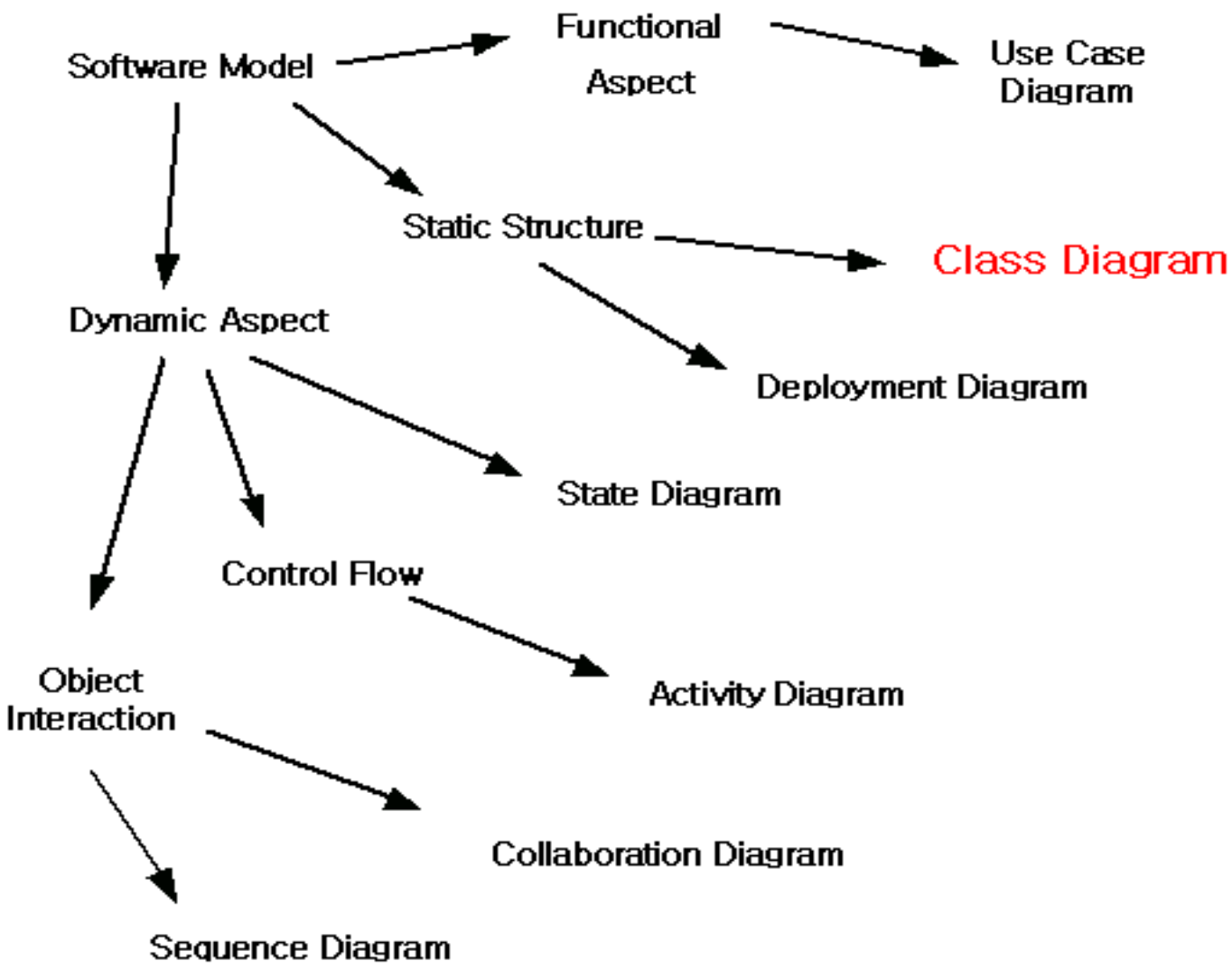
4장. UML
(Unified Modeling Language)

4.1 다이어그램 개요

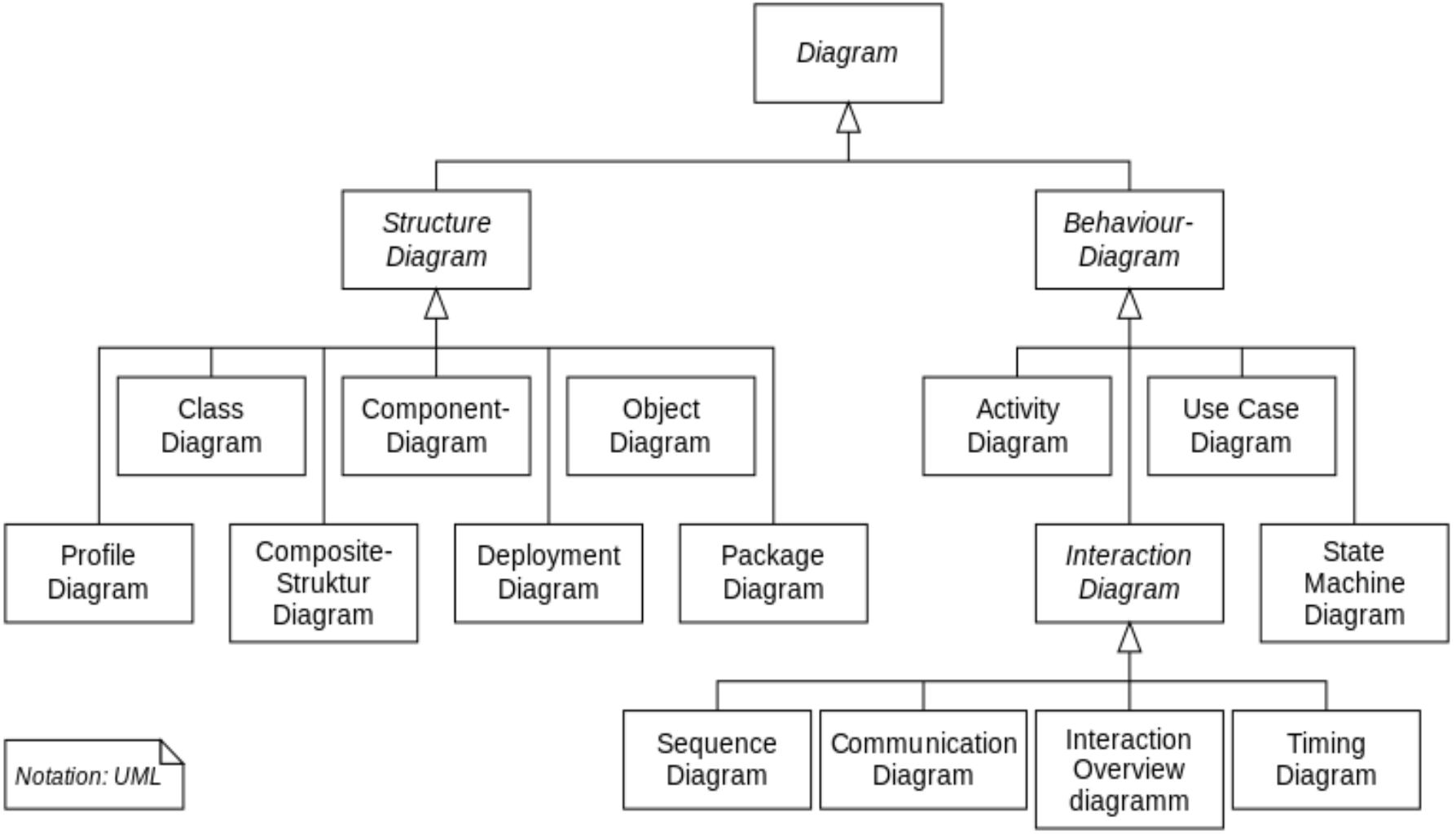
- UML에서는 시스템을 보는 다양한 관점을 각각 모델링하기 위해 다양한 다이어그램 정의
  - 논리적인 관점, 기능적인 관점, 동적/정적인 관점 등을 나타내어 시스템을 설계하기 위한 도구

  - ✓ Use Case Diagram
  - ✓ Class Diagram
  - ✓ Sequence Diagram
  - ✓ Collaboration Diagram (협동도)
  - ✓ State Diagram (상태도)
  - ✓ Activity Diagram (행위도)
  - ✓ Deployment Diagram (배치도)

UML 다이어그램이 지원하는 관점들



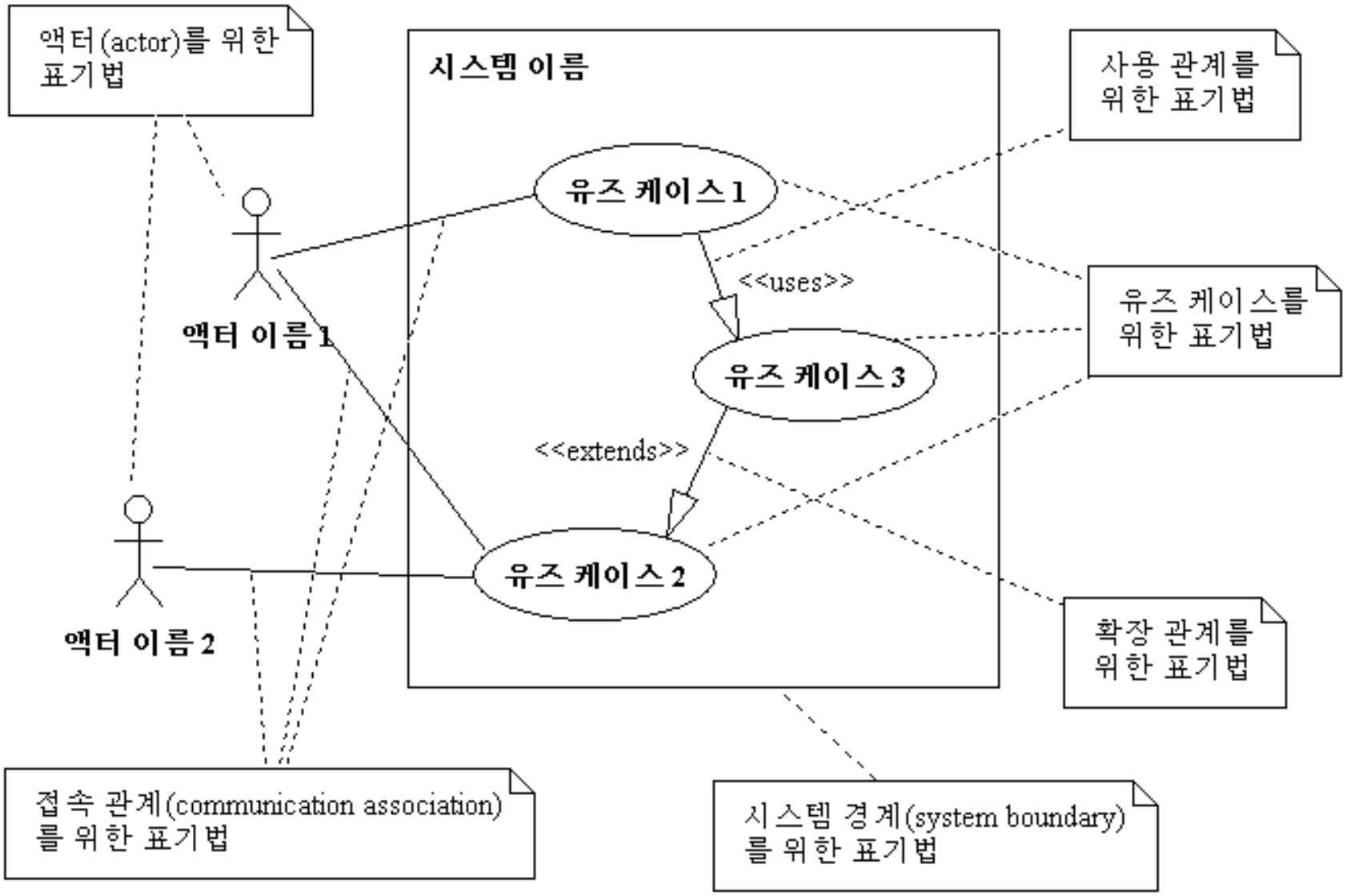
UML 다이어그램의 종류



http://en.wikipedia.org/wiki/File:UML_diagrams_overview.svg

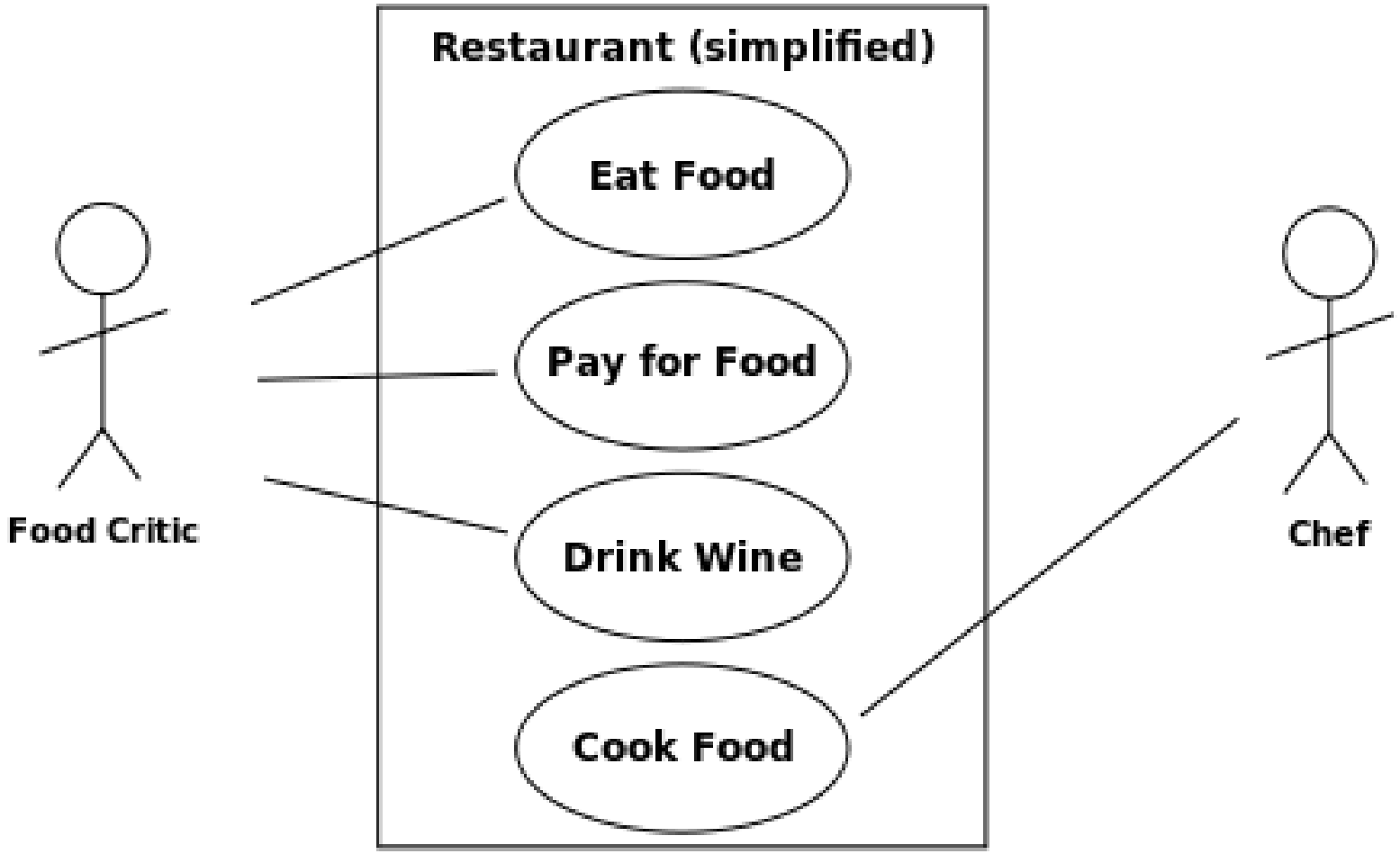
4.1.1 Use Case Diagram

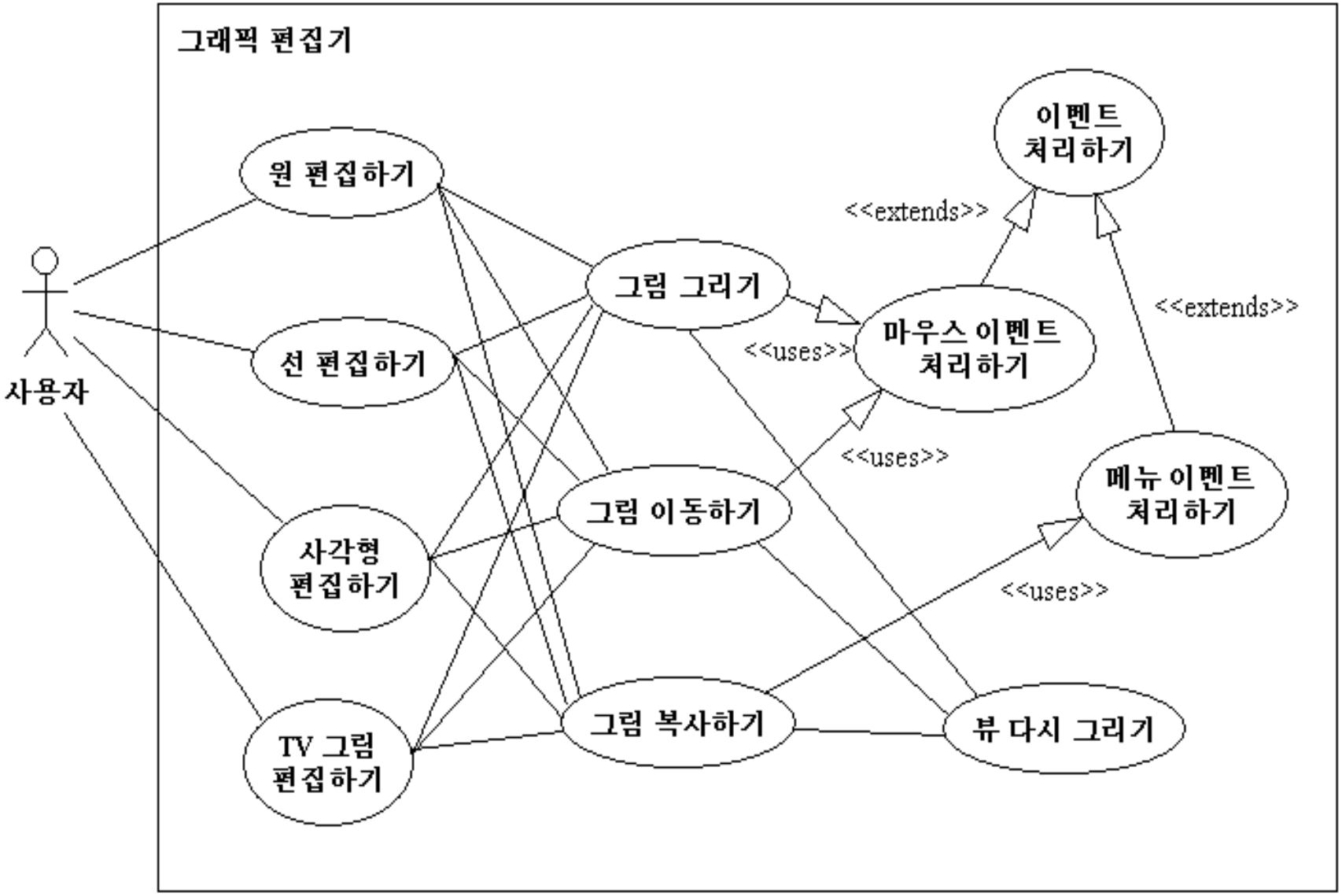
- 유스케이스 : “사용하는 경우”로 해석
- 기능적인 측면에서 사용자의 요구 사항을 명시하기 위한 것
  - 사용자가 구현될 시스템으로부터 어떠한 기능을 필요로 하는가?
  - 어떠한 기능이 수행되기 위해서 또 다른 어떤 기능을 필요로 하는가?
- 시스템 사용자가 시스템을 사용하는 경우를 모두 나열
  - 시스템 내부에서 프로세스 간 사용하는 경우도 포함
- 기능적 관점
- 분석용 다이어그램
- Jacobson의 영향
- 가장 많은 양을 작성해야 함



<유스케이스 다이어그램 표기법>

- 액터 (Actor)
  - 시스템의 서비스를 이용하는 외부 객체
- 유스케이스 (Use Case)
  - 시스템이 제공해야 할 개별적인 서비스 표현
- 시스템 경계 (System Boundary)
  - 어떤 시스템이 제공해야 하는 유스케이스들의 범위를 결정
- 접속 관계 (Communication Association)
  - 액터와 유스케이스 사이 또는 유스케이스 간에 연결되는 관계
  - 사용 관계 (Use Association)
  - 확장 관계 (Extends Association) : 기능 추가

