  - 즉, 값이 주어진 속성과 행동을 가지고 있는 개별적인 개체

- UML 객체 아이콘



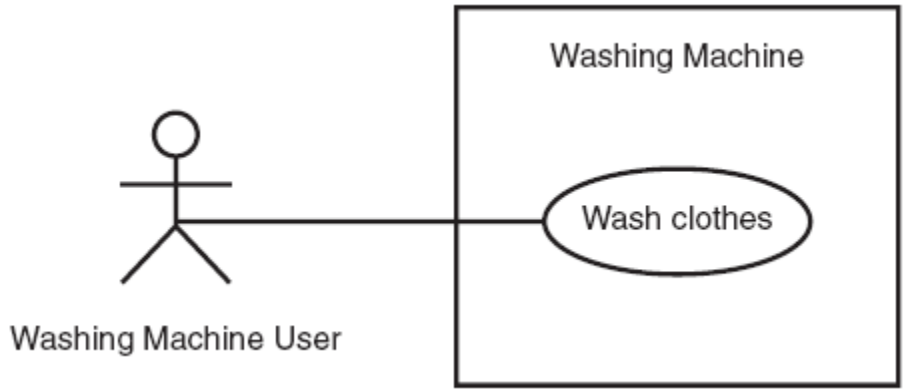
(출처: J. Schmuller, Teach Yourself UML in 24 Hours, 3rd Ed., Sams, 2004)

1.3 UML의 구성요소

(3) 유스케이스 다이어그램 (Usecase Diagram)

- 유스케이스(use case)
  - 사용자의 입장에서 본 시스템의 행동

- 예: 세탁기 유스케이스



(출처: J. Schmuller, Teach Yourself UML in 24 Hours, 3rd Ed., Sams, 2004)

1.3 UML의 구성요소

(4) 상태 다이어그램 (State Diagram)

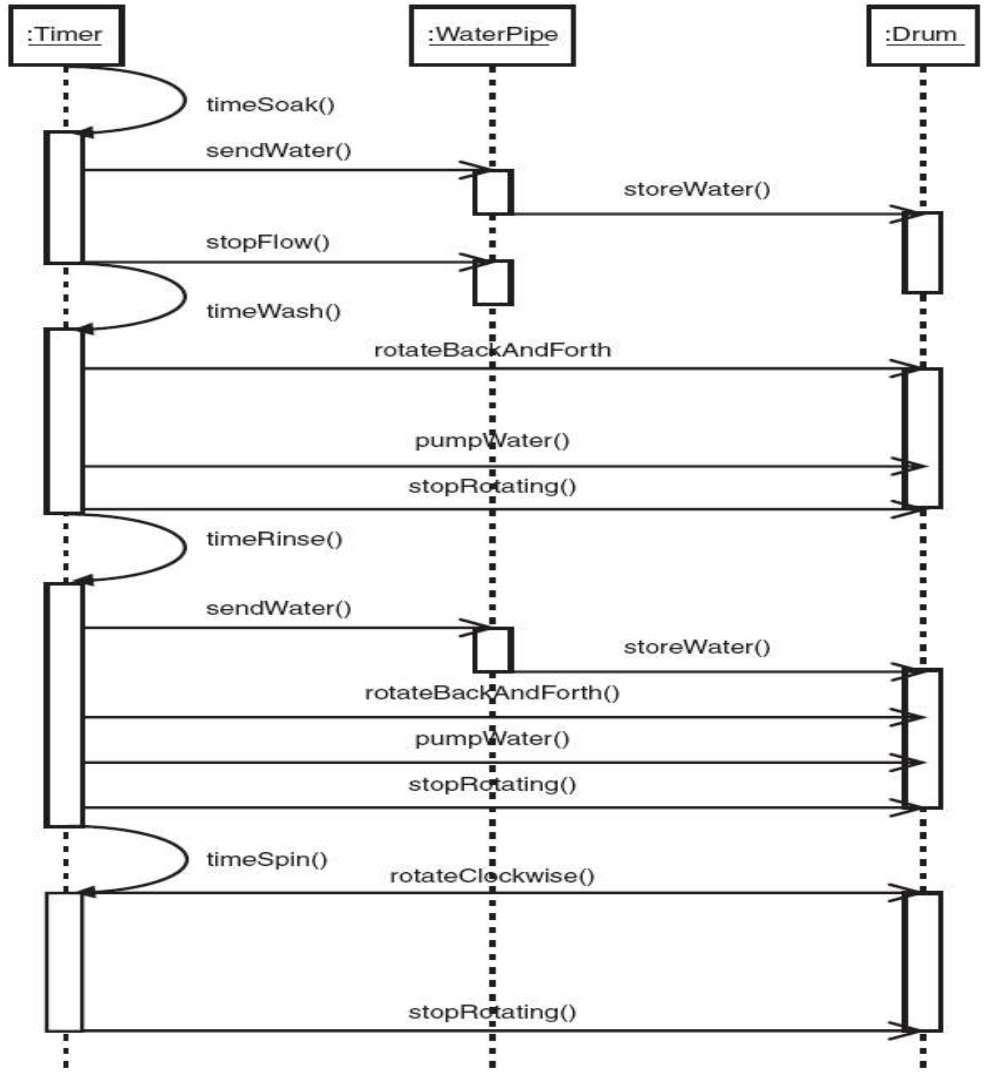
- 객체는 시간에 따라 각기 다른 상태에 있을 수 있다.
- 예: 세탁기 상태 다이어그램 - 세탁기의 상태가 단계적으로 변해감을 알 수 있다.