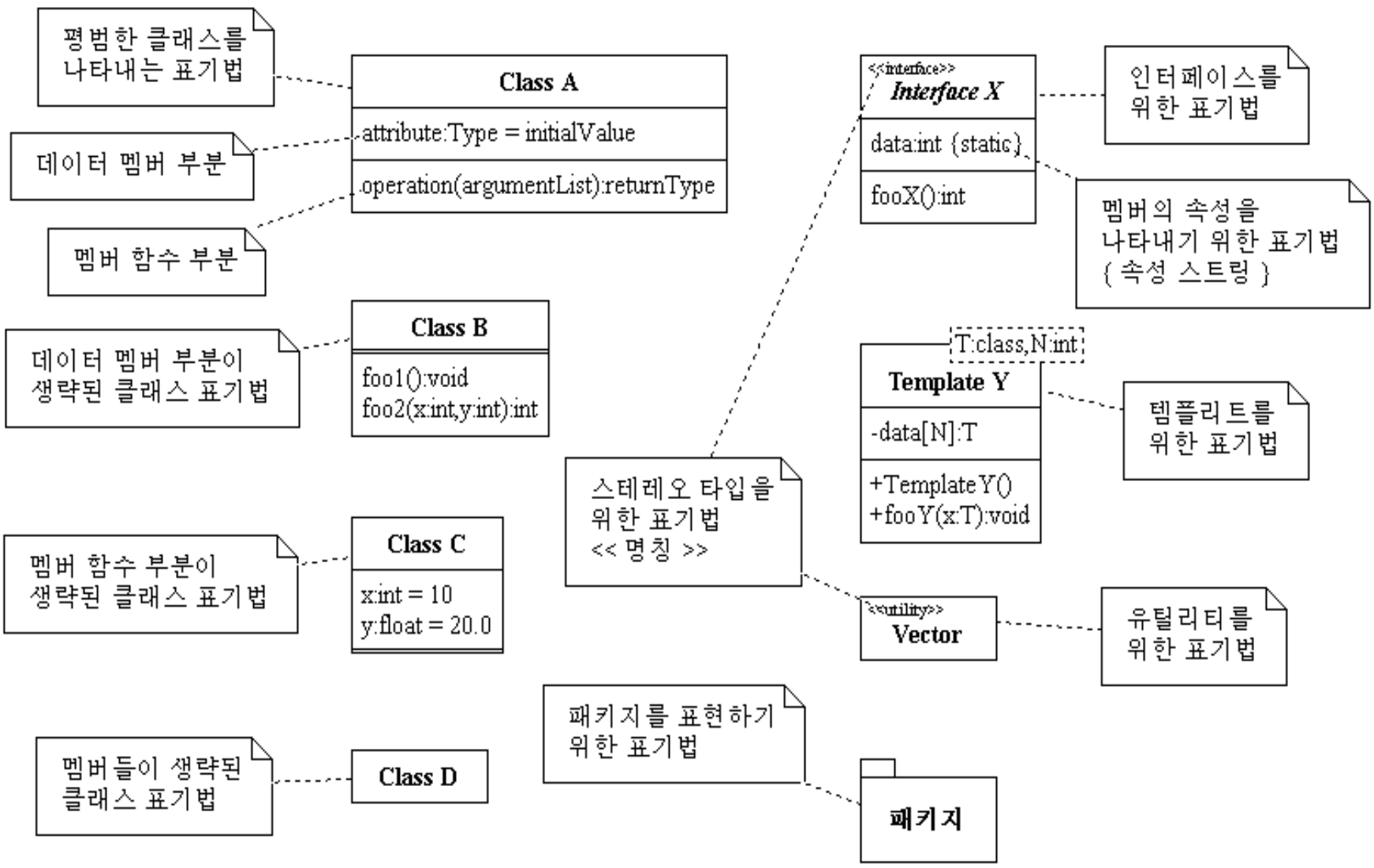

Restaurant (simplified)
Eat Food
Pay for Food
Drink Wine
Cook Food
Food Critic
Chef



<그래픽 편집기의 기능 기술을 위한 유스케이스 다이어그램의 예>

4.1.2 Class Diagram

- 객체들과 그들 간의 관계를 추상화한 모델을 기술
- 시스템 구성 요소에 관한 정적인 관점을 기술
  - 시간의 흐름이나 외부에서 발생하는 사건에 관계없이 항상 일정하게 해석될 수 있는 시스템의 특징을 기술
- 가장 중요한 설계 문서
- 전체 공정(분석, 설계, 구현 단계)에서 지속적으로 사용
- 프로세스가 진행될수록 상세한 클래스 다이어그램을 작성할 수 있어야 함



<클래스 다이어그램에서 클래스 관련 표기법>

- 클래스의 데이터 멤버 기술 문법

  <가시성> <변수명>:<타입> = <초기값> {<속성스트링>}

  - <가시성> : +, -, #
  - 변수명이 타입보다 더 중요함을 암시하기 위해 먼저 표기함

  - 예)

    value:int = 0

    +name:String = “Kim”

    -scale:float {transient}

    #ok:Boolean = true {static}

- 클래스의 멤버 함수 기술 문법

  <가시성> <함수명>(<인수 리스트>):<리턴타입> {<속성 스트링>}

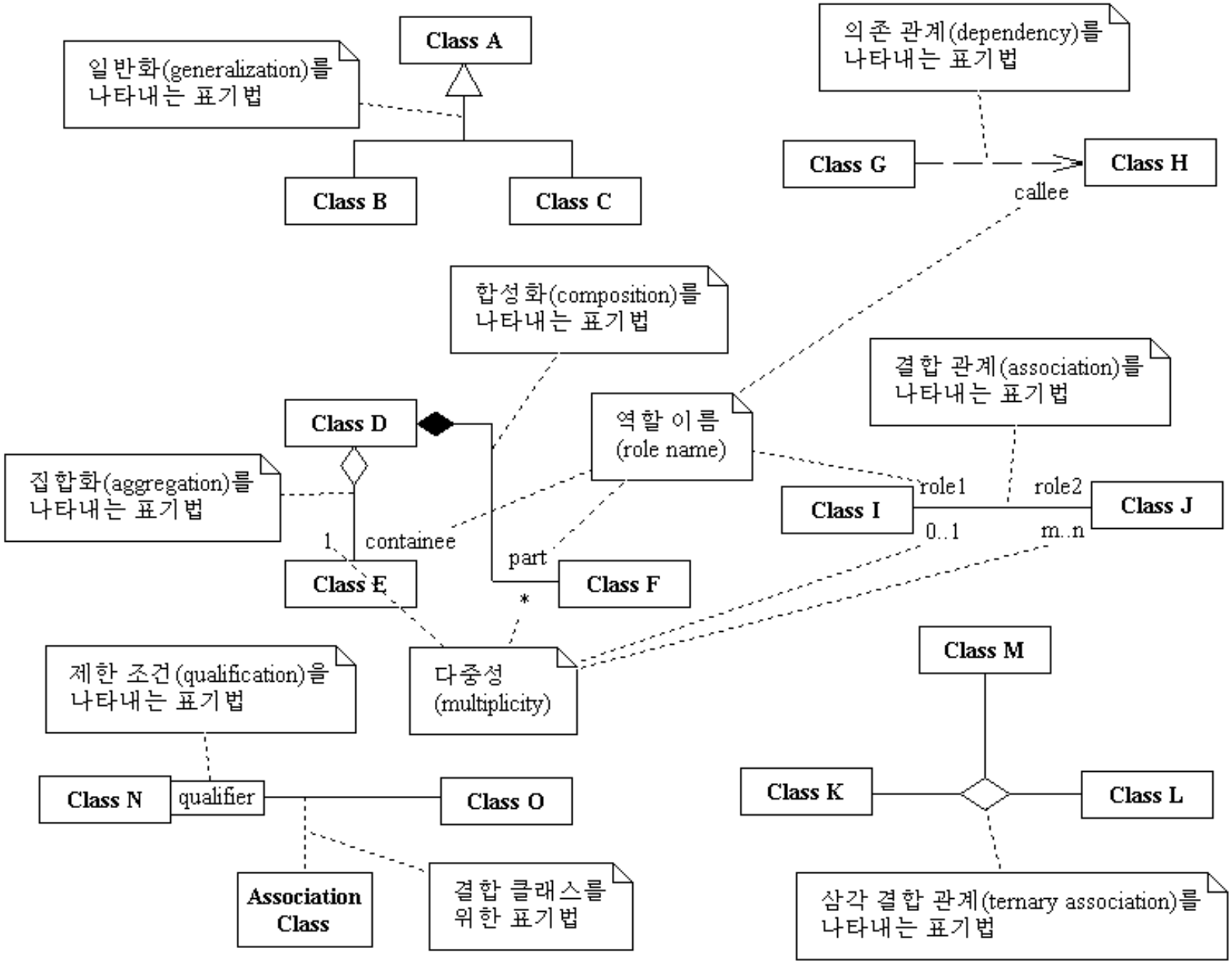
  - 예)

    Foo()

    +sort(data:Array):Array {static}

    -moveCoordinate(dx:int, dy:int):void

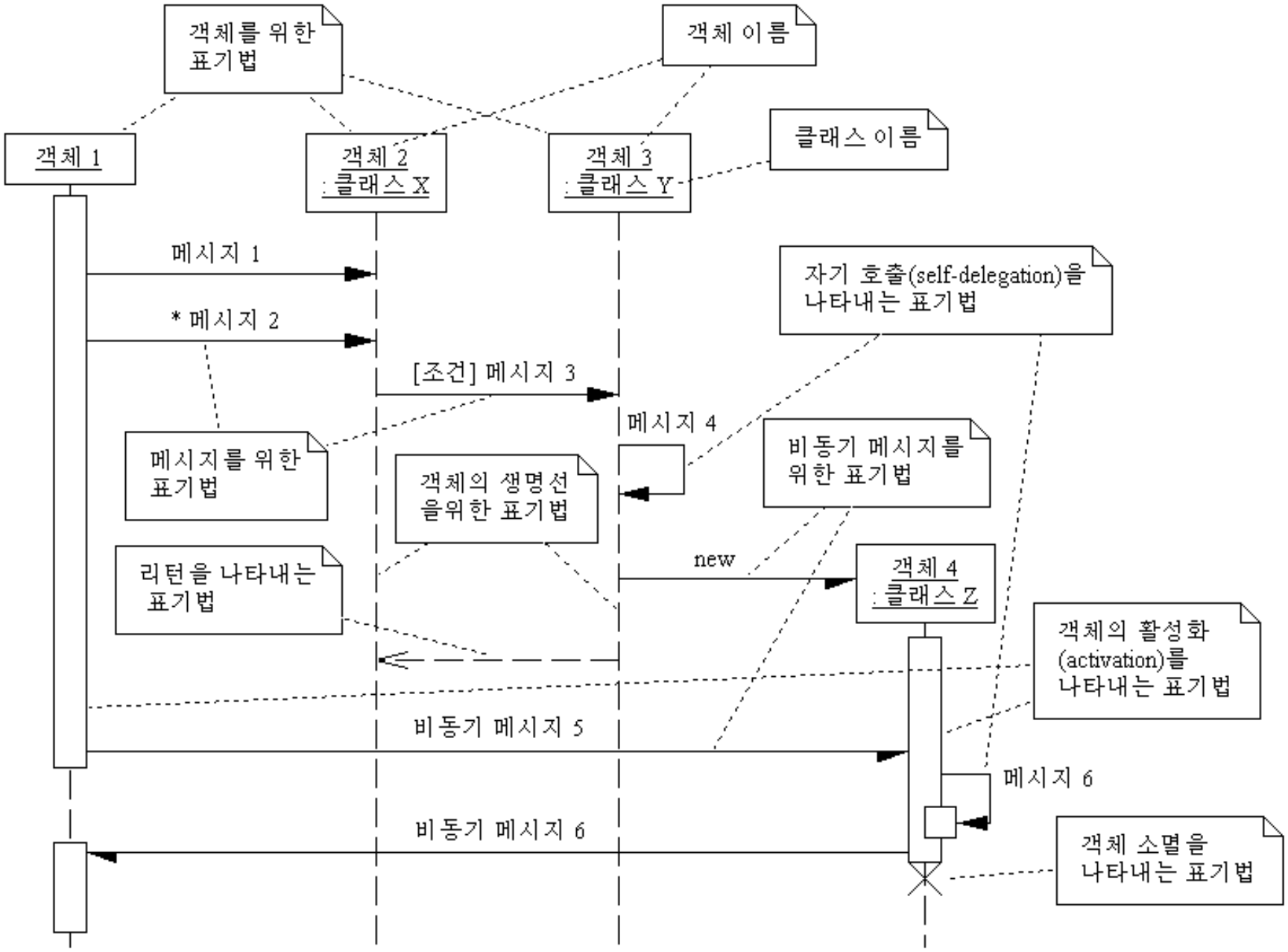
    #copy():Figure {virtual}



<클래스 다이어그램에서 관계 관련 표기법>

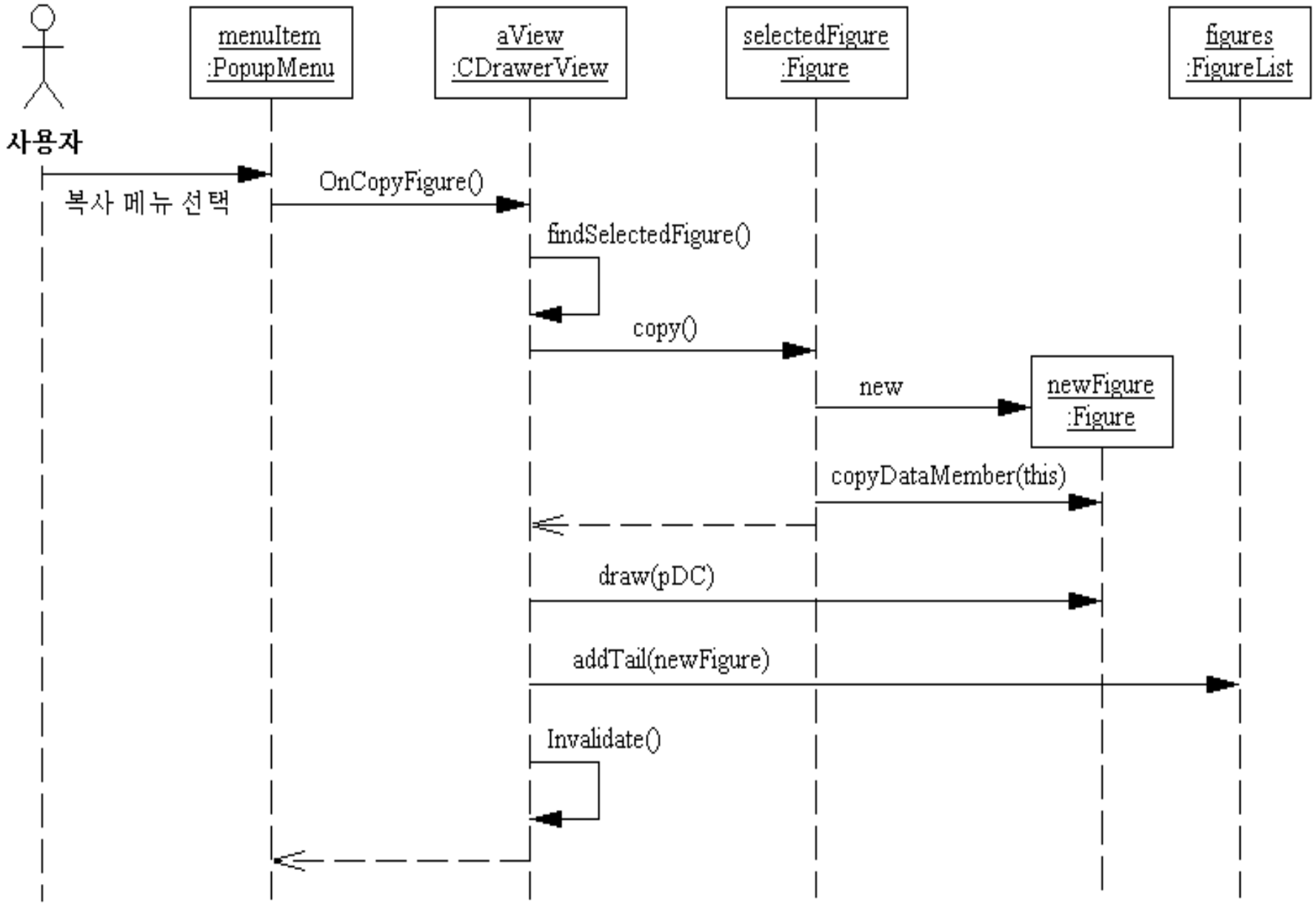
4.1.3 Sequence Diagram

- 시간의 흐름에 따라 시스템의 객체들이 어떠한 순서로 메시지를 주고 받는가를 모델링한 것
  - 메시지를 이용한 객체 간의 대화
- 동적인 관점
- 하나의 유스케이스에 대한 시나리오 역할
  - 어떤 객체들이 필요하며, 객체들이 어떤 서비스를 제공해야 하는지 구체적으로 인식케 됨
- 목적
  - 서비스 제공을 위한 시나리오를 명확하게 정의하고자 함
  - 클래스, 함수 이름 추출하여 클래스 다이어그램의 자원을 좀더 자세히 명시하고자 함
- 최근 CASE 도구들은 역공학 기능 제공
  - UNIX에서 cflow 도구가 제공하는 함수 호출 그래프 생성 기능과 유사



<시퀀스 다이어그램의 표기법>

- 완전한 화살표
  - 일반 메시지
- 반쪽 화살표
  - 비동기 메시지 (asynchronous message)
- 자기 호출 (Self-delegation)
  - 객체 자신에게 보내는 메시지
- 메시지 내용 기술 방식
  - 함수 호출 방식으로 표현
  - 약간의 표기법 추가
    - * : 그 메시지가 반복적으로 보내짐을 의미
    - [...] : 메시지가 보내지기 위한 조건
- 생명선 (Lifelane)
  - 객체 밑에 점선으로 표시되는 수직선
  - 객체 간의 상호작용 중에서 그 객체가 살아있음을 표현
- 활성화 (Activation)
  - 생명선 위에 긴 사각형 모양의 표기법
  - 객체의 메소드가 실행 중임을 명확하게 표현하기 위해 사용

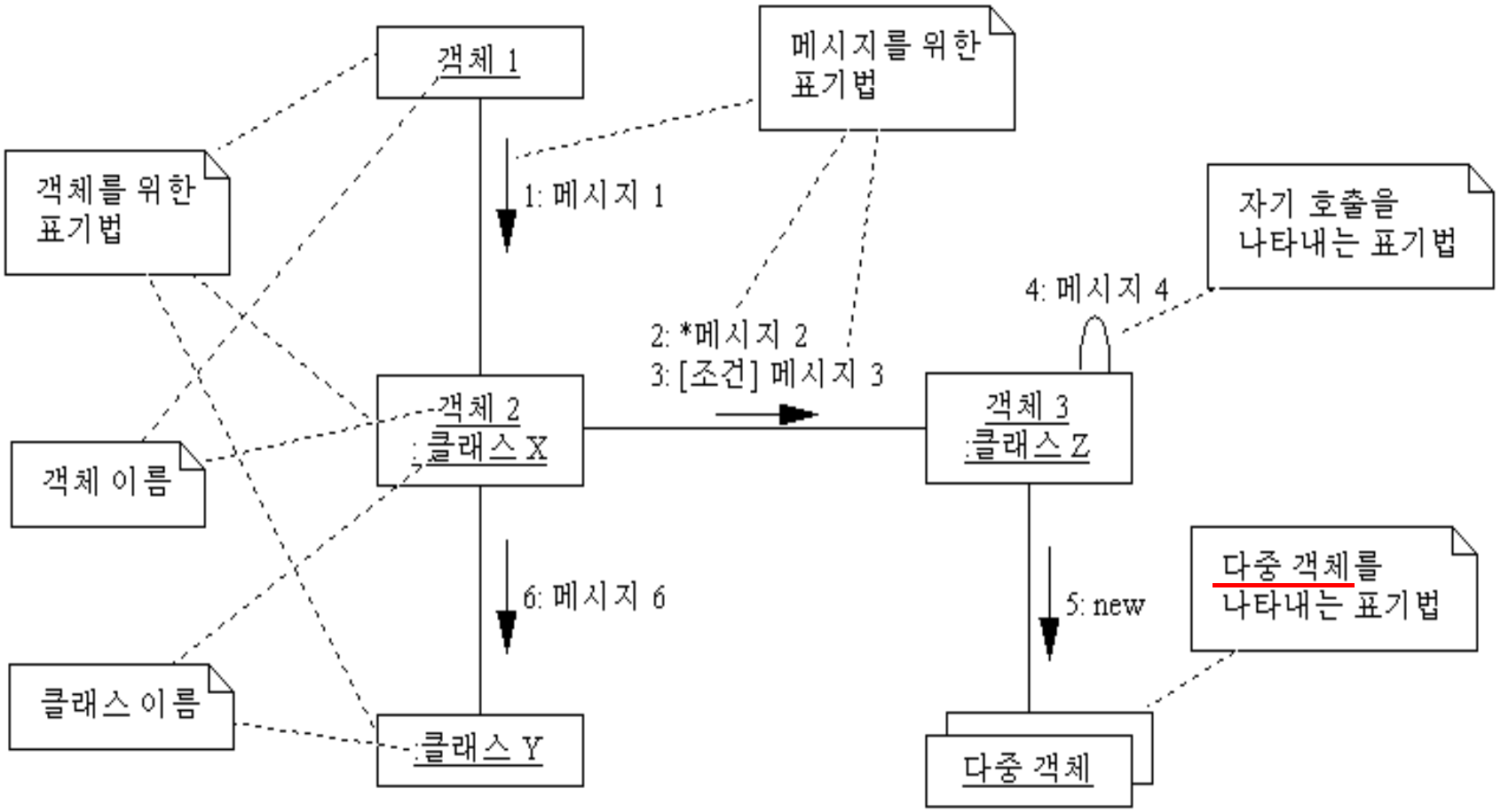


<시퀀스 다이어그램 작성 예>

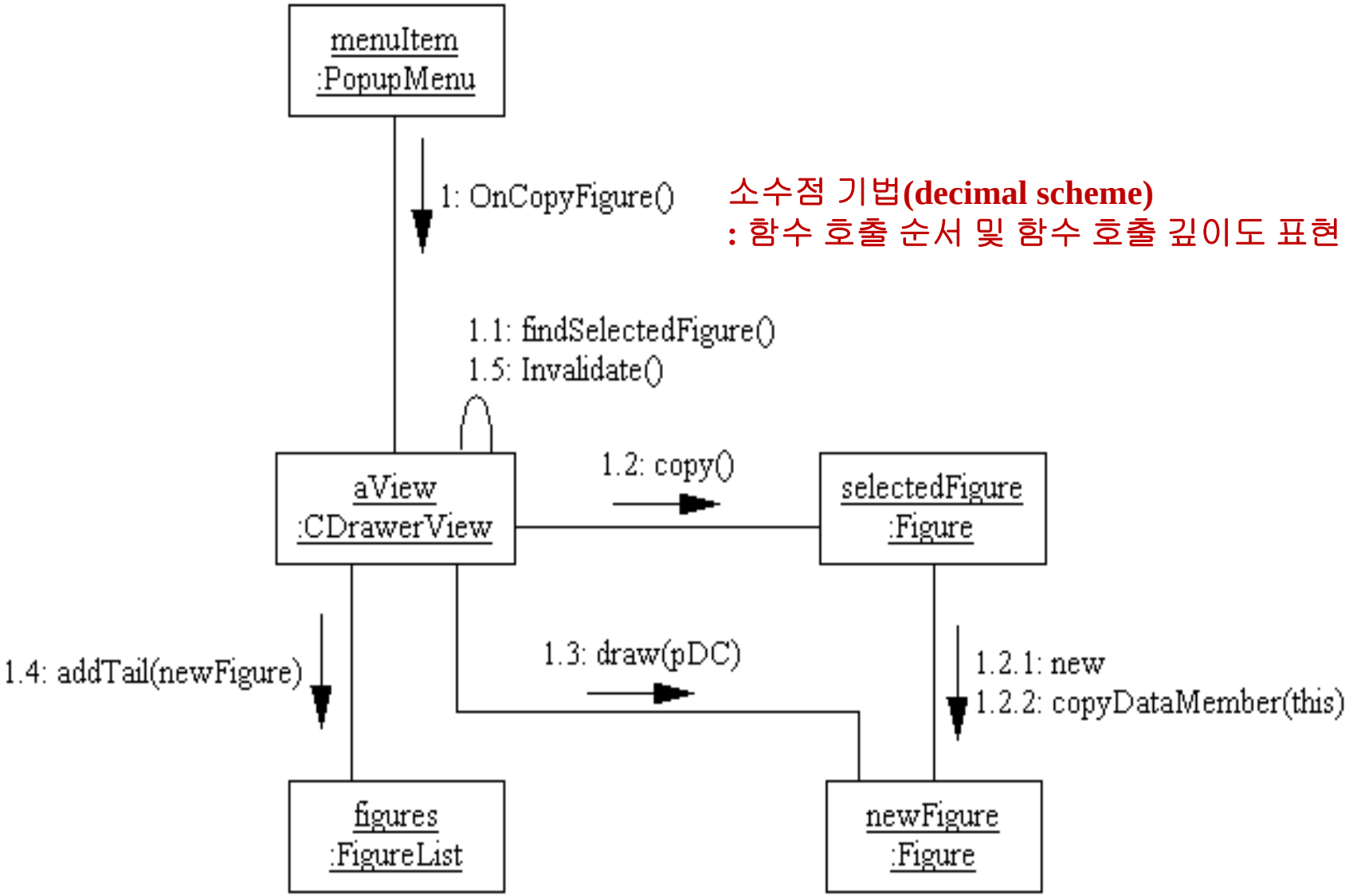
그래픽 에디터에서 하나의 그림 객체를 복사하는 과정

4.1.4 Collaboration Diagram

- 사용 목적과 내용이 시퀀스 다이어그램과 거의 동일
- Sequence Diagram의 또다른 표현 방식
- 동적인 관점
- 시간적인 측면 보다는 상호 작용에 대한 표현에 중점을 둠
  - 시퀀스 다이어그램
    - 생명선을 이용해 시간의 흐름에 따른 객체간의 상호 작용이 강조되어 표현됨
    - 메시지 순서가 확연히 드러남
  - 협동도
    - 객체들이 평면적으로 배치되며 실행 순서를 나타내는 번호를 이용함
    - 표기법의 종류가 시퀀스 다이어그램보다 적고, 더 간결하게 작성됨
- 설계자의 기호에 따라 위의 둘 중 하나를 작성하면 됨


객체 1
메시지를 위한
표기법
객체를 위한
표기법
1: 메시지 1
자기 호출을
나타내는 표기법
4: 메시지 4
2: *메시지 2
3: [조건] 메시지 3
객체 2
:클래스 X
객체 3
:클래스 Z
객체 이름
6: 메시지 6
5: new
다중 객체를
나타내는 표기법
클래스 이름
:클래스 Y
다중 객체

<협동도 작성 예>



소수점 기법**(decimal scheme)**
: 함수 호출 순서 및 함수 호출 깊이도 표현 가능

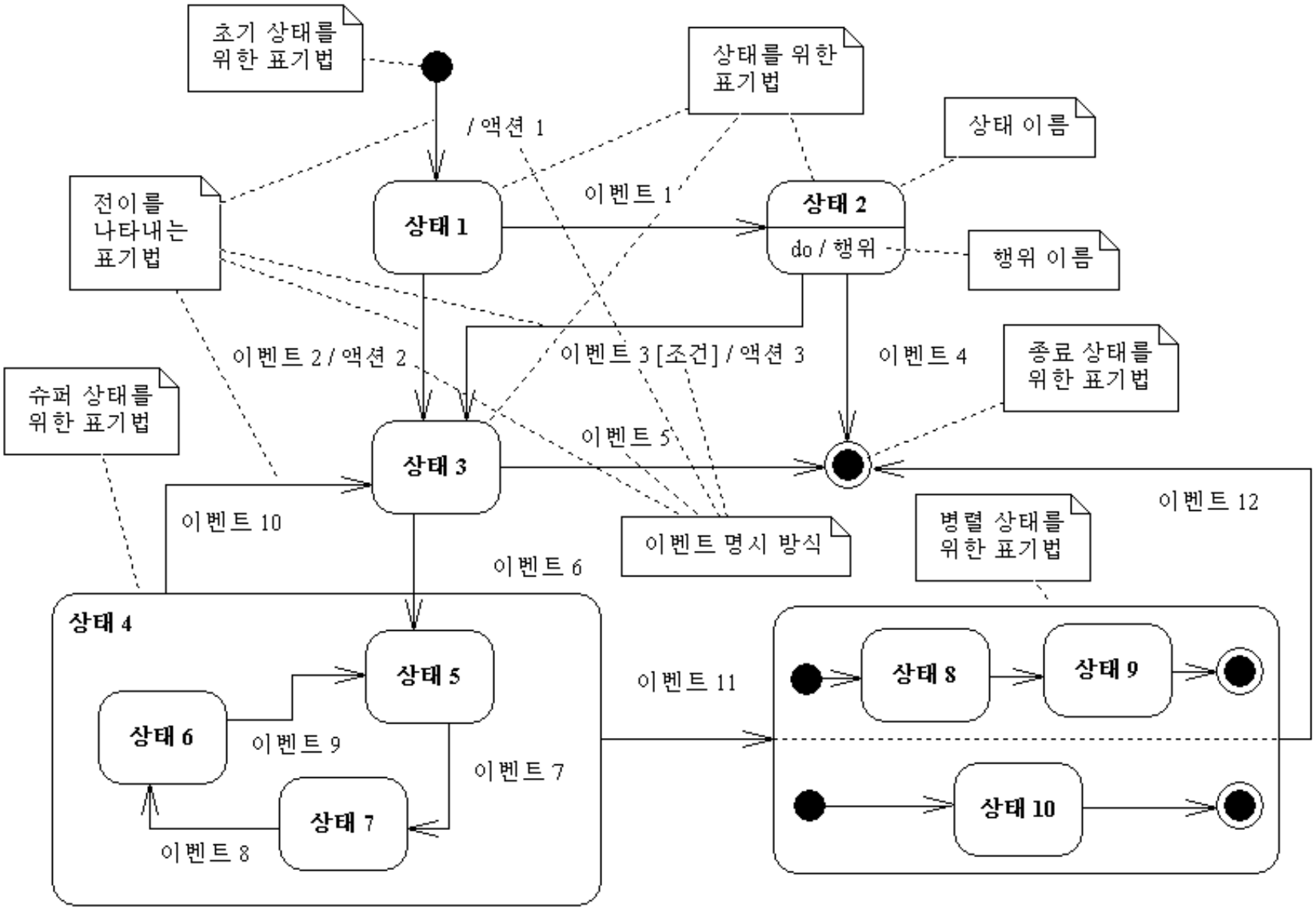
<협동도 표기법>

앞 시퀀스 다이어그램 작성 예와 동일한 내용

4.1.5 State Diagram

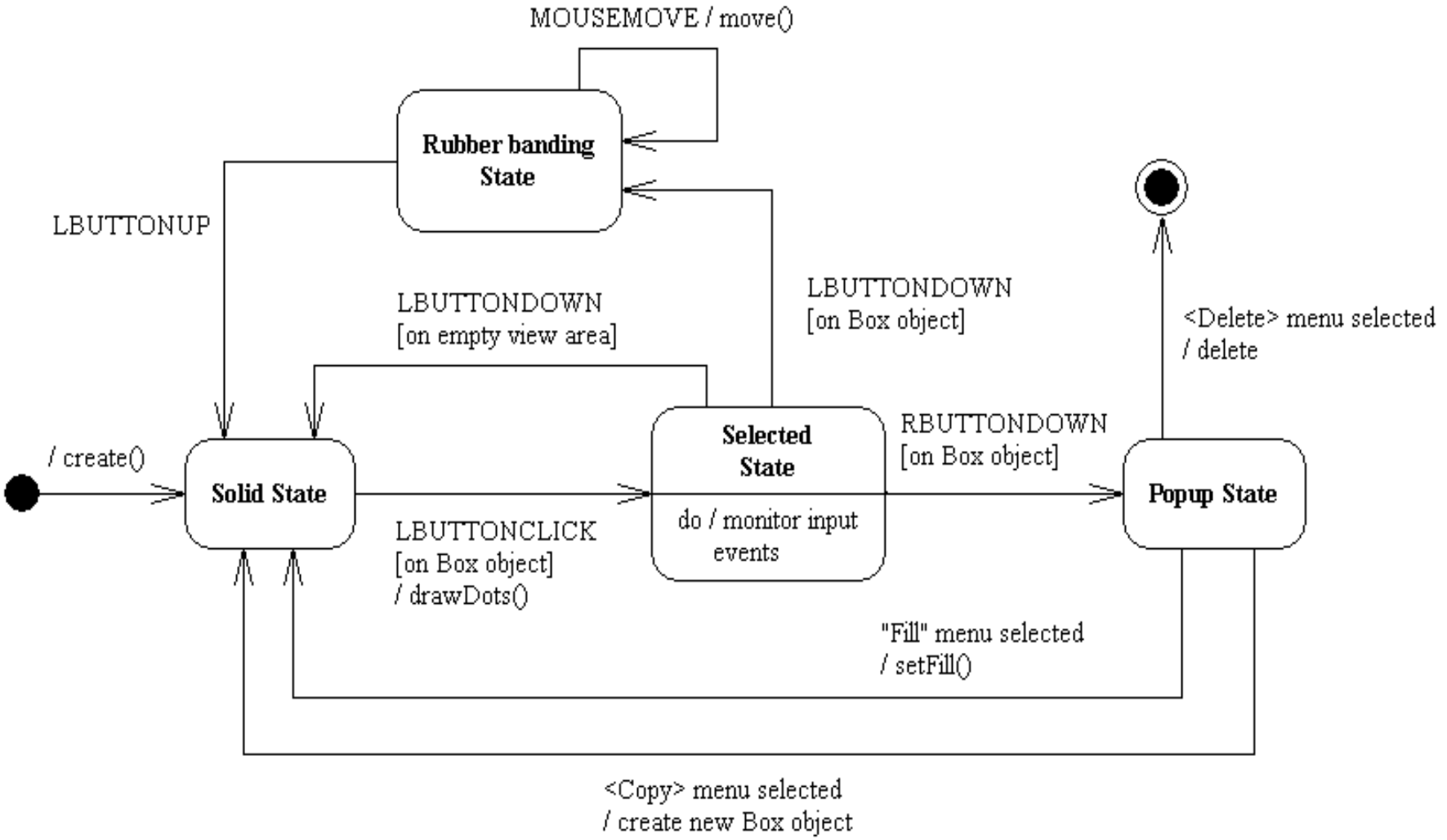
- 동적인 관점
- 객체의 상태 변화를 모델링
- 객체의 상태 변화를 야기하는 외부 이벤트에 초점
- 한 객체의 상태에 초점
  - 몇 가지 객체 상태가 필요한가?
    → 클래스의 상태 변수
  - 상태 변화 시에 수행해야 하는 서비스는 무엇인가?
    → 클래스 멤버 함수의 시그너처

  → 클래스 다이어그램의 상세 설계를 이루기 위한 중요 자원
- 시퀀스 다이어그램
  - 시간에 초점
  - 객체들간의 상호 작용에 초점


초기 상태를
위한 표기법
/ 액션 1
상태를 위한
표기법
상태 이름
전이를
나타내는
표기법
상태 1
이벤트 1
상태 2
do / 행위
행위 이름
이벤트 2 / 액션 2
이벤트 3 [조건] / 액션 3
이벤트 4
종료 상태를
위한 표기법
슈퍼 상태를
위한 표기법
상태 3
이벤트 5
이벤트 10
이벤트 명시 방식
병렬 상태를
위한 표기법
이벤트 12
이벤트 6
상태 4
상태 5
상태 6
이벤트 9
이벤트 7
상태 7
이벤트 8
이벤트 11
상태 8
상태 9
상태 10

<상태도 표기법>

- 상태(state)
  - 시작 상태(start state) : 채워진 원
  - 종료 상태(end state) : 두 개의 원으로 표기
  - 슈퍼 상태(superstate)
    - 여러 개의 상태를 묶어서 하나의 상태인 것처럼 표기한 것
    - 상태의 수가 늘어나는 것을 계층적인 형태로 제어하기 위함
    - 예) 상태 4
  - 병렬 상태(concurrent state)
    - 예) 상태 8, 상태 9, 상태 10
- 전이(transition)
  - 객체의 상태가 외부 이벤트의 발생에 따라 변해 가는 상황을 표현
  - 문법 : <이벤트> [<조건>] / <액션>
    - RBUTTONDOWN [selectedFigure != NULL] / display selected popup menu

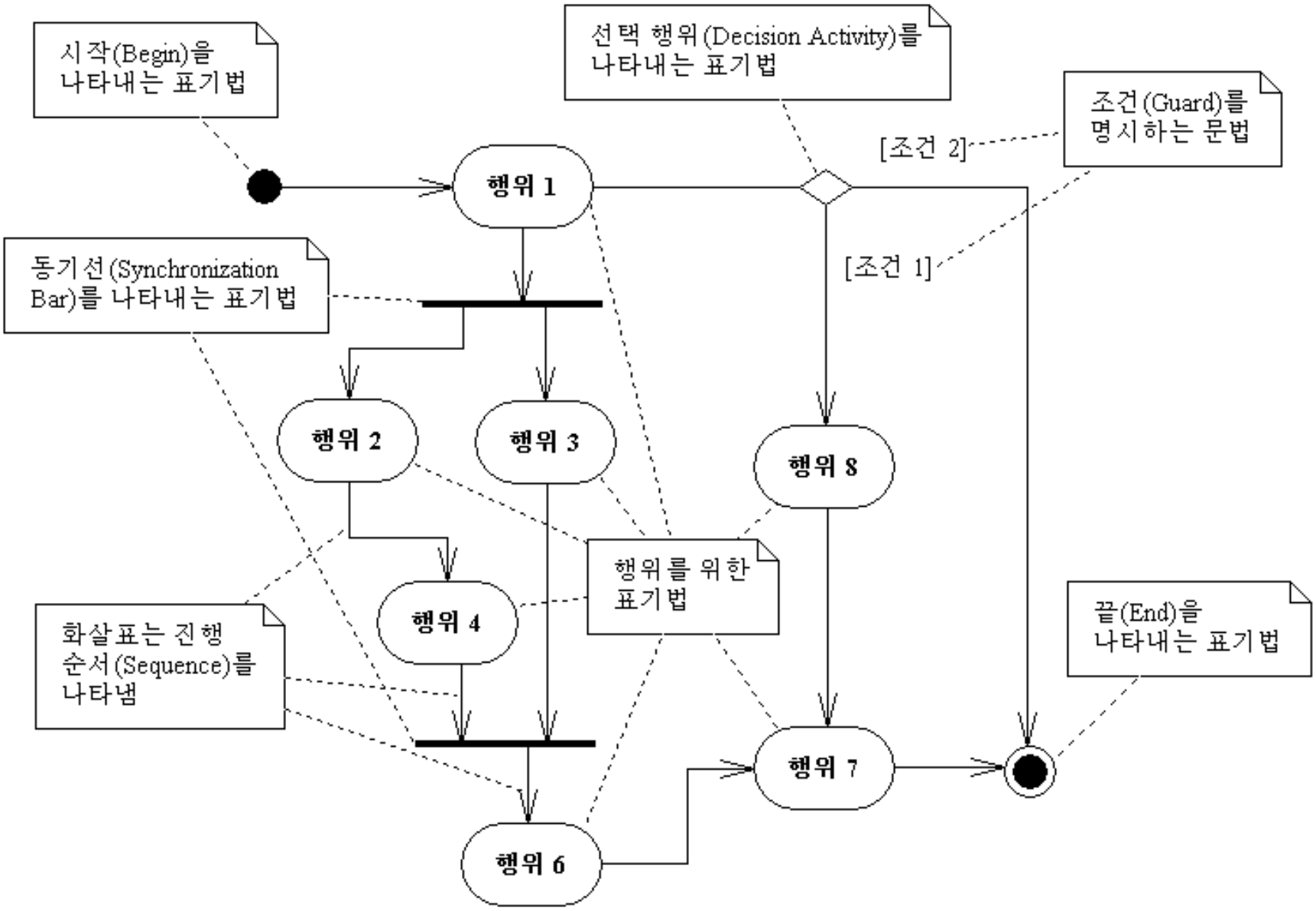

MOUSEMOVE / move()
Rubber banding State
LBUTTONUP
LBUTTONDOWN [on empty view area]
LBUTTONDOWN [on Box object]
<Delete> menu selected / delete
/ create()
Solid State
Selected State
do / monitor input events
RBUTTONDOWN [on Box object]
Popup State
LBUTTONCLICK [on Box object] / drawDots()
"Fill" menu selected / setFill()
<Copy> menu selected / create new Box object

<상태도 작성 예>

4.1.6 Activity Diagram

- 동적인 관점
- 시스템의 제어 흐름을 표현하기 위한 것
  - 워크플로우나 병렬 프로세스 표현 시 유용
  - 거시적 관점: 사람/컴퓨터 태스크의 작업 흐름
  - 미시적 관점: 클래스 메소드의 제어 흐름

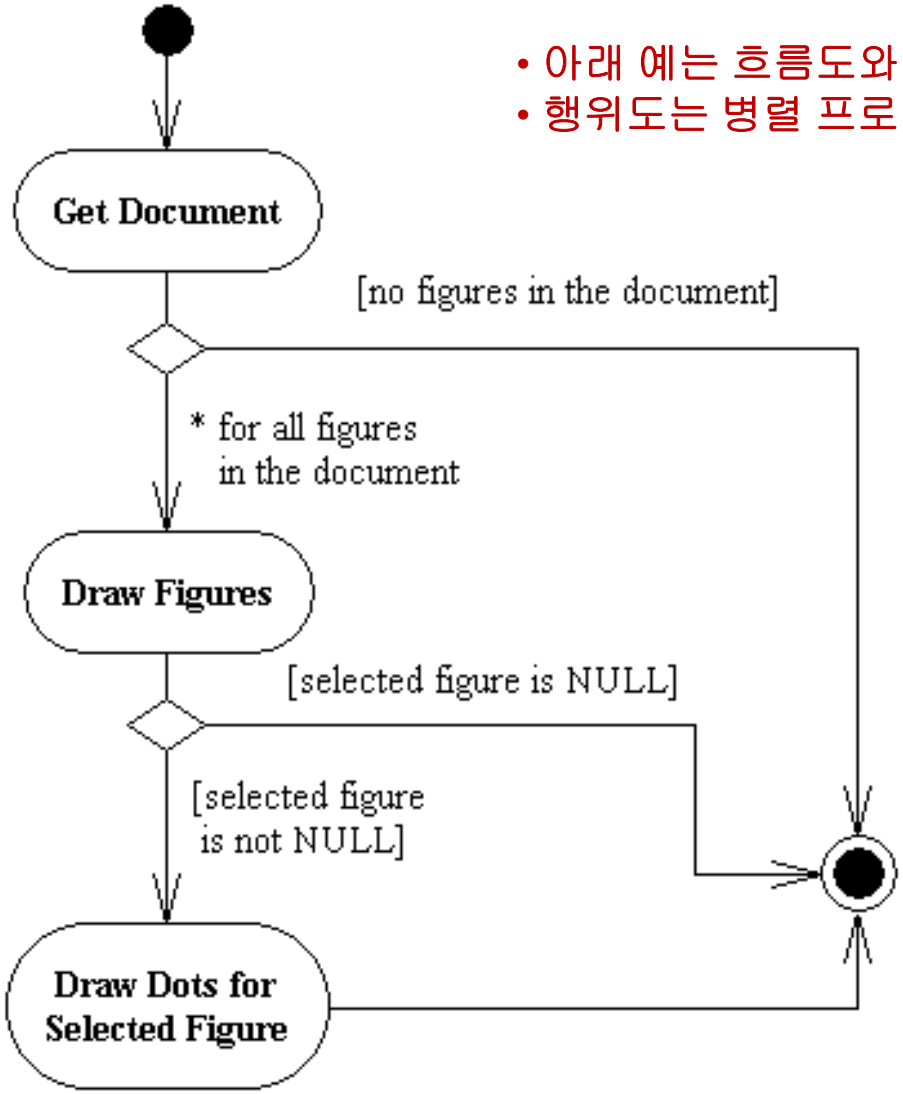
- 기존 표기법의 짬뽕
  - 다소 체계적이지 못함
  - Flowchart, Petri net, SDL


시작(Begin)을
나타내는 표기법
선택 행위(Decision Activity)를
나타내는 표기법
조건(Guard)를
명시하는 문법
[조건 2]
[조건 1]
행위 1
동기선(Synchronization
Bar)를 나타내는 표기법
행위 2
행위 3
행위 8
행위를 위한
표기법
행위 4
화살표는 진행
순서(Sequence)를
나타냄
끝(End)을
나타내는 표기법
행위 7
행위 6