(출처: J. Schmuller, Teach Yourself UML in 24 Hours, 3rd Ed., Sams, 2004)

1.3 UML의 구성요소

(5) 시퀀스 다이어그램 (Sequence Diagram)

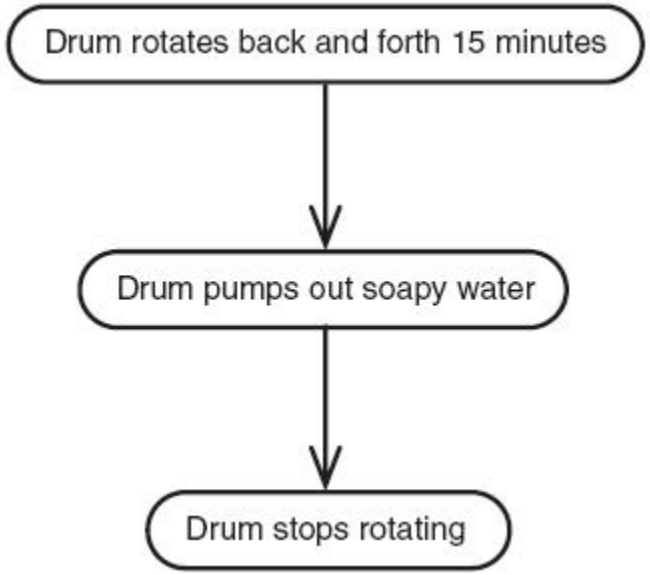
- 객체들끼리 주고받는 메시지의 순서를 시간의 흐름에 따라 표현
- 예: 세탁기 시퀀스 다이어그램
  - Wash clothes 유스케이스



1.3 UML의 구성요소

(6) 활동 다이어그램
(Activity Diagram)