<행위도 표기법>

- 동기선(synchronization bar)
  - 행위의 흐름이 병행적으로 발생하는 경우 표현
  - 예) “행위 1” 이후 2개의 행위가 병렬적으로 수행됨
    - “행위 6은” “행위 4”와 “행위 3”이 모두 끝난 후에 실행됨
- 선택 행위(decision activity)
  - 마름모로 표기
  - 제어의 흐름이 조건에 따라 선택적으로 갈라짐

- 행위도에서 표현하고자 하는 제어의 흐름은 다른 UML 다이어그램에서는 표현할 수 없는 정보

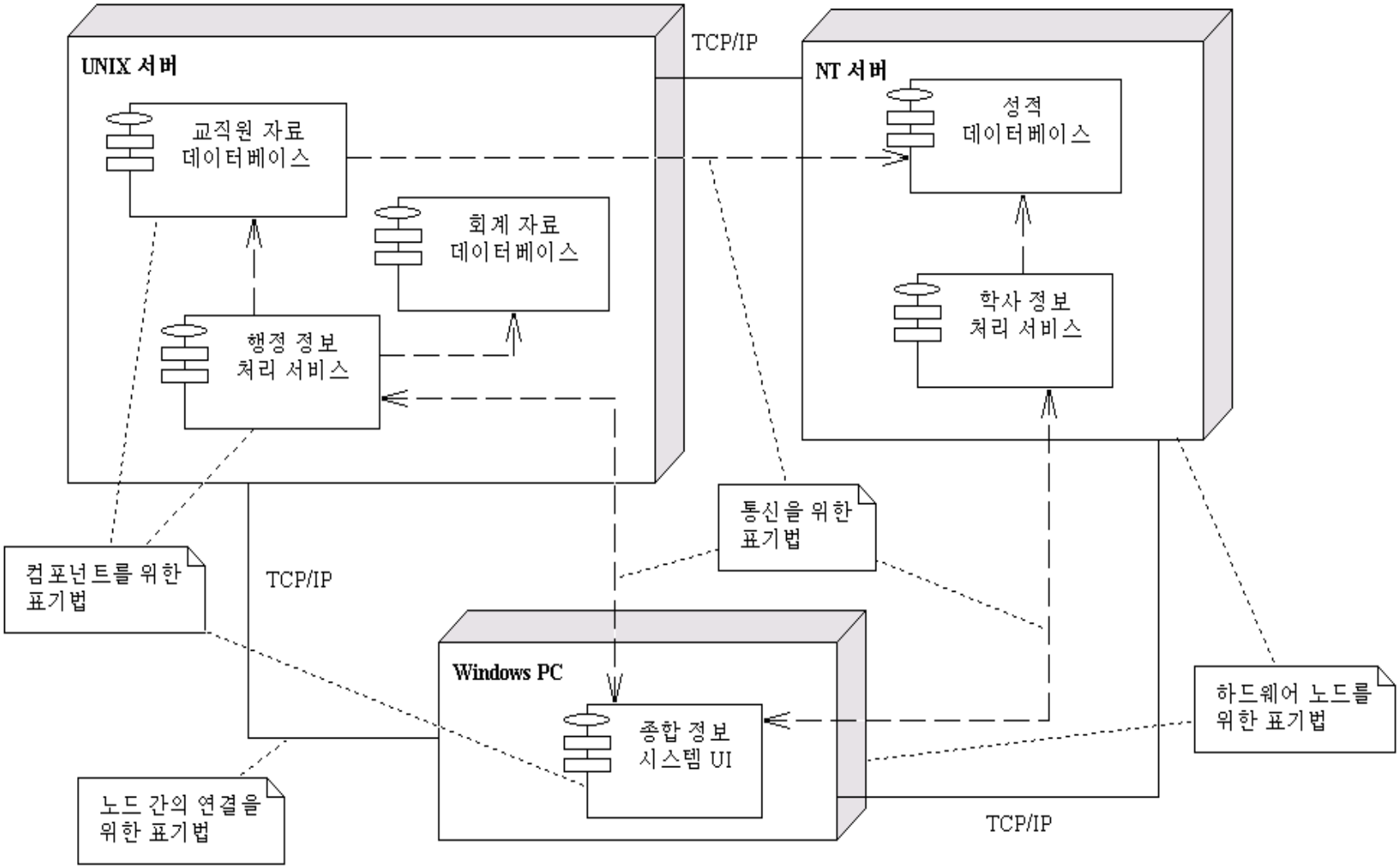


- 아래 예는 흐름도와 거의 유사하므로 효용가치 없음
- 행위도는 병렬 프로세스 모델링 시에 유용

<행위도 작성 예>

4.1.7 Deployment Diagram

- 정적인 관점: H/W, S/W 자원의 배치
- 분산 환경으로 구현되는 시스템의 H/W와 S/W 컴포넌트들이 물리적으로 어떠한 관계를 맺으며 배치되는가를 표현
- 굳이 표준화된 표기법을 사용할 필요는 없음
  - 만화, 아이콘, 이미지 등 사용 가능
- 개략 설계
- UML의 다른 다이어그램과 연관성이 적은 모델


UNIX 서버
교직원 자료
데이터베이스
회계 자료
데이터베이스
행정 정보
처리 서비스
TCP/IP
NT 서버
성적
데이터베이스
학사 정보
처리 서비스
통신을 위한
표기법
컴포넌트를 위한
표기법
TCP/IP
Windows PC
종합 정보
시스템 UI
하드웨어 노드를
위한 표기법
노드 간의 연결을
위한 표기법
TCP/IP

<배치도 표기법 및 작성 예>

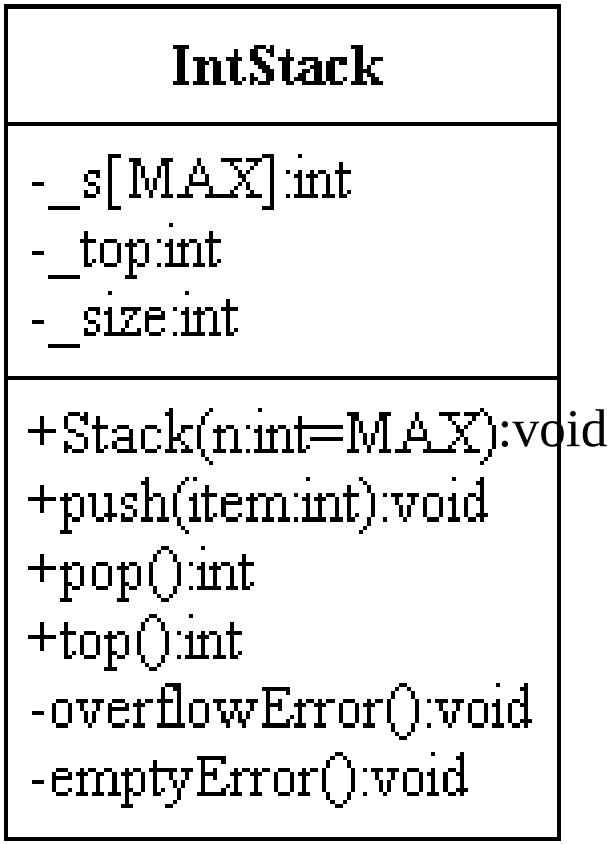
- 하드웨어 노드(node)
  - 커다란 사각형으로 표기
  - 예) 메인프레임 컴퓨터, PC 등
- 연결(connection)
  - 노드 사이의 실선으로 표기
  - 하드웨어 간 통신 프로토콜

- 의존 관계 or 통신 관계
  - 점선의 화살표로 표기
  - 소프트웨어들이 상호 간 서비스를 이용하는 상황 표현

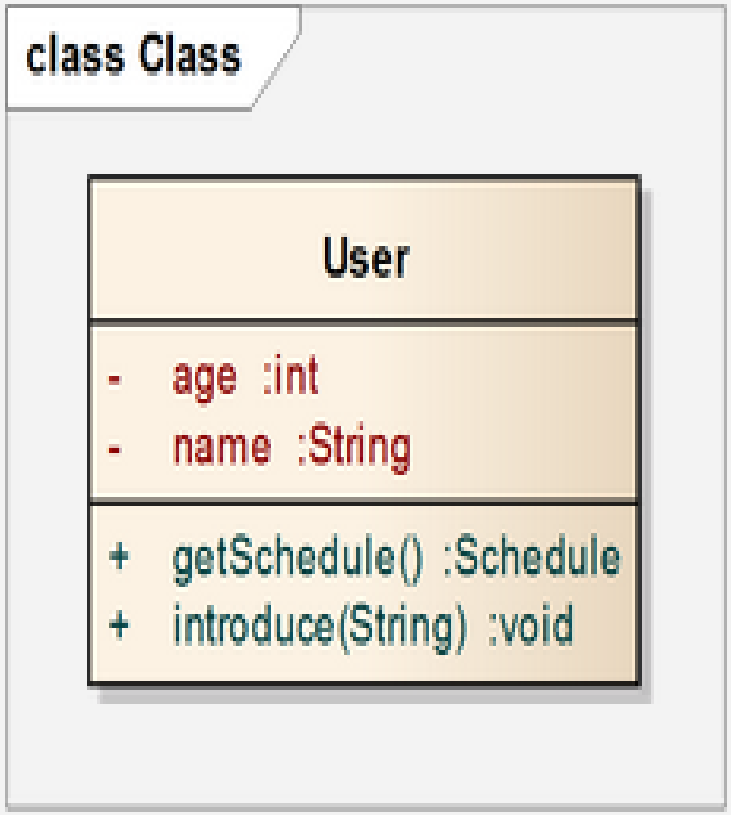
4.2 클래스 다이어그램의 이용법

- 클래스 다이어그램의 중요성
  - 구현 코드와 가장 밀접
  - 분석 단계부터 구현 단계까지 개념적인 통일성 제공
  - 솔기없는 프로세스를 가능케 함

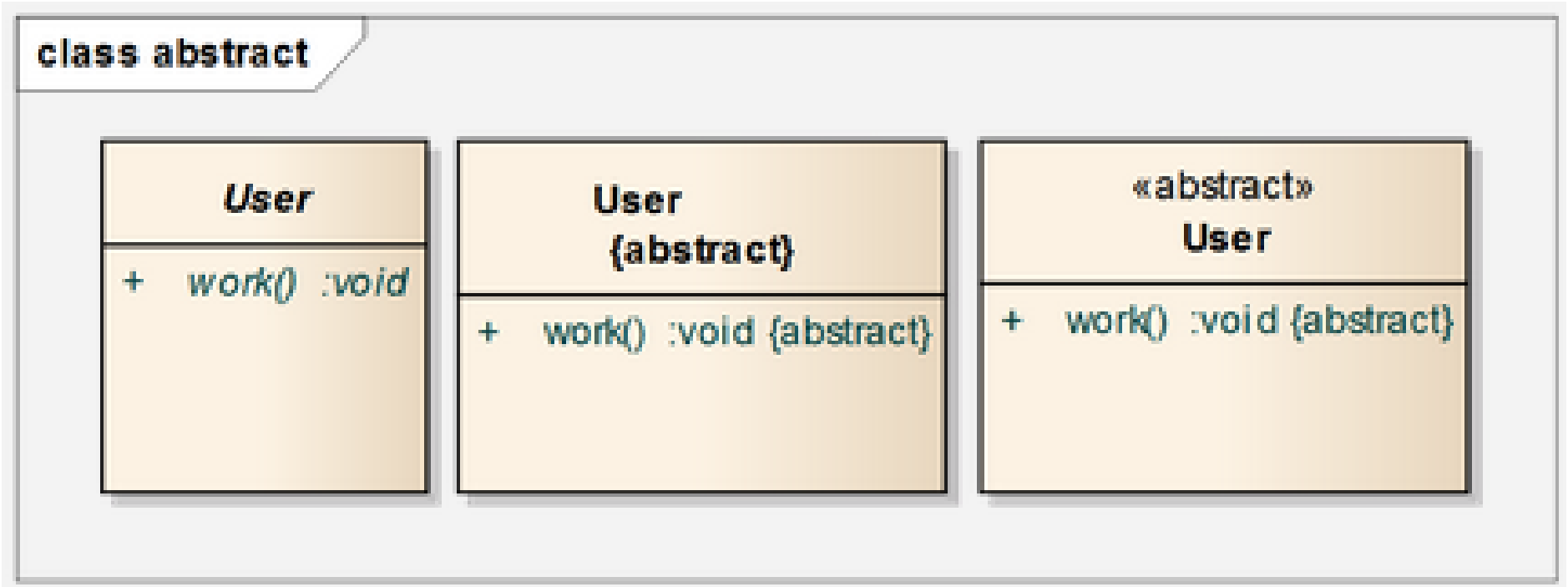
4.2.1 클래스 표기법의 이용



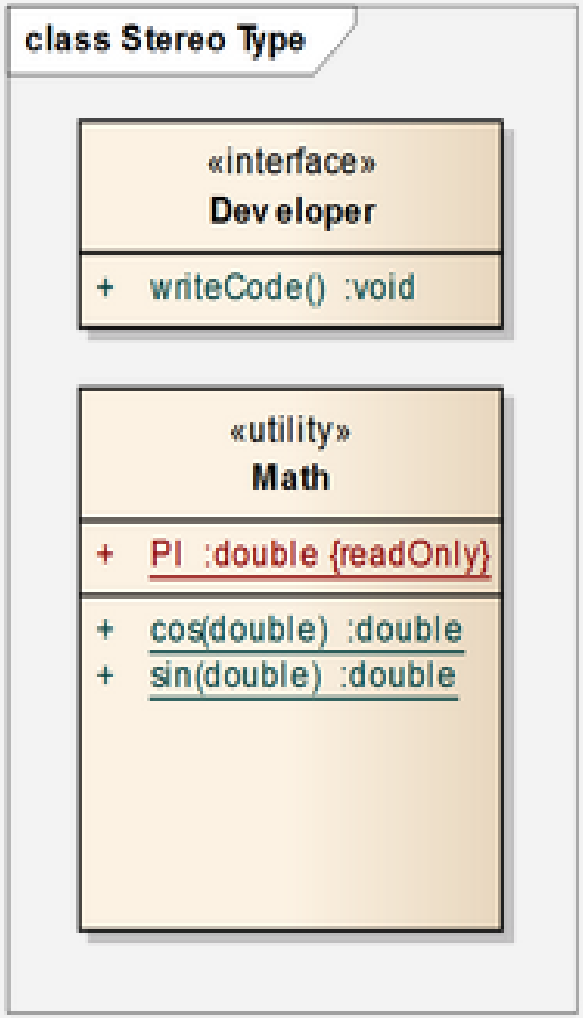

```
class IntStack {
// Attributes
private:
    int _s[MAX];
    int _top;
    int _size;

// Operations
private:
    void overflowError();
    void emptyError();
public:
    void Stack(int n=MAX);
    void push(int item);
    int pop();
    int top();
};
```




```
4
5 public class User {
6     private int age;
7     private String name;
8
9     public Schedule getSchedule() {
10        // 스케쥴을 본다.
11        return null;
12    }
13    public void introduce(String introduce) {
14        // 자기소개를 한다.
15    }
16 }
17
```




```
2
3 public abstract class User {
4
5     public abstract void work();
6 }
```

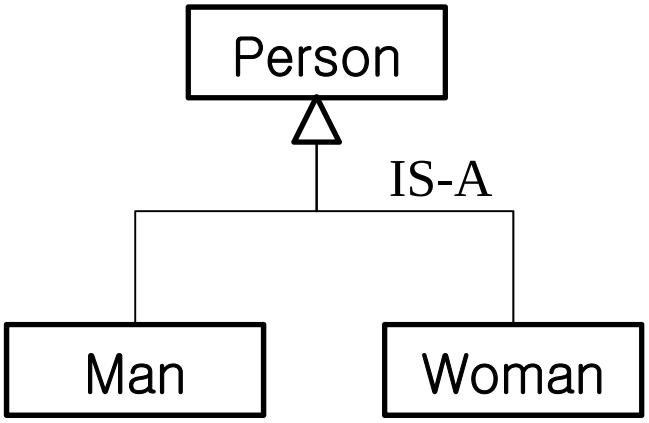



```
2
3 public interface Developer {
4
5     public void writeCode();
6 }
7
8
```

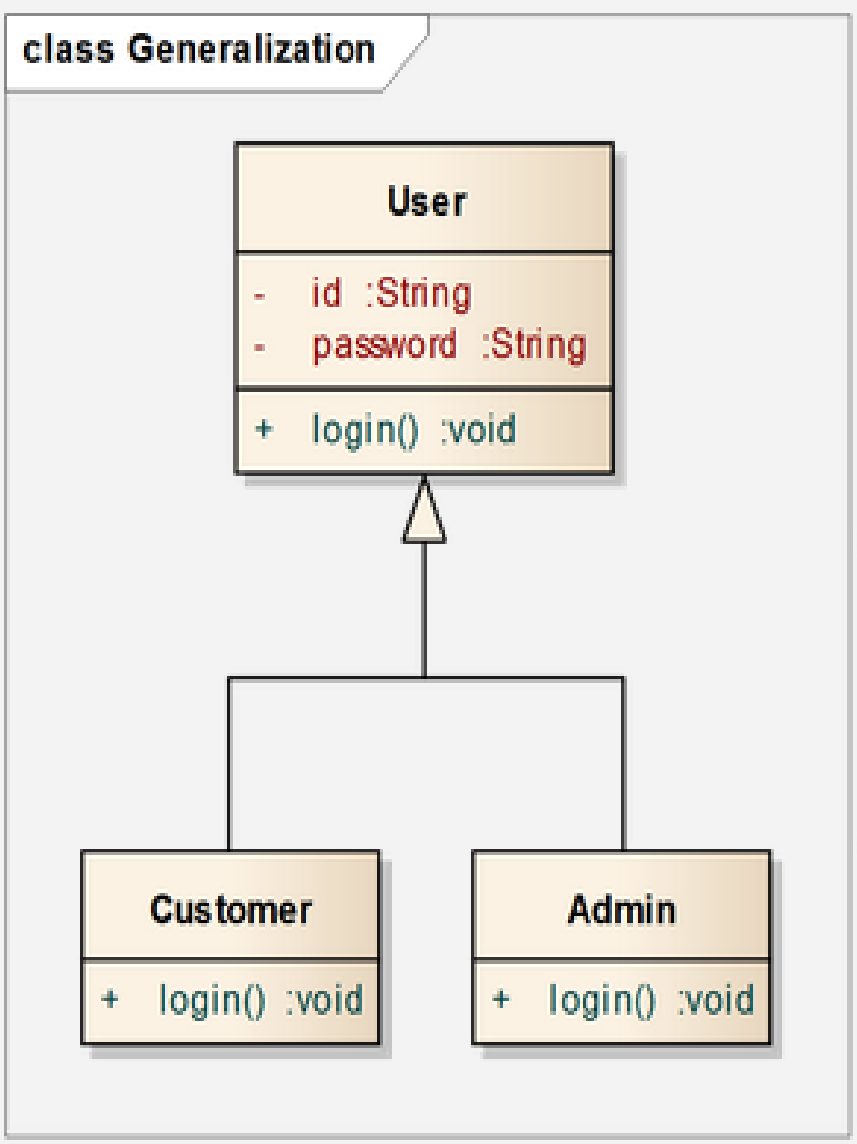
```
3 public class Math {
4
5     public static final double PI = 3.14159;
6
7     public static double sin(double theta) {
8         // Sine 계산...
9         return 0;
10    }
11    public static double cos(double theta) {
12        // Cosine 계산...
13        return 0;
14    }
15 }
```

4.2.2 일반화 표기법의 이용

- 일반화(Generalization)
  - 상속관계를 표현하는 표기법
  - 기계적으로 코딩 가능




```
class Person {
    ...
};
class Man : public Person {
    ...
};
class Woman : public Person {
    ...
};
```



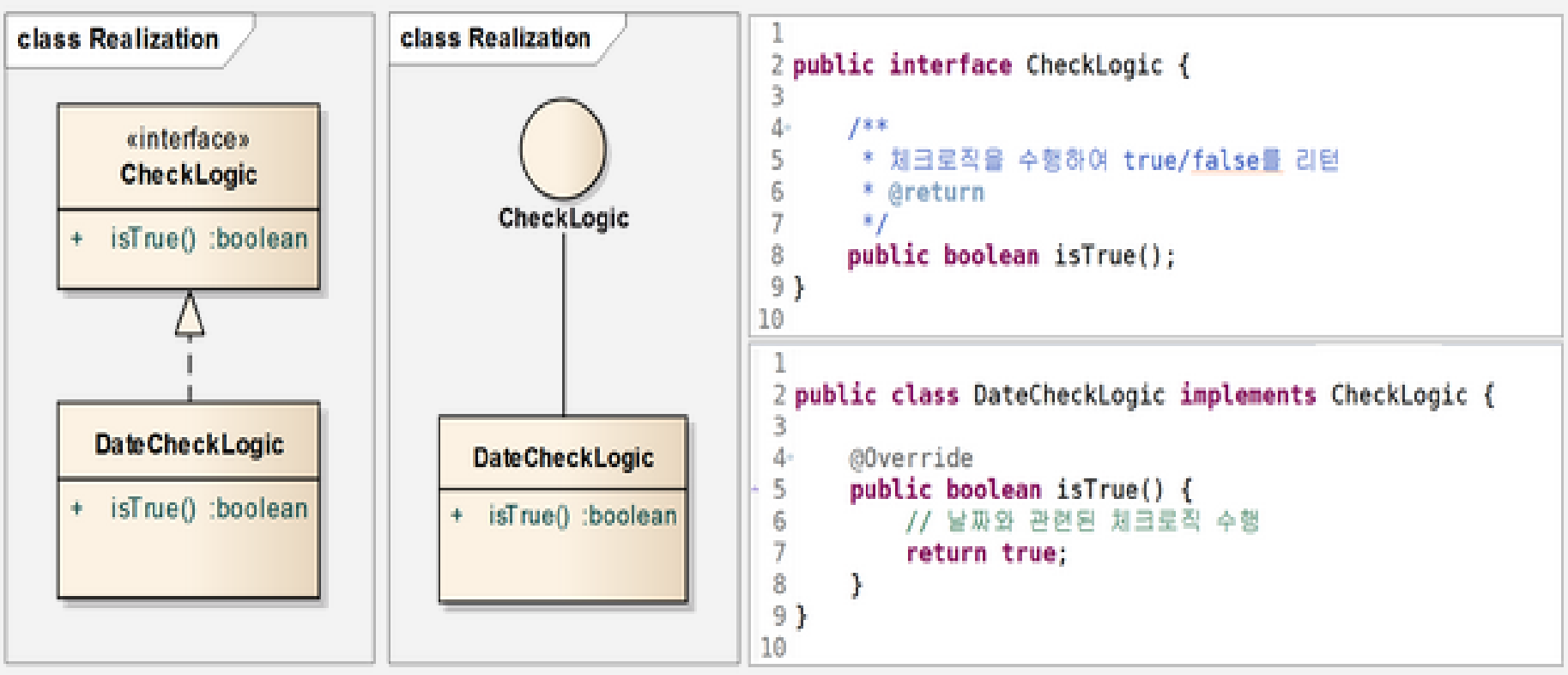

```java
public class User {

    private String id;

    private String password;

    /**
     * 로그인
     */
    public void login() {
        // 일반 사용자의 로그인
    }
}
```

```java
public class Customer extends User {

    @Override
    public void login() {
        // Customer의 로그인..
    }
}
```