- 오퍼레이션이나 처리 과정이 수행되는 동안 일어나는 일들을 표현
- 플로우차트(flowchart)와 흡사
- 예: 세탁기 활동 다이어그램

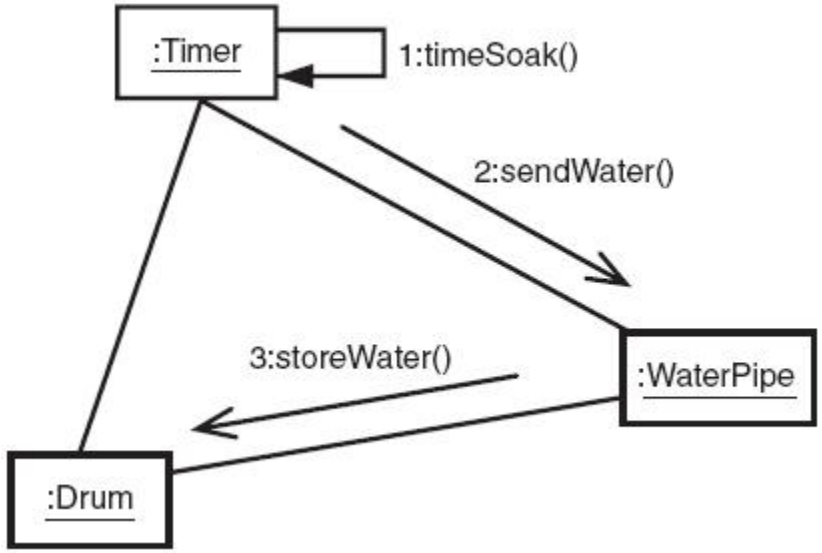


(출처: J. Schmuller, Teach Yourself UML in 24 Hours, 3rd Ed., Sams, 2004)

1.3 UML의 구성요소

(7) 통신 다이어그램 (Communication Diagram)

- 하나의 시스템 구성요소는 다른 구성요소들과 손발을 맞추면서 시스템 전체의 목적을 이루어 나아간다.
- Interaction에 참여하는 객체들의 연관을 나타냄
- 시퀀스 다이어그램과 함께 표현



예: 세탁기 통신 다이어그램

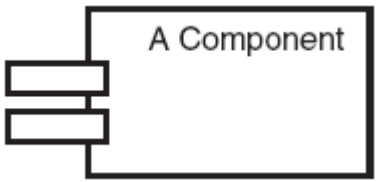
(출처: J. Schmuller, Teach Yourself UML in 24 Hours, 3rd Ed., Sams, 2004)

1.3 UML의 구성요소

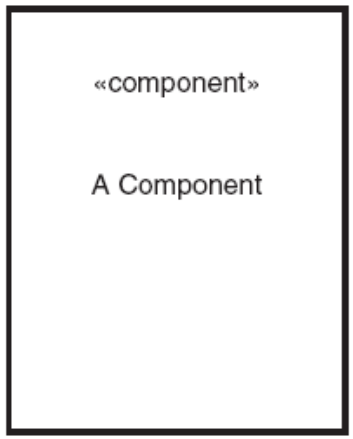
(8) 컴포넌트 다이어그램 (Component Diagram)

- 컴포넌트 : 기계 부품과 같이 소프트웨어도 부품으로 제작한 다음 이를 조립해 더 복잡한 소프트웨어를 제작할 수 있는 조립형 소프트웨어

- UML 1.x



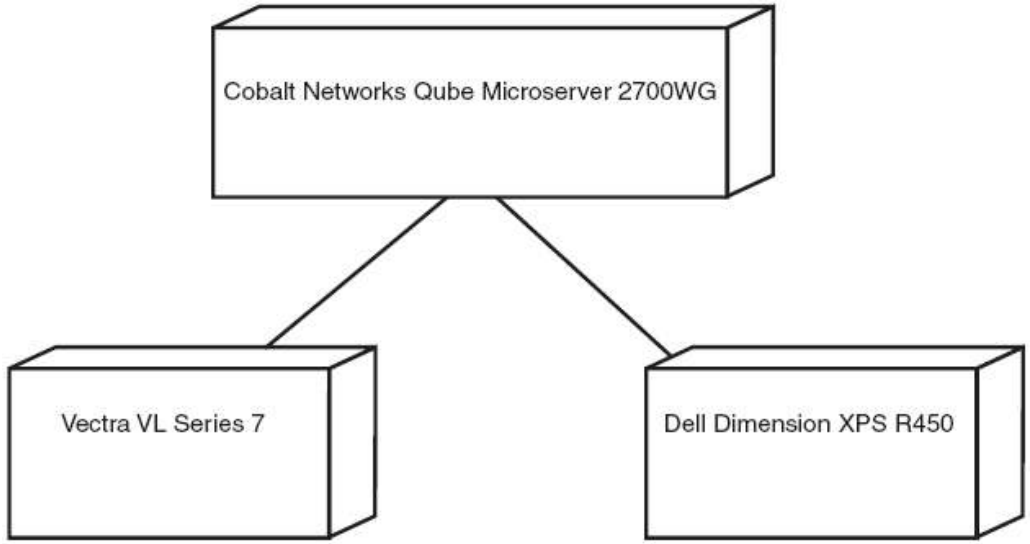
UML 2.0



1.3 UML의 구성요소

(9) 배포 다이어그램 (Deployment Diagram)

- 컴퓨터를 기반으로 하는 시스템의 물리적 구조를 표현
- 컴퓨터와 부가장치, 연결관계, 기계에 설치된 소프트웨어까지 표시



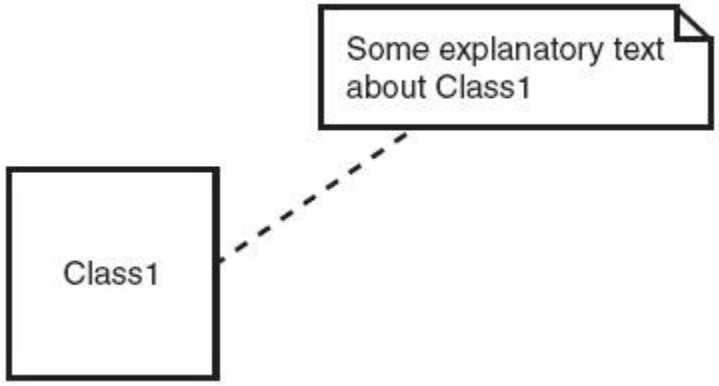
1.4 이외의 것들

- UML은 다이어그램을 조직화하고 확장하는 방법을 제공
  - 노트 (note)
  - 스테레오 타입 (stereotype)
  - 키워드 (keyword)

1.4 이외의 것들

- 노트(note)

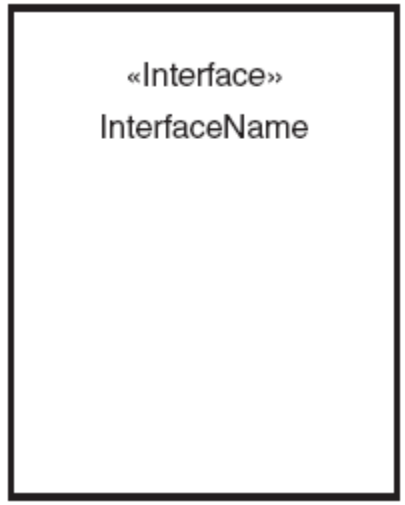
  - 붙였다 떼었다 할 수 있는 노란 메모지
  - 모든 다이어그램 요소에 추가적인 설명 제공



(출처: J. Schmuller, Teach Yourself UML in 24 Hours, 3rd Ed., Sams, 2004)

1.4 이외의 것들

- 키워드와 스테레오타입
  - 스테레오타입(stereotypes) - 기존의 UML 요소를 기본으로 하여 다른 요소를 새로 만들 수 있게 하는 장치
  - 키워드(keyword) - UML 요소가 원래 의미가 아닌 새로운 다른 의미로 사용되었음을 나타내며 거듭 인용부호(« ») 안에 위치



(출처: J. Schmuller, Teach Yourself UML in 24 Hours, 3rd Ed., Sams, 2004)

1.5 UML2.0의 새로운 다이어그램

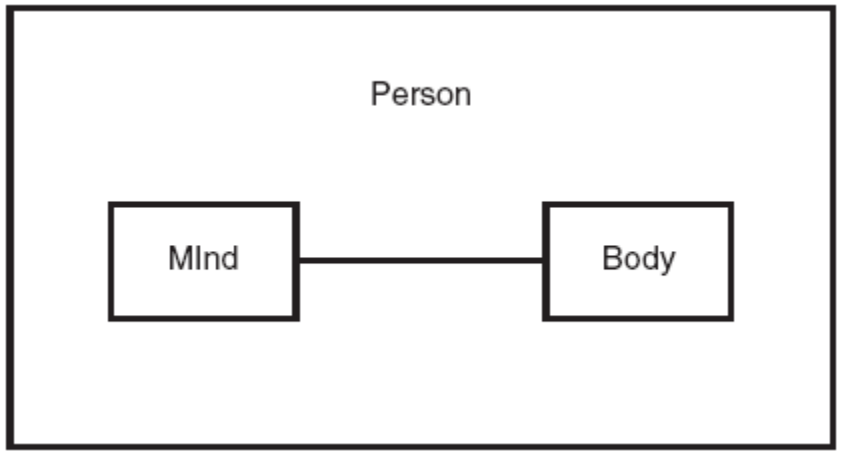
- UML 2.0 의 다이어그램의 변화/추가
  - UML 1.X의 다이어그램이 새롭게 변화된 것
    - 컴포넌트 다이어그램
  - 새로운 아이디어가 여러 방면으로 추가된 다이어그램
    - 복합체 구조 다이어그램
    - 교류 개요 다이어그램
    - 타이밍 다이어그램