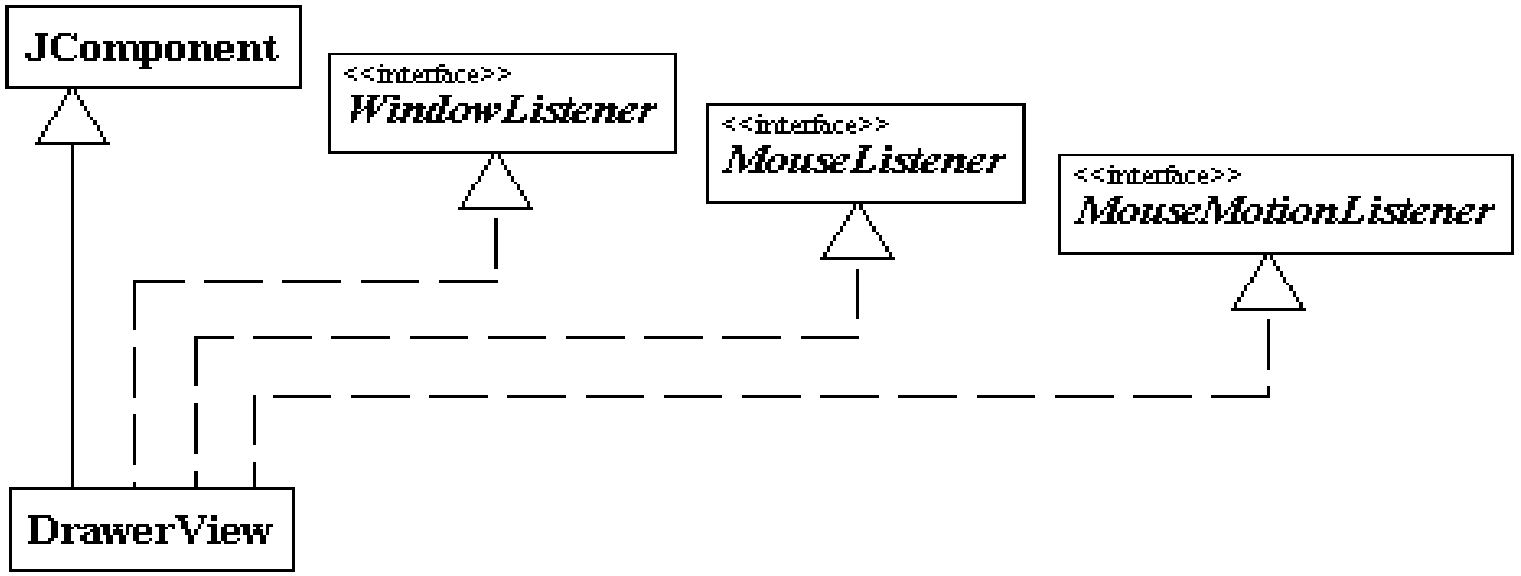
```java
public class Admin extends User {

    @Override
    public void login() {
        // 관리자의 로그인...
    }
}
```

< 일반화 (Generalization) >

- 정제화(refinement)
  - 일반화의 변형
  - 인터페이스를 구현하는 관계를 표현
  - 기계적으로 코딩 가능



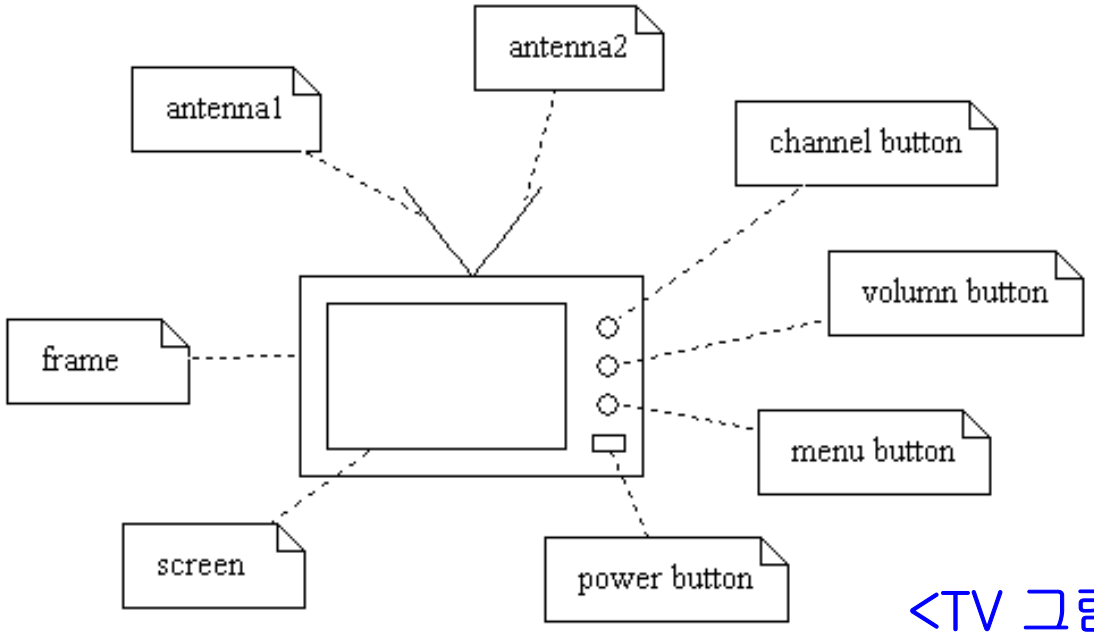
< 실체화 (Realization) = 정제화 (Refinement) >




```
class DrawerView extends Jcomponent
    implements WindowListener, MouseListener, MouseMotionListener {
    ...
}
```

4.2.3 집합화/합성화

- 집합화(aggregation)/합성화(composition)는 객체의 부속품 관계를 모델링
  - 기계적으로 코딩 되지는 않음 : 취향에 따라 다른 코딩 스타일 가능
  - 간단한 변환 절차에 의해 코드에 반영 가능
- 사물을 구성하는 부속품의 존재 인식



<TV 그림의 부속 객체>

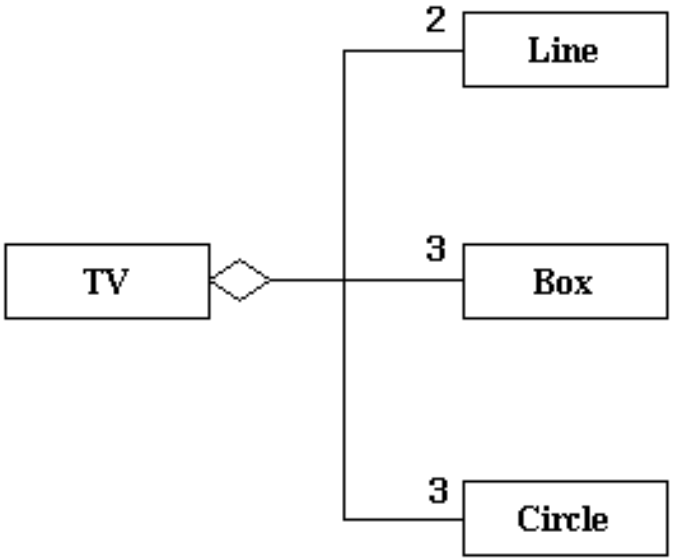
- 부속품 관계를 집합화와 합성화로 구분
  - 주 객체와 부속 객체간의 생명 시간이 얼마나 밀접하게 연관되어 있는가?

  - 부속품이 없을 때, 주 객체의 서비스가 정상적으로 제공될 수 없다면 → 합성화로 모델링
    - 주 객체와 부속 객체가 생성도 동시에, 소멸도 동시에
    - 예) “사람”과 “머리”, “몸통” 등의 관계

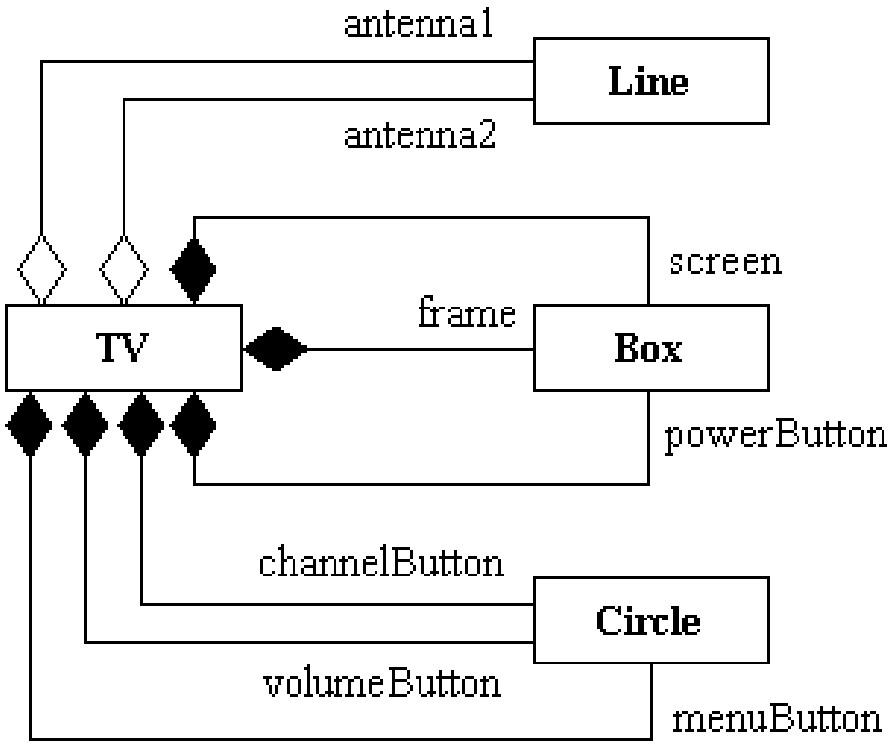
  - 필수적인 부속품이 아닌 경우 → 집합화로 모델링
    - 주 객체와 부속 객체의 생명 시간이 일치하지 않음
    - 주 객체 소멸 시 부속 객체도 소멸
    - 예) “사람”과 “팔”, “다리” 등의 관계

1. 모델링 시 우선 집합화를 위주로 작성
2. 집합화 중 일부를 합성화로 설정
   - 생명 시간 고려와 필수 기능 제공 여부를 통해

- 예) 개략 설계 단계에서 작성된 TV 객체 모델
  - "TV" 객체는 2개의 "선"과 3개의 사각형, 3개의 "원"으로 이루어짐 → 일단 집합화(마름모)로 표현

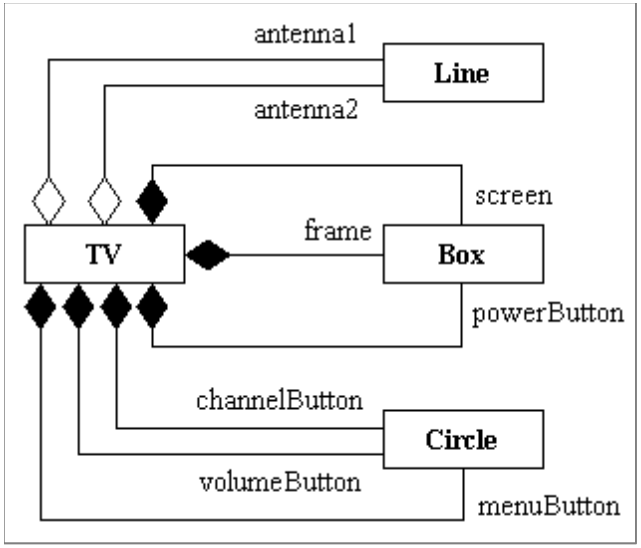


- 예) TV 객체에 대한 상세 설계 모델
  - 부속 객체의 역할 표현
  - 합성화(색칠한 마름모로 표기) 반영



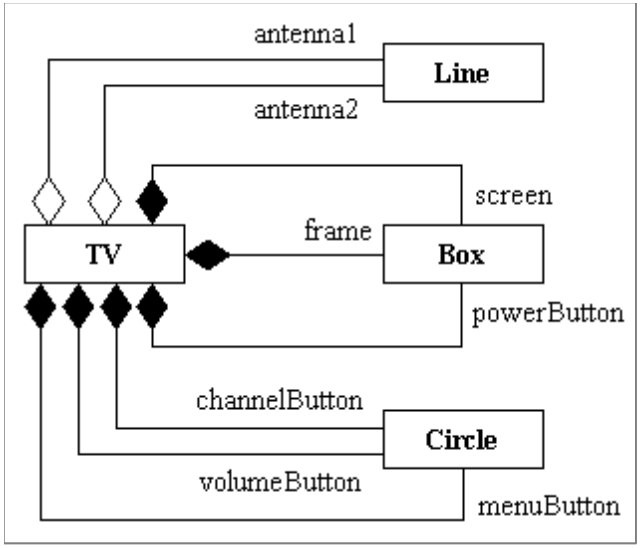
[방식 1] 집합화/합성화 모델의 코드 변환 절차

1. 집합화/합성화에 명시된 역할 이름을 포인터 타입의 데이터 멤버로 명시
2. 생성자에서 합성화에 해당하는 데이터 멤버는 무조건 동적으로 할당하고, 집합화에 해당하는 데이터 멤버는 조건에 따라 동적 할당
3. 소멸자에서 합성화에 해당하는 데이터 맴버는 무조건 소멸하고, 집합화에 해당하는 데이터 멤버는 값이 NULL이 아닌 경우에 소멸
4. 주 객체의 서비스를 구현하는 멤버 함수의 구현 시에 서비스 위임(delegation)을 이용한 코드 작성
   - 주 객체가 수행해야 하는 서비스를 부속 객체에 전가



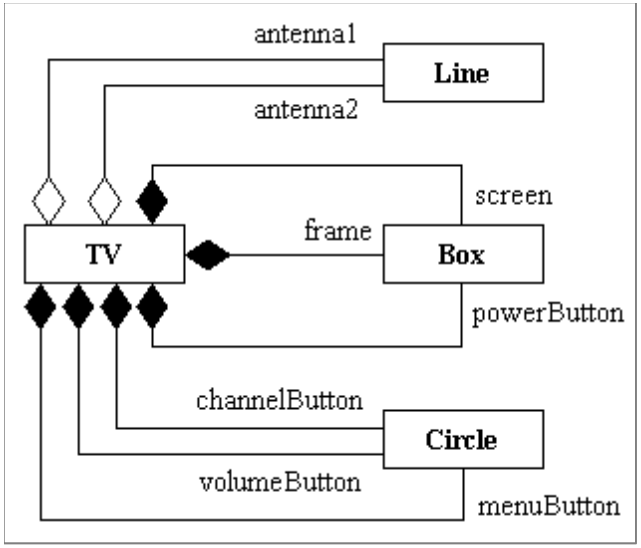
1. 집합화/합성화에 명시된 역할 이름을 포인터 타입의 데이터 멤버로 명시

```
class TV : public SuperClass {
    …
    BOOL _antennaFlag;    // 안테나 사용 여부
    Box* _frame;          // 합성화 멤버 부속 객체
    Box* _screen;         // 합성화 멤버 부속 객체
    Circle* _channelButton;  // 합성화 멤버 부속 객체
    Circle* _volumnButton;   // 합성화 멤버 부속 객체
    Circle* _menuButton;     // 합성화 멤버 부속 객체
    Box* _powerButton;       // 합성화 멤버 부속 객체
    Line* _antenna1;      // 집합화 멤버 부속 객체
    Line* _antenna2;      // 집합화 멤버 부속 객체

// Operations
public:
    TV(…, BOOL antennaOption);
    virtual ~TV();
    ...
};
```



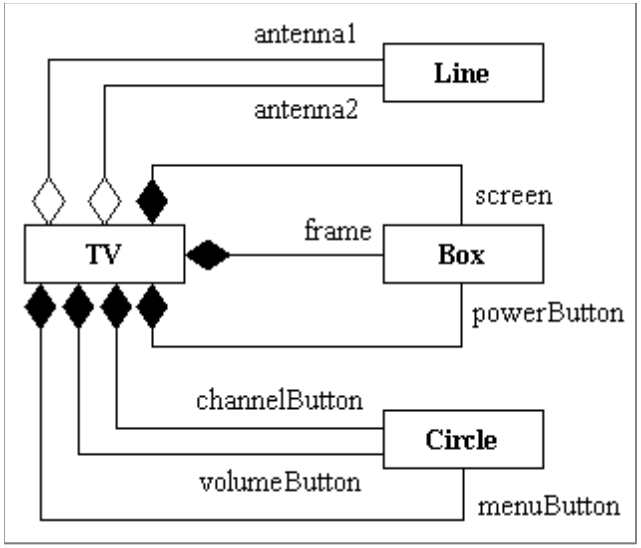
2. 생성자에서 합성화에 해당하는 데이터 멤버는 무조건 동적으로 할당하고, 집합화에 해당하는 데이터 멤버는 조건에 따라 동적 할당

```
// 생성자
TV::TV(…,BOOL antennaOption) : SuperClass(…) {
    ...
    _frame = new Box(…);       // 합성화 멤버는 무조건 생성
    _screen = new Box(…);      // 합성화 멤버는 무조건 생성
    _channelButton = new Circle(…);  // 합성화 멤버는 무조건 생성
    ...
    if (antennaOption == TRUE) {
        _antenna1 = new Line(…);  // 집합화 멤버는 조건에 따라 생성
        _antenna2 = new Line(…);  // 집합화 멤버는 조건에 따라 생성
    } else {
        _antenna1 = NULL;
        _antenna2 = NULL;
    }
    ...
}
```



3. 소멸자에서 합성화에 해당하는 데이터 맴버는 무조건 소멸하고, 집합화에 해당하는 데이터 멤버는 값이 NULL이 아닌 경우에 소멸

```
// 소멸자
TV::~TV() {
    delete _frame;              // 합성화 멤버는 무조건 소멸
    delete _screen;             // 합성화 멤버는 무조건 소멸
    delete _channelButton;      // 합성화 멤버는 무조건 소멸
    ...
    if (antenna1 != NULL) {
        delete _antenna1;       // 집합화 멤버는 조건에 따라 소멸
    }
    if (antenna2 != NULL) {
        delete _antenna2;       // 합성화 멤버는 무조건 소멸
    }
    ...
}
```



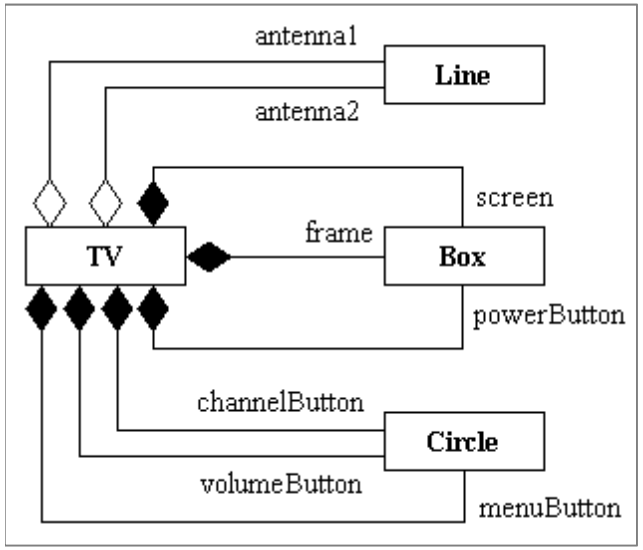
4. 주 객체의 서비스를 구현하는 멤버 함수의 구현 시에 서비스의 위임(delegation)을 이용한 코드 작성

예) "TV" 객체를 움직이는 함수의 구현

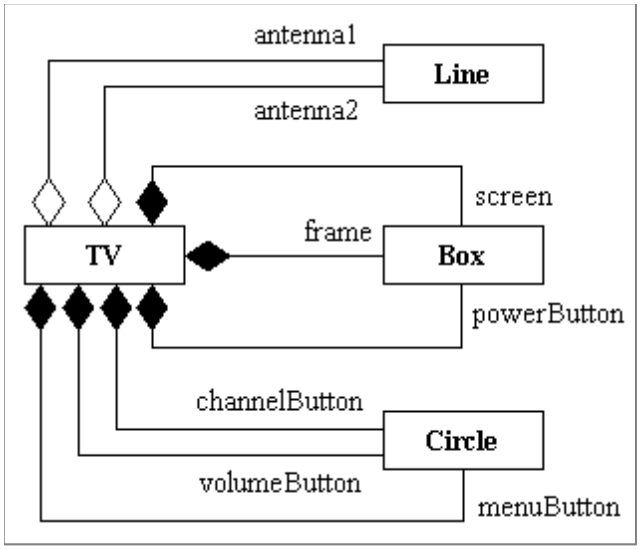
```
void TV::move(int dx,int dy) {
    SuperClass::move(dx,dy); // TV 클래스와 관계되는 이동 작업을 한 후,
    _frame->move(dx,dy);     // 프레임을 이동
    _screen->move(dx,dy);    // 스크린을 이동
    _channelButton->move(dx,dy); // 버튼들을 이동
    ...
    if (antenna1 != NULL) {  // 안테나가 있는 경우에만 이동
        _antenna1->move(dx,dy);
    }
    if (antenna2 != NULL) {  // 안테나가 있는 경우에만 이동
        _antenna2->move(dx,dy);
    }
    ...
}
```