(출처: J. Schmuller, Teach Yourself UML in 24 Hours, 3rd Ed., Sams, 2004)

1.5 UML2.0의 새로운 다이어그램

(10) 복합체 구조 다이어그램(composite structure diagram)

- 각 컴포넌트 클래스를 전체 클래스 안에 위치시킴으로써 클래스의 내부 구조가 어떤 것으로 이루어져 있는지 살펴보는 데 매우 유용
- UML 2.0에서 새로 도입



(출처: J. Schmuller, Teach Yourself UML in 24 Hours, 3rd Ed., Sams, 2004)

1.5 UML2.0의 새로운 다이어그램

(11) 교류 개요 다이어그램(interaction overview diagram)

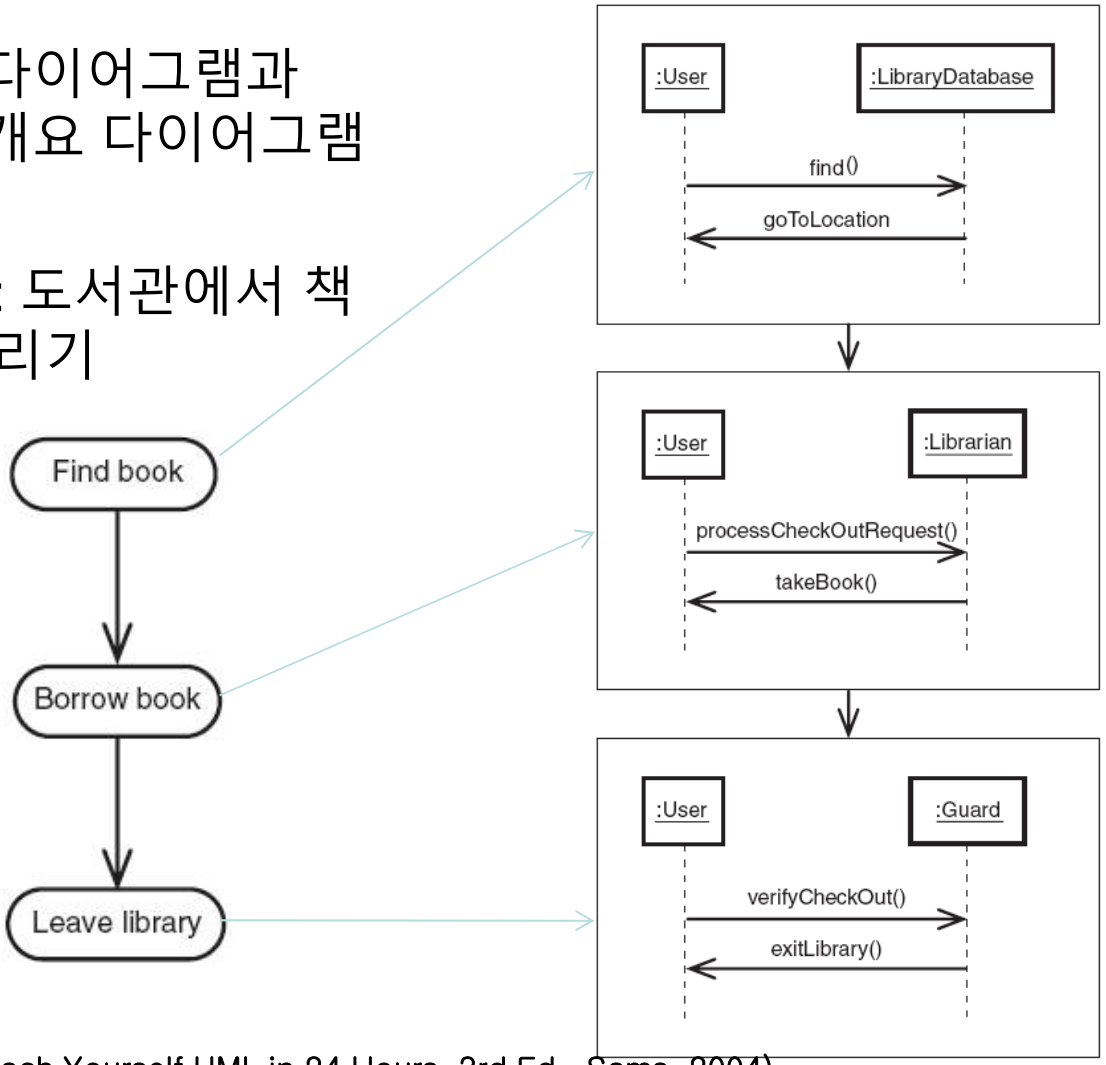
- 각 활동마다 객체 사이에 시간의 흐름을 갖는 메시지가 존재한다면 몇몇 활동 부분은 시퀀스 다이어그램이나 통신 다이어그램(혹은 두 다이어그램의 조합)으로 바뀌어야 한다.