[방식 2] 집합화/합성화 모델의 다른 구현 방식

- 데이터 멤버들을 자동 변수로 정의
  - 일단 디폴트 생성자 호출하여 객체 생성
  - 추후 Create*****() 함수 호출하여 정상적인 객체 생성




```
class TV : public SuperClass {
    ...
    BOOL _antennaFlag;    // 안테나 사용 여부
    Box _frame;           // 합성화 멤버 부속 객체
    Box _screen;          // 합성화 멤버 부속 객체
    Circle _channelButton;  // 합성화 멤버 부속 객체
    Circle _volumnButton;   // 합성화 멤버 부속 객체
    Circle _menuButton;     // 합성화 멤버 부속 객체
    Box _powerButton;     // 합성화 멤버 부속 객체
    Line _antenna1;       // 집합화 멤버 부속 객체
    Line _antenna2;       // 집합화 멤버 부속 객체
// Operations
public:
    TV(..., BOOL antennaOption);
    virtual ~TV();
    ...
};
```



2. 생성자

- 소멸자를 위한 코딩이 없어지거나 매우 간단해짐
- 동적 바인딩 사용 못함
- 유연성 떨어지고 자연스럽지 못한 코딩

```
// 생성자
TV::TV(…,BOOL antennaOption) : SuperClass(…) {
    _frame.Create(…);                  // 합성화 멤버는 무조건 생성
    _screen.Create(…);                 // 합성화 멤버는 무조건 생성
    _channelButton.Create(…);          // 합성화 멤버는 무조건 생성
    ...
    _antennaFlag = antennaOption;
    if (_antennaFlag == TRUE) {
        _antenna1.Create(…);           // 집합화 멤버는 조건에 따라 생성
        _antenna2.Create(…);           // 집합화 멤버는 조건에 따라 생성
    }
    ...
}
```

```
// 소멸자
TV::~TV() {
    _frame.Destroy();
    _screen.Destroy();
    _channelButton.Destroy();
    ...
    if (_antennaFlag) {
        _antenna1.Destroy();
        _antenna2.Destroy();
    }
    ...
}
```

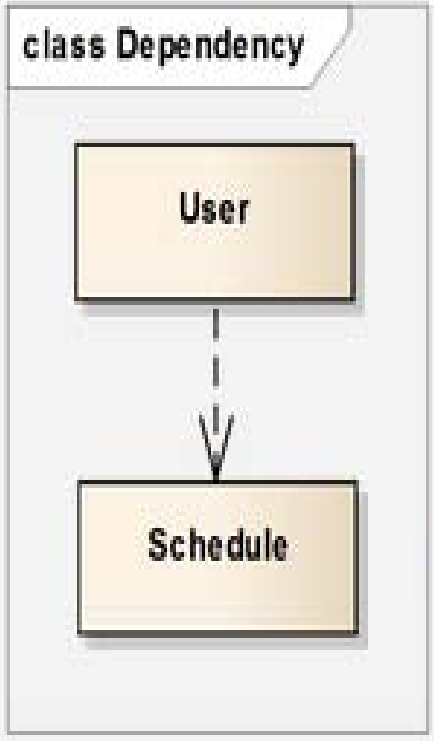
3. 소멸자

4. 서비스 위임을 이용한 멤버 함수 구현

```
void TV::move(int dx,int dy) {
    SuperClass::move(dx,dy); // TV 클래스와 관계되는 이동 작업을 한 후,
    _frame.move(dx,dy);      // 프레임을 이동
    _screen.move(dx,dy);     // 스크린을 이동
    _channelButton.move(dx,dy);  // 버튼들을 이동
    …
    if (_antennaFlag) {      // 안테나가 있는 경우에만 이동
        _antenna1.move(dx,dy);
        _antenna2.move(dx,dy);
    }
    …
}
```

4.2.4 의존관계

- 함수 호출 관계와 비슷함
- 서비스 사용자와 서비스 공급자의 관계
  - 생명 시간 관계가 서로 독립적
    - 집합화와 다른 점
    - 공급자 객체가 소멸될 때, 소비자 객체가 소멸될 필요 없음
- 기계적으로 코드로 변환되지 않음



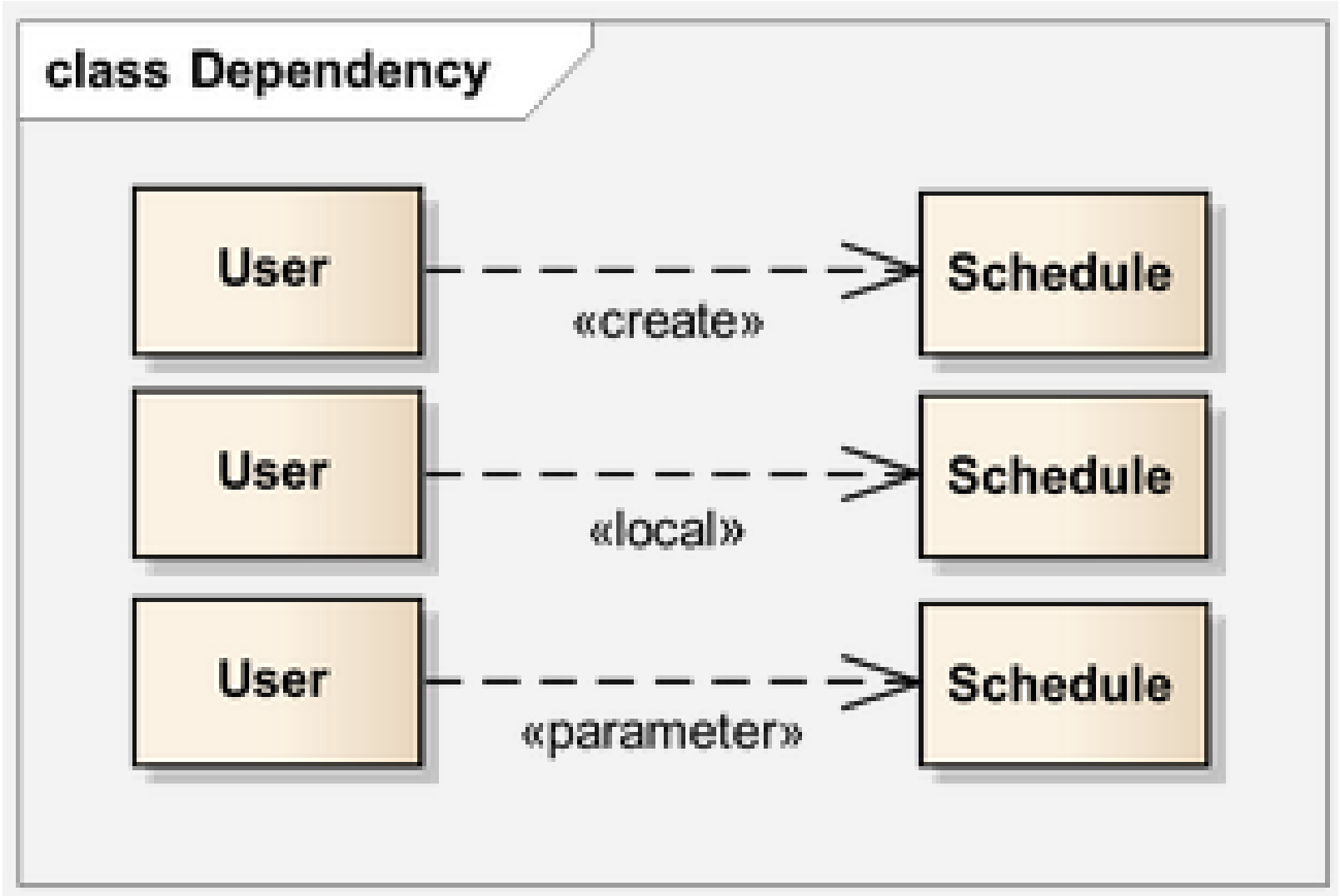

```
2
3 public class User {
4
5     public Schedule crateSchedule() {
6         // 객체 생성 및 리턴
7         return new Schedule();
8     }
9
10    public void useSchedule(Schedule schedule) {
11        // 객체를 매개변수로 받아 사용
12        // use schedule...
13        Schedule schedule2014 = schedule.getScheduleByYear(2014);
14    }
15 }
16
```

❖ 의존관계가 프로그램에 반영되는 스타일

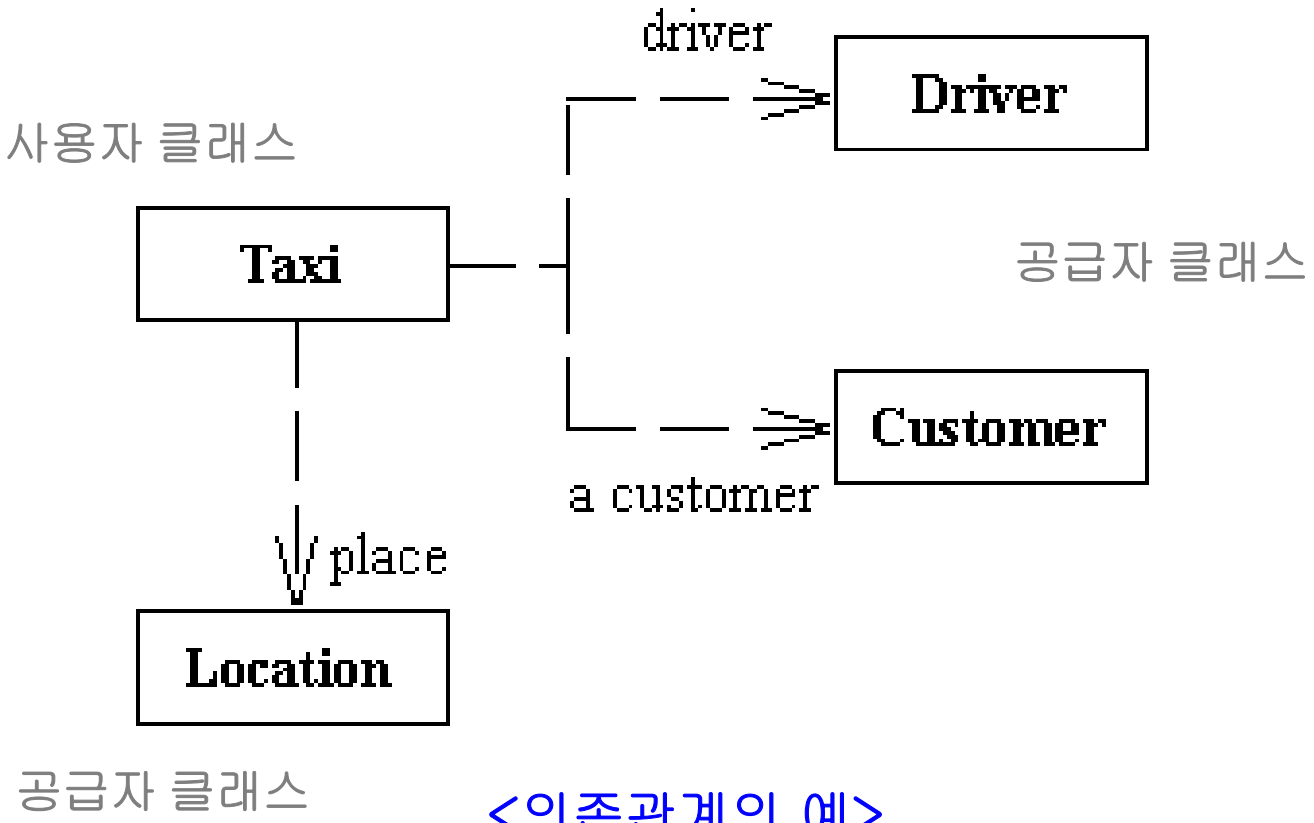
1. 사용자 클래스의 데이터 멤버에 공급자 객체가 포인터로 정의되는 경우
   - 사용자 객체와 공급자 객체가 빈번하게 서비스를 주고 받는 경우

2. 사용자 멤버 함수의 시그니처에 형식 인자로 공급자 객체가 명시되는 경우
   - 공급자 객체의 서비스가 사용자 클래스의 한 멤버 함수가 실행되는 동안에만 필요한 경우

3. 사용자 멤버 함수의 바디에서 지역 변수로 공급자 객체가 인스턴시에이션 되는 경우
   - 공급자 객체가 사용자 멤버 함수의 일부분에서만 필요한 경우



< Dependency Stereo Type >



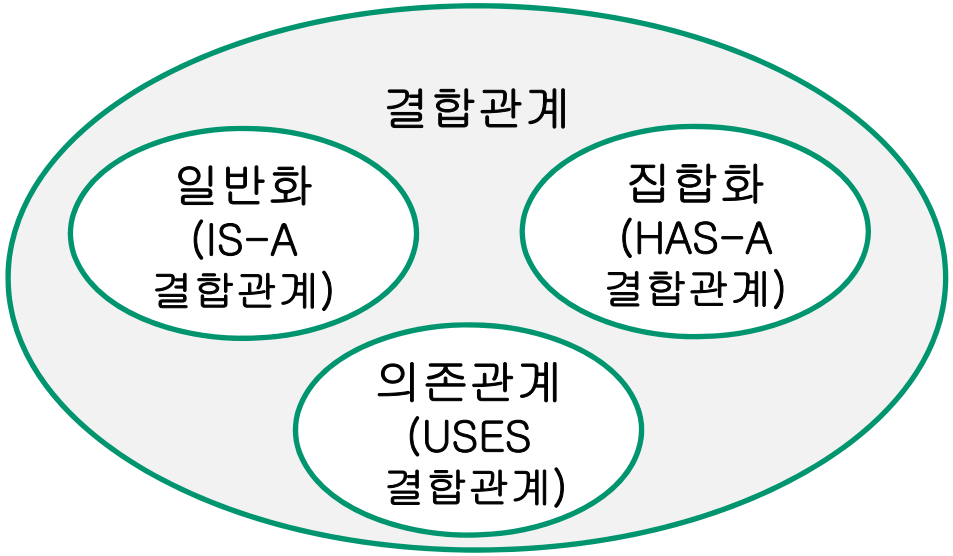
<의존관계의 예>

- 택시 회사에서 택시 운행상황, 기사 근무 상황, 고객 지불방식 등을 처리하기 위한 클래스 다이어그램의 일부분
- 의존관계는 매우 많기 때문에 모든 의존관계를 표현할 필요는 없음
- 특이하거나 중요한 의존관계만 명시

```
class Taxi {
    int _totalIncome;
    Driver* _driver;      // (1) 포인터로 정의된 경우
public:
    void getOn(Driver* employee) {
        _driver = employee;
        _driver->checkIn(GetTime());  // 출근 시간 등록
    }
    Driver* getOff() {
        _driver->checkOut(GetTime()); // 퇴근 시간 등록
        Driver *tmp = _driver;
        _driver = NULL;
        return tmp;
    }
    void ride(Customer* pCustomer) { // (2) 멤버 함수의 형식 인자로 정의
        Location* place = pCustomer->whereToGo(); // (3) 지역 변수로 정의

        // 운행하면서 Taxi, Driver, Location 객체의 상태를 변화 시킴
        int fare = place->calculateFare();
        _totalIncome = _totalIncome + pCustomer->pay(fare);
    }
};
```

4.2.5 일대일(1:1) 결합관계

- 객체와 객체 사이에 존재하는 임의의 관계를 모델링한 것
  - 추상화 수준이 높음
  - 일반화, 집합화/합성화, 의존관계는 결합관계 중에서 특화된 것
    - 즉, 결합관계의 부분 집합
  - 기타 관계 : 다중성, 결합 클래스, 삼각 관계 등
- 다소 복잡하게 코드에 반영됨

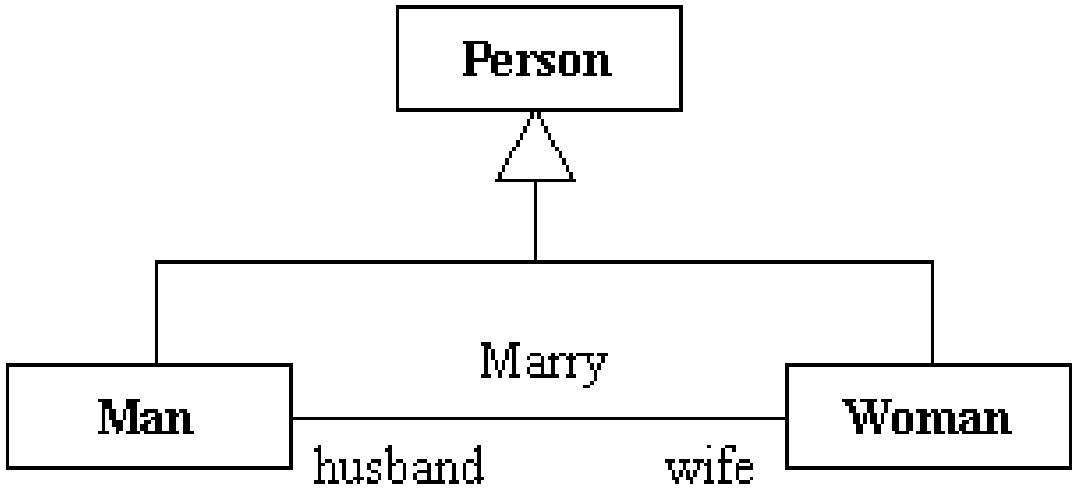


❖ 다중성(multiplicity)

- 일대일(1:1) 결합관계
- 일대상수(1:K) 결합관계
- 일대다(1:N) 결합관계
- 다대다(M:N) 결합관계

- 어떠한 다중성을 갖는가는 분석가나 설계자가 애플리케이션에 대한 정확한 이해를 바탕으로 결정해야 함

- 결합관계의 명칭은 주로 동사
  - 요구사항 명세서에서 유추



<일대일 결합관계의 예>

- 어떠한 남자와 어떠한 여자가 결혼하는 사실을 모델링한 것
- 다중성이 일대일로 설정됨 → 일부일처제이기 때문

❖ 1:1 결합관계를 코드로 반영하는 3가지 방식

1. 단지 결합관계의 명칭을 하나의 함수로 구현하는 경우
   - 두 객체가 평상시에는 상호 작용을 하지 않다가 관계를 맺는 순간에만 어떠한 임무를 수행하는 경우
2. 한쪽 객체에서 상대편 객체로의 링크를 설정하는 의존관계로 변형하는 경우
   - 두 객체 사이에 관계가 설정된 후, 수시로 한 쪽 객체에서 상대편 객체의 서비스를 이용하는 경우
3. 두 객체가 서로 상대편의 링크를 갖는 의존관계로 변형하는 경우
   - 양쪽 객체 모두에서 상대방에 대한 정보를 접근하고자 할 때

1. 단지 결합관계의 명칭을 하나의 함수로 구현하는 경우

- 가족을 구성한 이후에 남자와 여자 객체가 서로 직접적으로 서비스를 주고받지는 않기 때문에 하나의 함수로 구현

```
// In the case of no link is needed since they are bounded.
Family* Man::marry(Woman* wife) {
   // 새로운 가정을 구성한다.
   return new Family(this, wife);
}
```

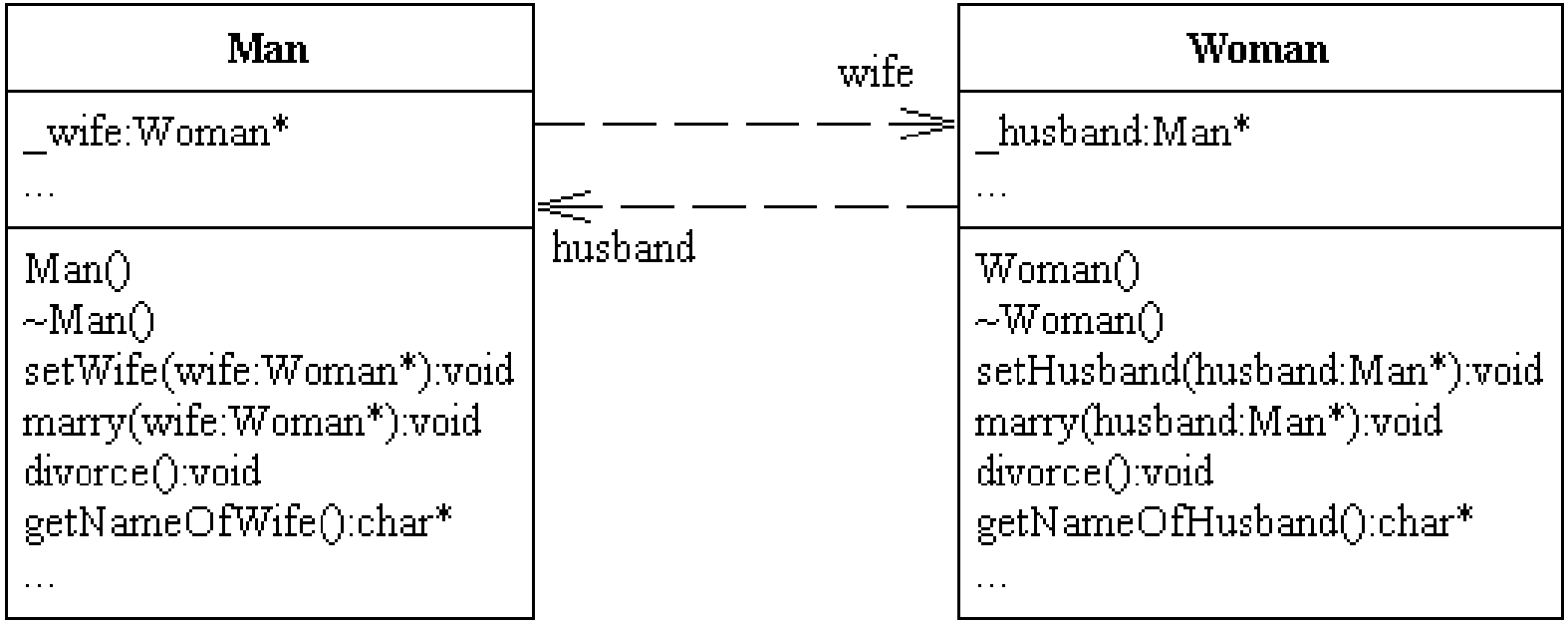
2. 한쪽 객체에서 상대편 객체로의 링크를 설정하는 의존관계로 변형하는 경우

- 포인터 타입의 데이터 멤버를 클래스에 정의
- 관계 설정, 관계 해지, 관계 사용 함수 정의
- 아래와 같은 코딩은 어떤 여자의 남편 이름은 알 수 없다.

```
class Man : public Person {
    ...
    Woman* _wife;          // 부인 쪽으로의 링크 설정을 위한 포인터

Public:
    Man(...) : Person (...) {
        ...
        _wife = NULL;      // 처음에는 이 남자의 부인이 없다.(총각)
    }
    void marry(Woman* wife) { // 관계가 설정되는 순간에 호출되는 함수
        _wife = wife;
    }
    void divorce() {       // 관계가 해지되는 시점에 호출되는 함수
        _wife = NULL;
    }
    char* getNameOfWife() {  // 링크의 사용 예를 보이는 함수
        if (_wife == NULL) return NULL;
        return _wife->getName();
    }
    ...
}
```

3. 두 객체가 서로 상대편의 링크를 갖는 의존관계로 변형하는 경우

- 결합관계의 해제 상황을 정확히 코드에 반영해야 함
  - 관계 설정과 해지 시 양쪽 포인터를 모두 변경해야 함
- 소멸자의 역할 매우 중요



<일대일 결합관계를 의존관계로 변경한 예>

```
class Man : public Person {
    ...
    Woman* _wife;          // 부인 쪽으로의 링크 설정을 위한 포인터
Public:
    Man(...) : Person (...) {
        ...
        _wife = NULL;      // 처음에는 이 남자의 부인이 없다.(총각)
    }
    ~Man() { // 소멸자: 남자가 죽는 경우에 부인으로부터 남편 쪽으로 설정된 링크 해제
        if (_wife != NULL) {
            _wife->setHusband(NULL);
            _wife = NULL;
        }
    }
    void setWife(Woman* wife) {
        _wife = wife;
    }
    void marry(Woman* wife) {     // 관계가 설정되는 순간에 호출되는 함수
        _wife = wife;             // 남자로부터 부인 쪽으로 링크 설정
        _wife->setHusband(this);  // 여자로부터 남편 쪽으로 링크 설정
    }
    void divorce() {              // 관계가 해지되는 시점에 호출되는 함수
        _wife->setHusband(NULL);  // 여자로부터 남편 쪽으로의 링크 해제
        _wife = NULL;             // 남자로부터 부인 쪽으로 링크 해제
    }
    char* getNameOfWife() {       // 링크의 사용 예를 보이는 함수
        if (_wife == NULL) return NULL;
        return _wife->getName();
    }
}
```

Woman 클래스도 유사하게 작성

4.2.6 일대상수(1:K) 결합관계

- 컨테이너 역할을 하는 객체의 수용 능력이 고정되는 경우
  - 예) 버스의 좌석 수, 호텔 객실 수 등
- 일대상수 결합관계를 코드로 변환 시
  - 컨테이너 역할을 하는 객체의 데이터 멤버에 포인터 배열 정의
    - 호텔 클래스의 데이터 멤버로 손님 객체를 가리키는 포인터 배열 정의

[참고] 컨테이너(container)

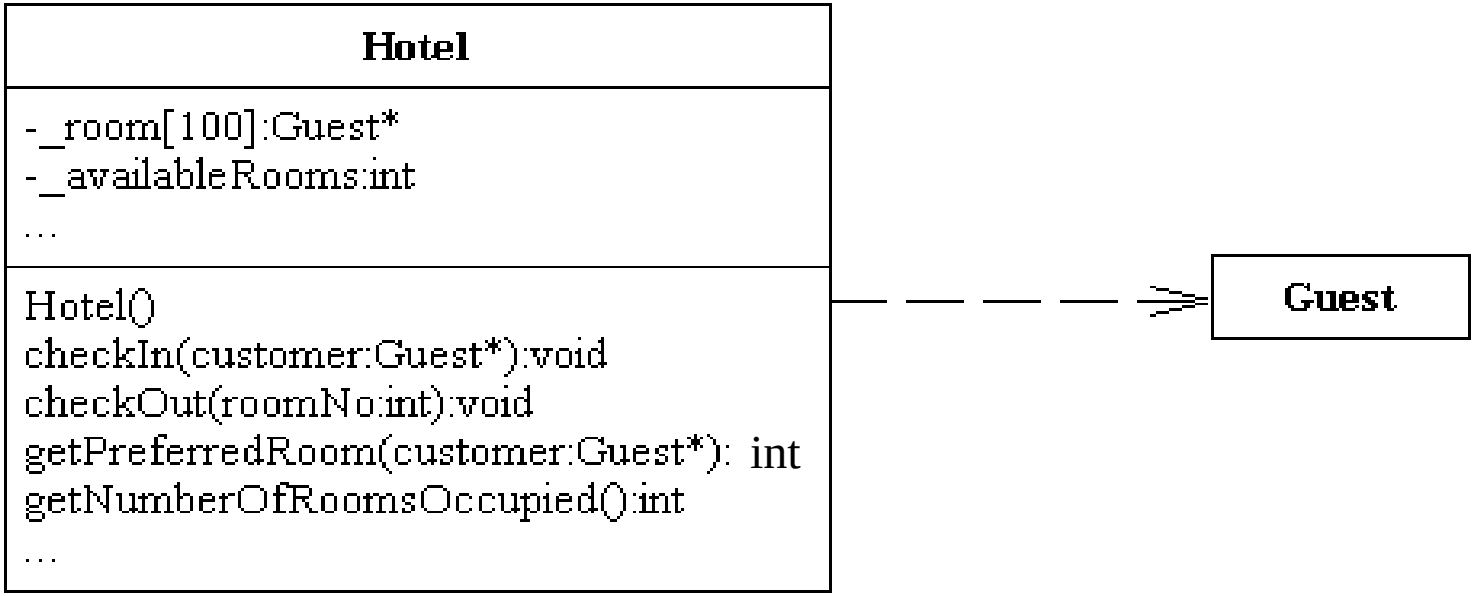
- 여러 객체에 대한 포인터나 레퍼런스를 저장하기 위해 사용되는 배열 또는 리스트와 같은 객체

- 예) 100개의 객실(1인 1실)을 갖는 호텔

<일대상수 결합관계의 예>

- 일대상수 결합관계에서도 일대일 결합관계와 같이 한 쪽 방향으로의 링크를 유지할 것인지, 양방향으로의 링크를 유지할 것인지 결정해야 함
  - 응용 목적에 따라 결정
  - 보통은 단방향 링크만 있어도 됨

- checkIn()
  - 호텔과 손님 객체가 관계를 맺음
- checkOut()
  - 호텔과 손님 객체가 관계를 끊음
- getPreferredRoom(), getNumberOfRoomsOccupied()
  - 포인터 배열을 사용하는 함수



<포인터 배열을 이용하여 일대상수 결합관계를 의존관계로 변경한 예>

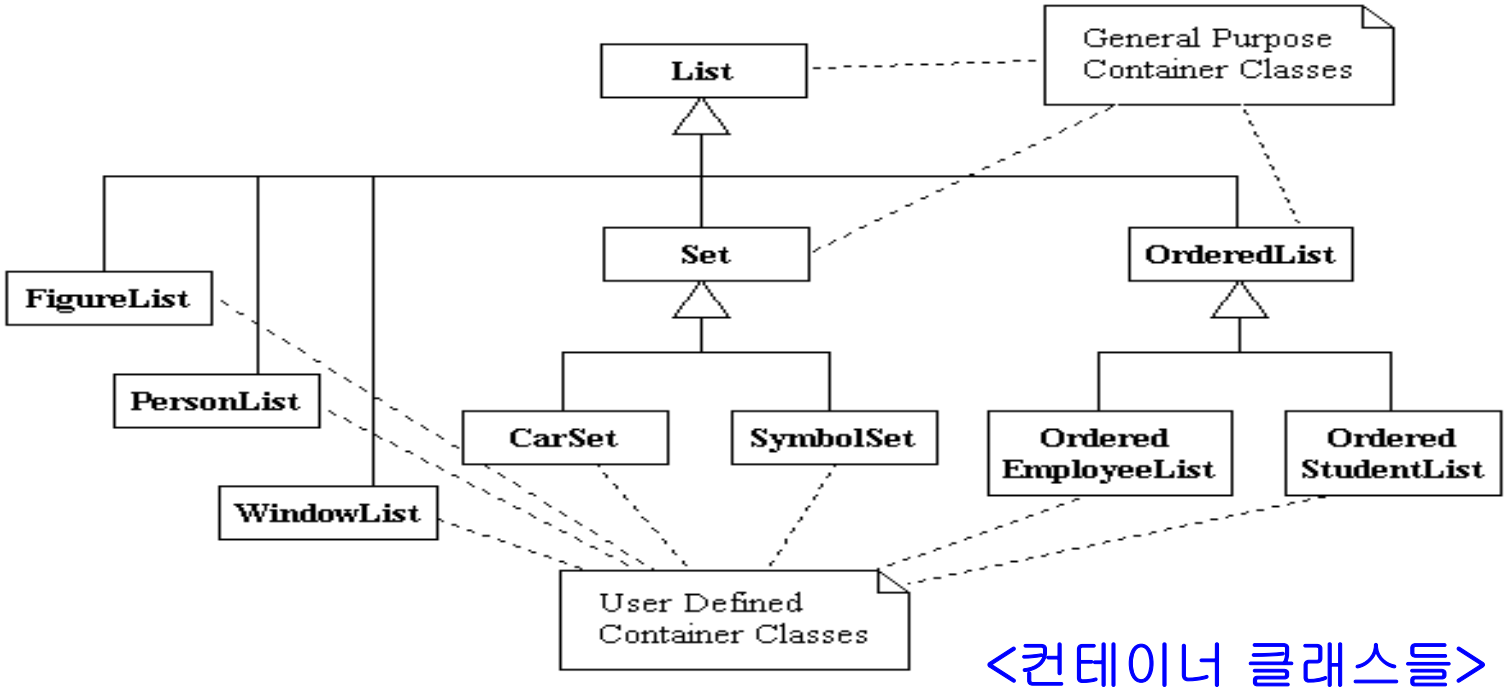
```
class Hotel {
private:
    Guest* _room[100];  // 포인터 배열
    int _availableRooms;
    ...
public:
    Hotel() {
        // 처음에는 100개의 객실이 비어 있음
        for (int i = 0; i < 100; i++)  _room[i] = NULL;
        _availableRooms = 100;
    }
    void checkIn(Guest* customer) { // 관계 설정 시 호출
        int roomNo = getPrefferedRoom(customer);
        _room[roomNo] = customer;
        _availableRooms = _availableRooms - 1;
    }
    void checkOut(int roomNo) {     // 관계 해지 시 호출
        _room[roomNo] = NULL;
        _availableRooms = _availableRooms + 1;
    }
    int getPreferredRoom(Guest* customer) {
        // 완전한 코드는 아님
        Preference* pref = customer->getPreference();
        int roomNo = searchRoomFor(pref);
        return roomNo;
    }
    int getNumberofRoomsOccupied() {
        int count = 0;
        for (int i = 0; i < 100; i++)
            if (_room[i] != NULL)  count++;
        return count;
    }
};
```

4.2.7 일대다(1:N) 결합관계

- 일대상수 결합관계와 다른 점은 다수의 값이 고정되지 않는다는 점
- 컨테이너의 역할을 수행하는 객체의 수용능력이 미리 결정되어 있지 않은 경우
- 일대다 결합관계를 코드로 변환 시
  - 컨테이너 역할을 하는 객체의 데이터 멤버에 연결 리스트(linked list), 폴리몰픽 리스트를 사용

- 폴리몰픽 컨테이너 or 폴리몰픽 리스트
  - 다형 개념의 적용을 위해 사용

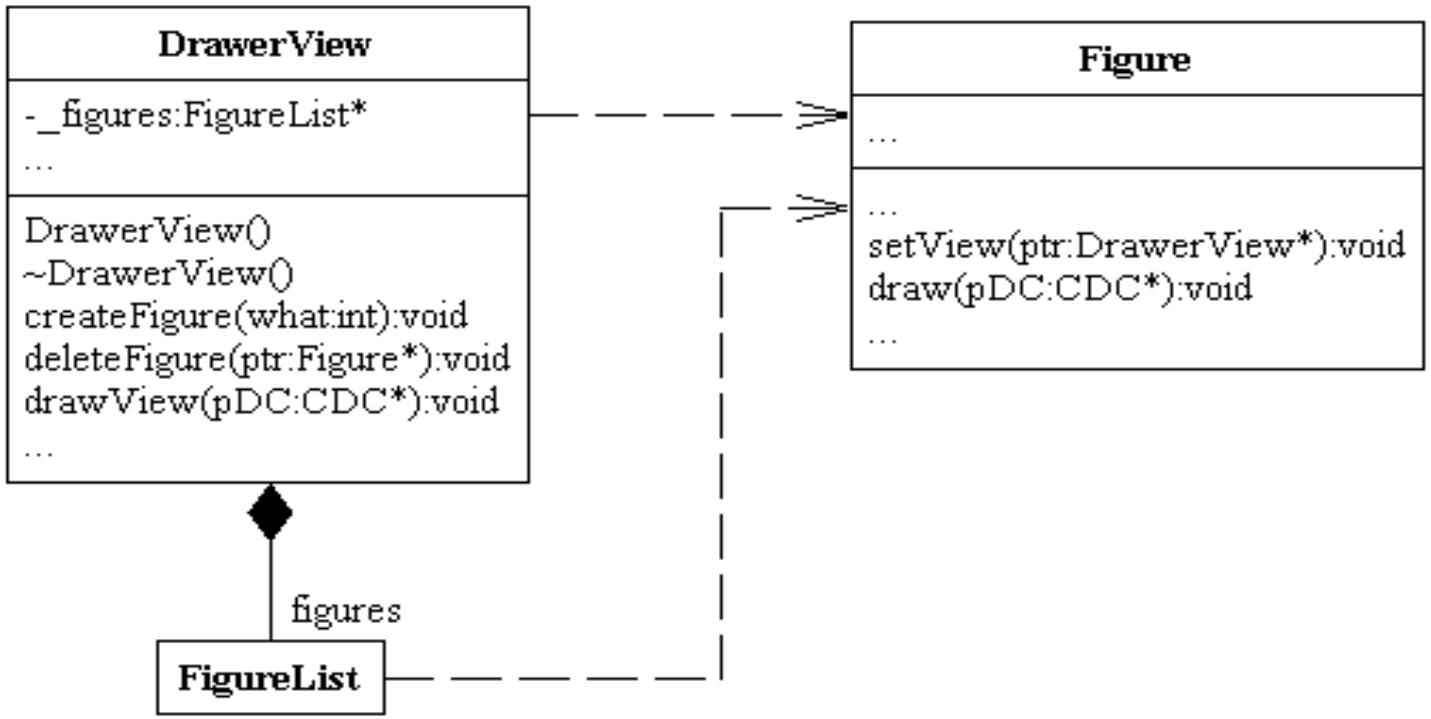
- 범용적인 컨테이너 클래스 또는 리스트
  - MFC의 경우 : CList, CObList 클래스 등
  - Java의 경우 : LinkedList, Vector 클래스 등
- 범용적인 클래스를 직접 사용하기 보다 사용자 정의 폴리몰픽 리스트를 사용하는 것이 좋음
  - 라이브러리 컨테이너의 기능을 그대로 이용하되, 타입 변환 작업만 수행



<컨테이너 클래스들>

- 일대다 결합관계의 예
  - 어떠한 회사에 근무하는 인원들
  - 어느 가족에 속하는 자녀들
  - 특정 은행에 가입한 고객들
  - 경부 고속도로에 운행 중인 자동차들
  - 그래픽 편집기에 그려진 그래픽 객체를 모델링한 경우
    - 코드 변환 시에는 많은 그림 객체를 담아 관리할 수 있는 컨테이너인 FigureList가 필요

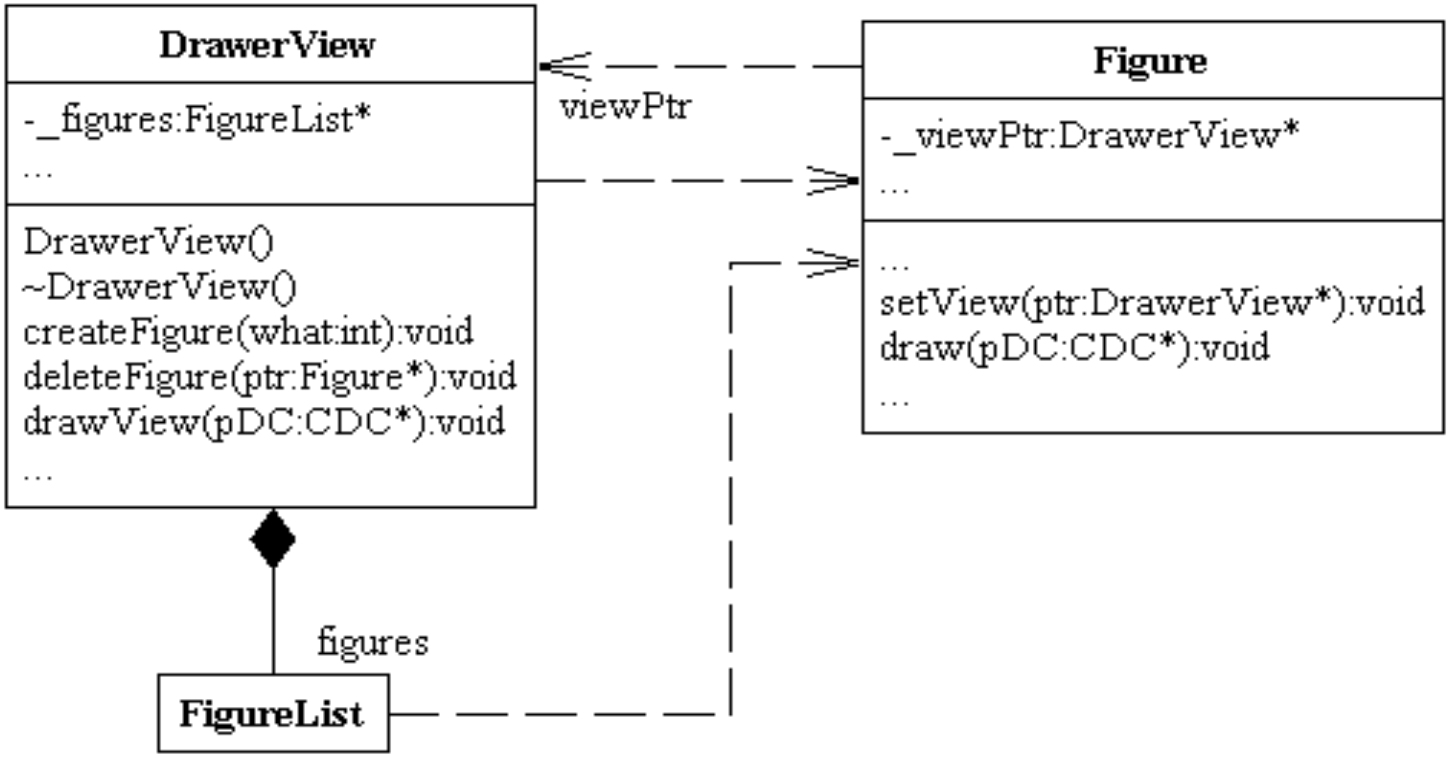
<일대다 결합관계의 예>



<컨테이너를 이용하여 일대다 결합관계를 의존관계로 변경한 예(단방향)>

- 컨테이너 객체인 _figures는 합성화를 이용해 화면 객체의 부속 객체로 모델링됨
  - 컨테이너 객체와 화면 객체가 동일한 시점에 생성, 소멸됨
- 보통의 경우, 컨테이너에서 상대편으로 한쪽 방향만 링크 유지하면 됨
  - 필요에 따라 양방향 설정도 가능