- UML 2.0에서 새로 도입

(출처: J. Schmuller, Teach Yourself UML in 24 Hours, 3rd Ed., Sams, 2004)

1.5 UML2.0의 새로운 다이어그램

- 활동 다이어그램과 교류 개요 다이어그램
  - 예: 도서관에서 책 빌리기



(출처: J. Schmuller, Teach Yourself UML in 24 Hours, 3rd Ed., Sams, 2004)

1.5 UML2.0의 새로운 다이어그램

(12) 타이밍 다이어그램 (Timing Diagram)

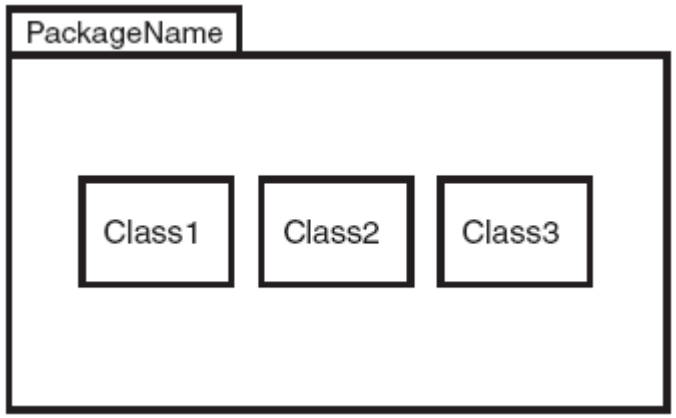
- 시퀀스 다이어그램은 시간에 관해서는 전혀 언급하지 않음
- 타이밍 다이어그램은 객체가 특정 상태에서 얼마나 오래 머무는지 명시

(출처: J. Schmuller, Teach Yourself UML in 24 Hours, 3rd Ed., Sams, 2004)

1.5 UML2.0의 새로운 다이어그램

(13) 패키지 다이어그램 (Package Diagram)

- 다이어그램들을 조직화 – 서브 시스템 표현
- 구성요소(클래스, 컴포넌트 등)들을 탭이 달린 폴더 안에 표현



(출처: J. Schmuller, Teach Yourself UML in 24 Hours, 3rd Ed., Sams, 2004)

1.6 왜 이렇게 다이어그램이 많을까?

- 왜 굳이 여러 가지 다이어그램을 사용할까?
  - 모든 참여자(stakeholder)를 만족시키기 위해서 다양한 다이어그램 사용
  - 좋은 시스템 설계는 모든 가능한 관점의 다이어그램이 포함되어 있어야 함
  - 각각의 UML 다이어그램은 자신이 나타내고 있는 관점을 하나로 통합하는 수단을 제공

1.7 요약

- 시스템 개발을 위한 이해하기 쉬운 표기 방식 필요
- UML은 시스템 개발 세계에서 표준으로 인정받은 표기 시스템
  - 시스템 분석가, 의뢰인, 프로그래머, 기타 참여자의 관점 제공
  - 각 시점에서 이해하는 다방면의 설계도 작성 표준
  - 제안된 그래픽 요소를 조합하여 다이어그램 작성
- UML 모델은 시스템이 “무엇(what)”을 의도하고 있는 지 말해줄 뿐, “어떻게(how)” 구현되는 지 말해주지 않는다.