- 앞 예를 양방향 링크로 설정한 경우



<일대다 결합관계에 대하여 양방향 링크를 이용하여 의존관계로 변경한 예>

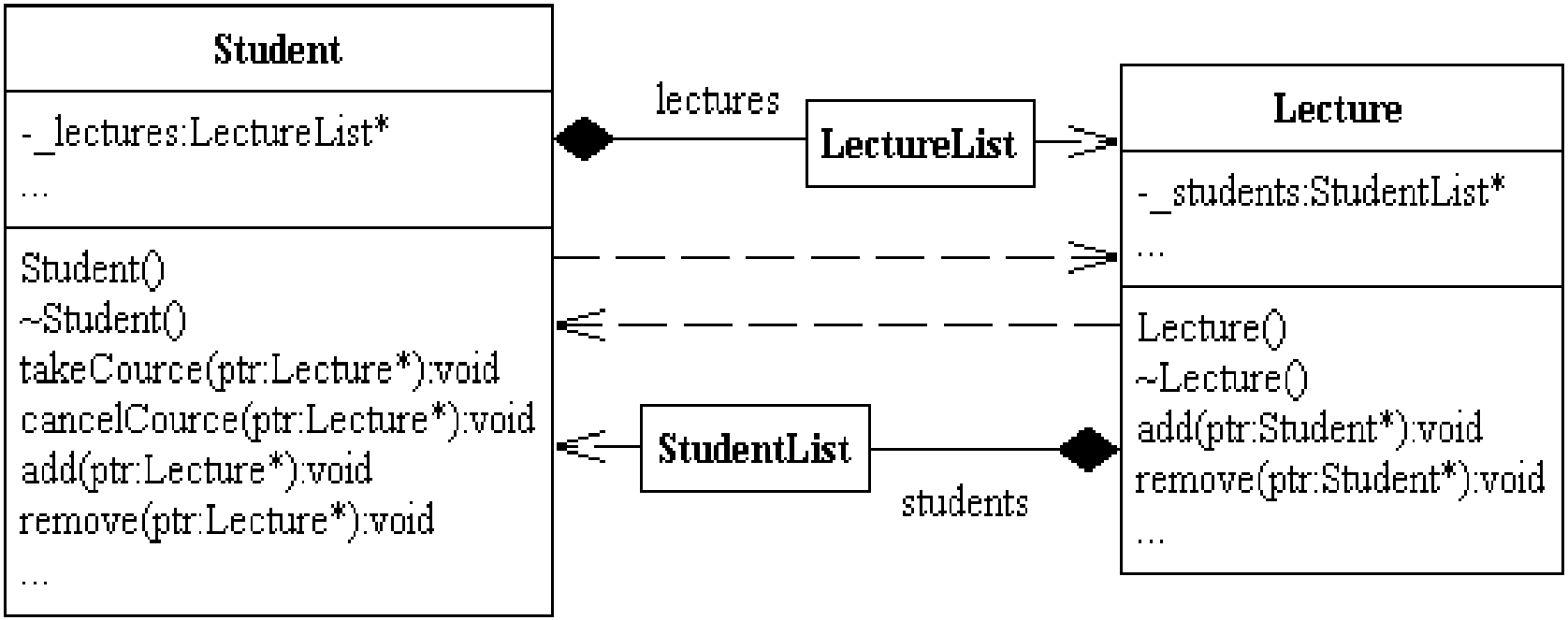
```
class DrawerView {
private:
    FigureList* _figures;     // 컨테이너 객체
public:
    DrawerView(...) {
        _figures = new FigureList();  // 컨테이너 객체 생성
        ...
    }
    ~DraweView() {
        delete _figures;      // 컨테이너 객체 소멸
        ...
    }
    void createFigure(int what) {    // 관계 설정 시 호출
        Figure* p = NULL;
        switch (what) {
        case POINT:
            p = new Point(...);
            break;
        case LINE:
            p = new Line(...);
            break;
            ...
        }
        p->setView(this);
        _figure->AddTail(p);  // 컨테이너에 새로 만들어진 그림 추가
    }
```

```
    void deleteFigure(Figure* ptr) {
        _figure->Remove(ptr); // 컨테이너로부터 그림 삭제
        delete ptr;
    }
    void drawView(CDC* pDC) {    // 폴리몰픽 컨테이너를 이용
        POSITION pos = _figure->GetHeadPosition();
        while (pos != NULL) {
          Figure* ptr = _figure->GetNext(pos);
          ptr->draw(pDC);
        }
    }
}
```

4.2.8 다대다(M:N) 결합관계

- A 클래스의 객체에 대해 B 클래스 객체가 여러 개 대응되며, B 클래스의 객체에 대해서도 A 클래스 객체가 여러 개 대응되는 경우
  - 백화점과 납품업체의 관계
  - 항공사와 공항과의 관계
  - 개설과목과 수강학생들과의 관계

<다대다 결합관계의 예>



<다대다 결합관계를 합성화와 의존관계로 변형한 예>

- 코드로 변환 시
  - 두 개의 폴리몰픽 리스트가 사용됨
  - 나머지는 일대다 결합관계와 유사하게 반영

```
class Student {
private:
    LectureList* _lectures; // 시간표 역할
    ...
public:
    Student() {
        _lectures = new LectureList();  // 컨테이너 객체 생성
        ...
    }
    ~Student() {
        POSITION pos = _lectures->GetHeadPosition();
        while (pos != NULL) {
            Lecture* ptr = _lectures->GetNext(pos);
            // 학생이 듣는 모든 강좌에서 학생 정보 제거
            ptr->remove(ptr);
        }
        delete _lectures;   // 컨테이너 객체 소멸
        ...
    }
    void takeCourse(Lecture* ptr) { // 관계 설정 시 호출
        _lectures->AddTail(ptr);    // 학생 시간표에 과목 정보 추가
        ptr->add(this);             // 강좌 출석부에 학생 정보 추가
    }
    void cancelCourse(Lecture* ptr) {   // 관계 해제 시 호출
        _lectures->Remove(ptr);     // 학생 시간표에 과목 제거
        ptr->remove(this);          // 강좌 출석부에 학생 정보 제거
    }
    void add(Lecture* ptr) {        // 컨테이너 객체에 추가
        _lectures->AddTail(ptr);
    }
    void remove(Lecture* ptr) {     // 컨테이너 객체에서 제거
        _lectures->Remove(ptr);
    }
}
```

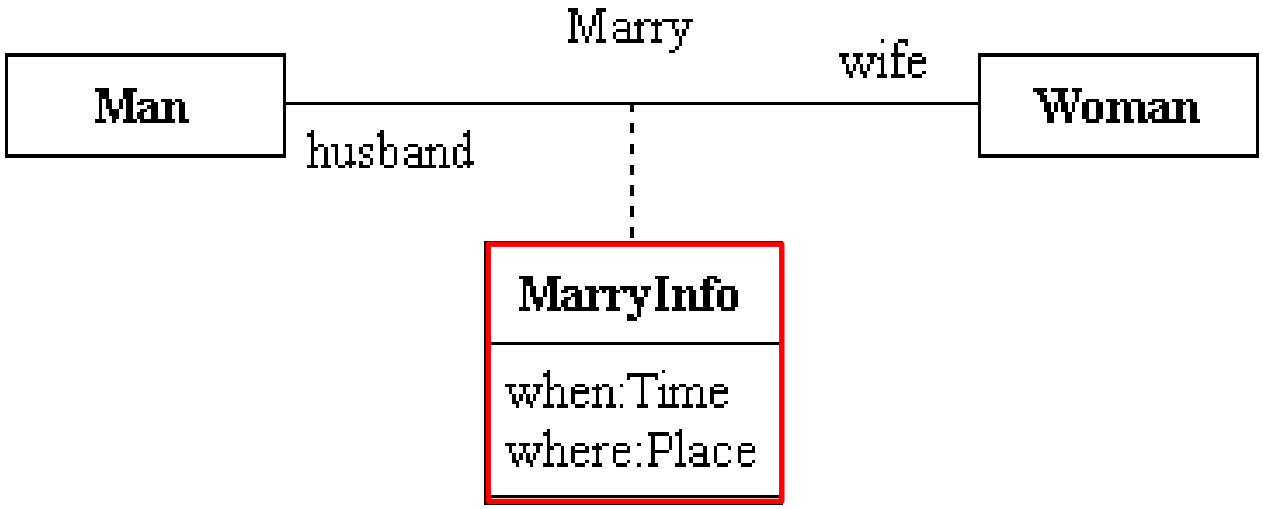
```
class Lecture {
private:
    StudentList* _students; // 출석부 역할
    ...
public:
    Lecture() {
        _students = new StudentList();  // 컨테이너 객체 생성
        ...
    }
    ~Lecture() {
        POSITION pos = _students->GetHeadPosition();
        while (pos != NULL) {
            Student* ptr = _students->GetNext(pos);
            // 이 강좌를 수강하는 학생들의 시간표에서 강좌 정보 제거
            ptr->remove(ptr);
        }
        delete _students;   // 컨테이너 객체 소멸
    }
    void add(Student* ptr) {    // 컨테이너 객체에 추가
        _students->AddTail(ptr);
    }
    void remove(Student* ptr) { // 컨테이너 객체에서 제거
        _students->Remove(ptr);
    }
    ...
}
```

❖ 결합관계를 코드에 반영 시 고려할 사항

- 결합관계에 명시된 명칭이 데이터 멤버나 함수 명칭으로 사용될 수 있는가?
- 결합관계의 방향성을 양방향으로 표현할 것인가, 단방향으로 표현할 것인가?
  - 양방향의 경우, dangling reference 생기지 않도록 조심
- 다중성에 따라 포인터 배열을 이용할 것인가? 폴리몰픽 리스트를 이용할 것인가?
  - 컨테이너의 명칭을 적절하게 선택해야
- 결합관계의 설정 및 해제를 위한 함수를 작성해야 함

4.2.9 결합 클래스

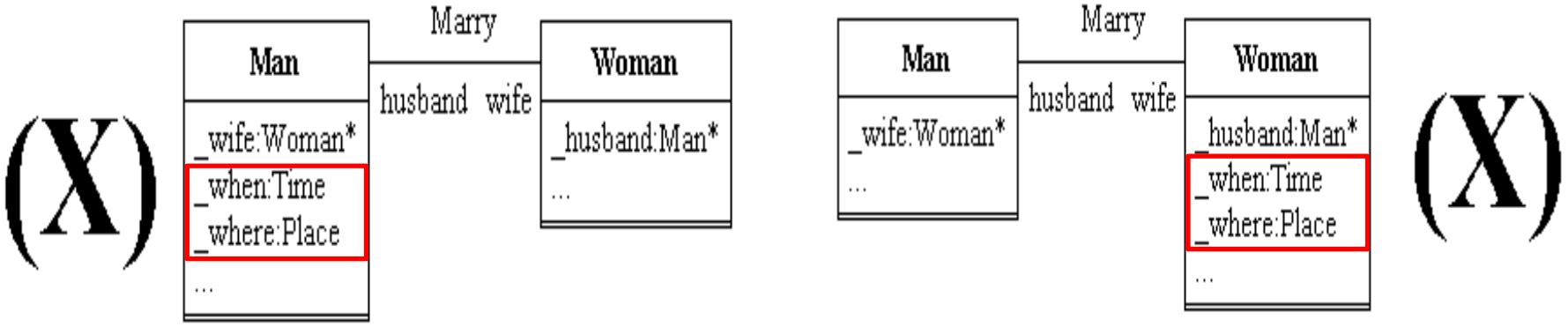
- 관계가 설정되는 시점에 관계 자체에 대한 정보를 기록하는 용도
- 예) “결혼 관계(Marry)”에서 언제, 어디서 결혼했는지에 대한 정보를 MarryInfo 결합 클래스에 저장



<결합 클래스의 사용 예>

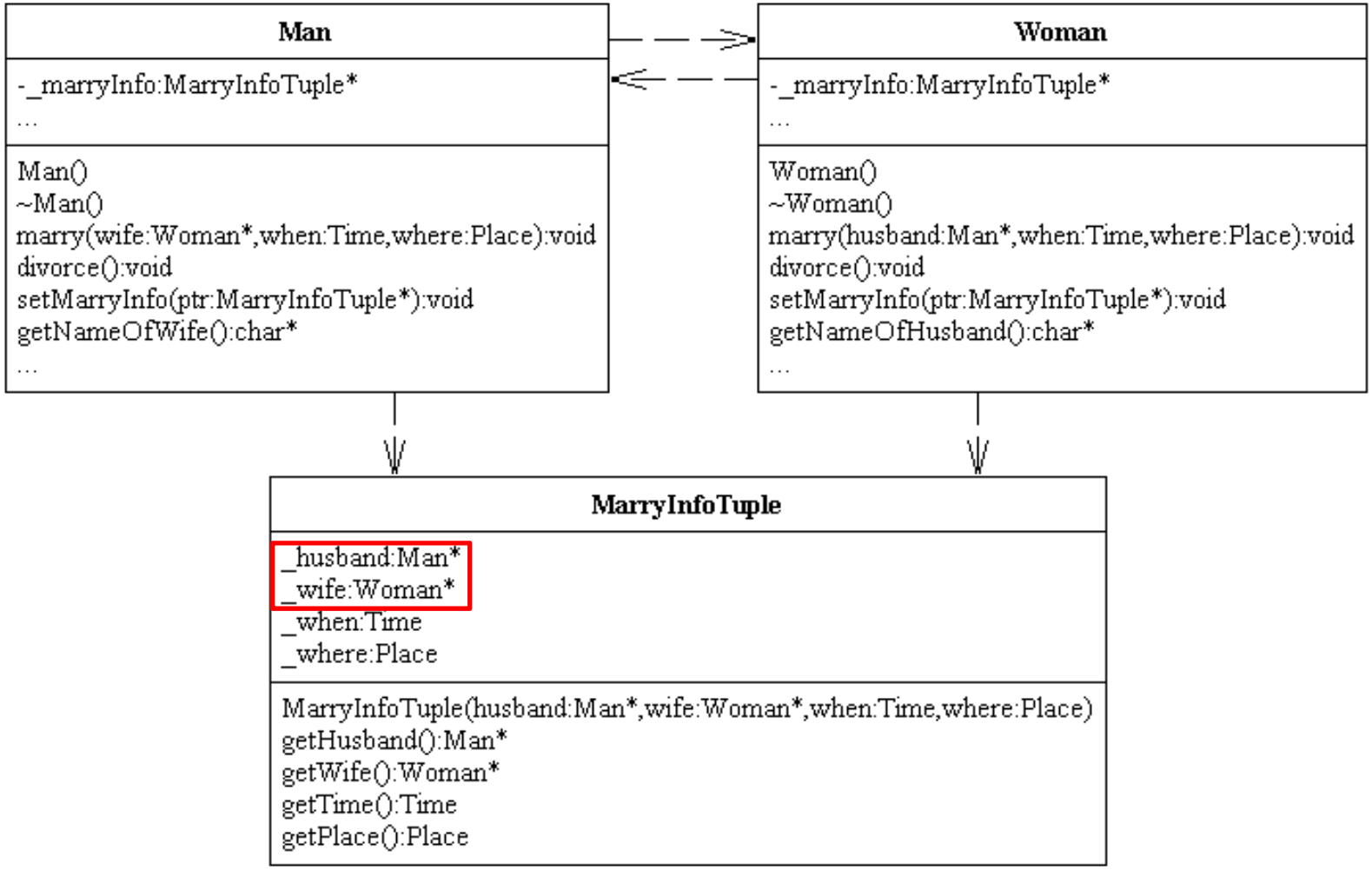
- 결혼에 대한 관계 정보가 남자 객체나 여자 객체에 의존적인 정보가 아님
  - 아래 예에서 독신자인 경우 해당 정보 불필요
- 따라서 결혼 시기와 장소 정보는 두 남녀가 결혼 관계를 맺는 시점에 생성되어야 함

  → 결합 클래스



<관계 정보를 잘못 표현한 모델의 예>

- 코드 변환
  - 튜플(tuple) 객체 필요



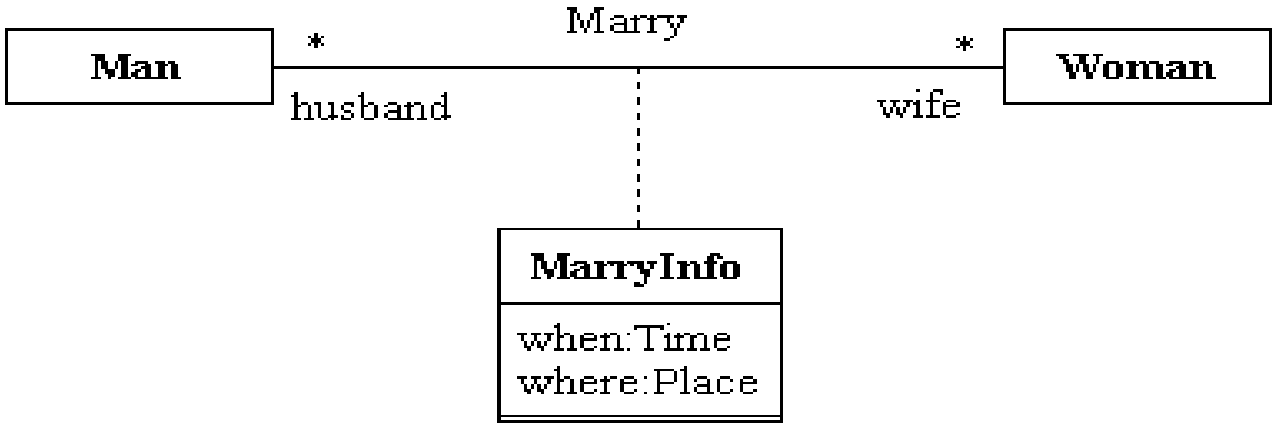
<튜플 클래스를 이용한 결합 클래스의 변환>

```
class Man {
private:
    MarryInfoTuple* _marryInfo;  // 결혼 정보를 저장하는 객체
    ...
public:
    Man() {
        _marryInfo = NULL;          // 총각임을 의미
        ...
    }
    ~Man() {
        if (_marryInfo != NULL) {
            // 남자가 죽는 경우, 그 부인의 결혼 정보와의 링크 제거
            Woman* wife = _marryInfo->getWife();
            wife->setMarryInfo(NULL);
            delete _marryInfo;
        }
        ...
    }
    void marry(Woman* wife, Time when, Place where) { // 관계 설정 시 호출
        if (_marryInfo != NULL) {
            throws SomeException;       // 중혼을 의미하므로 예외 발생
        }
        marryInfo = new MarryInfoTuple(this,wife,when,where);
        wife->setMarryInfo(_marryInfo);
    }
```

관계 맺는 시점에 관계 정보 생성

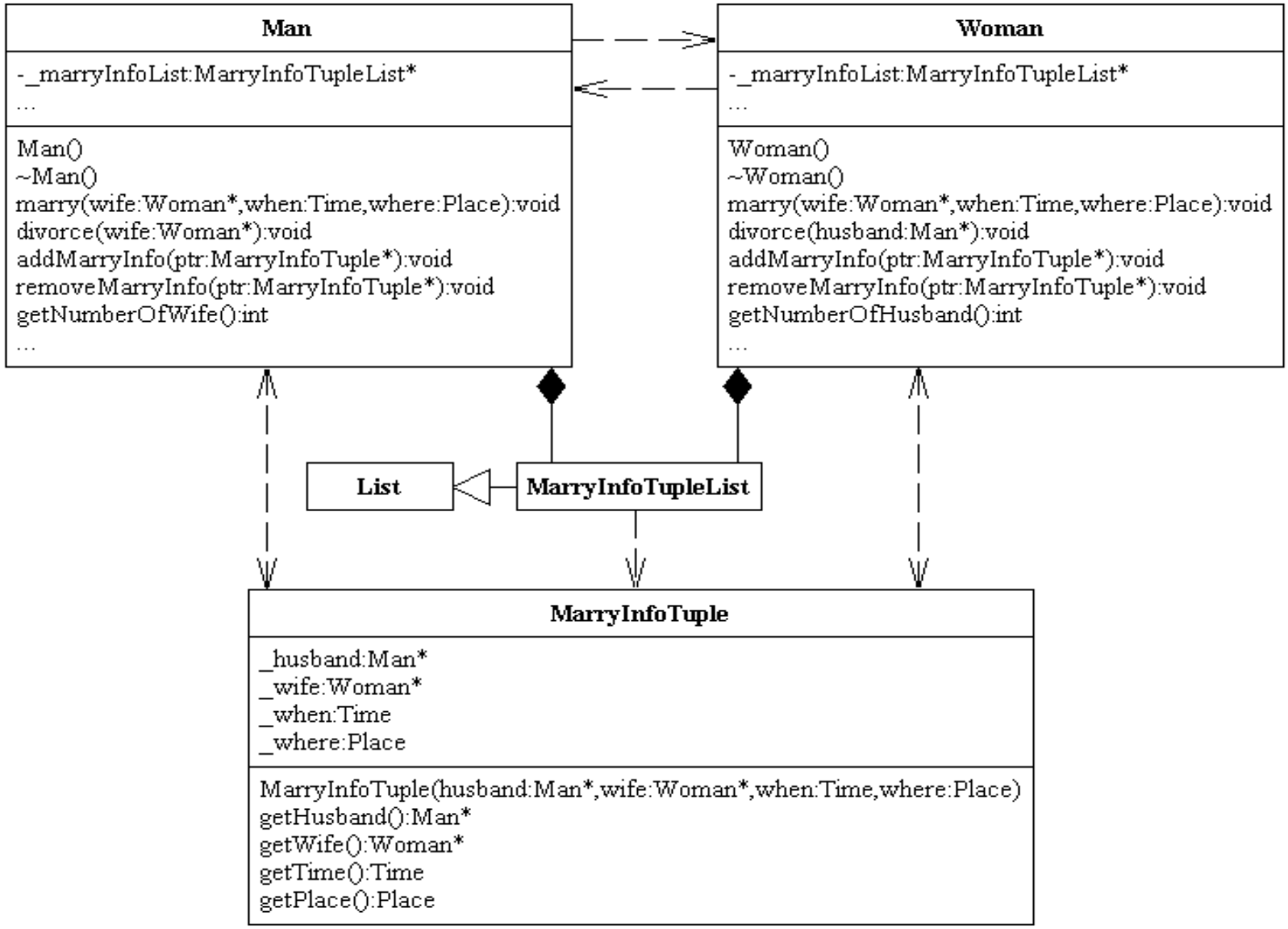
```
    void divorce()  {                       // 관계 해지 시 호출
        if (_marryInfo == NULL) {
            throws SomeException;           // 총각이 이혼을? : 예외 처리
        }
        Woman *wife = _marryInfo->getWife();
        wife->setMarryInfo(NULL);
        delete _marryInfo;
        _marryInfo = NULL;                  // 돌싱?
    }
    void setMarryInfo(MarryInfoTuple* ptr) {
        _marryInfo = ptr;
    }
    char* getNameOfWife() {                 // 튜플 객체의 사용 예
        if (_marryInfo == NULL) return NULL;
        return _marryInfo->getWife()->getName();
    }
    ...
};
```

- 결합관계의 다중성이 다대다인 경우
  - 컨테이너 객체 이용



<다대다 결합관계에 결합 클래스가 사용된 모델의 예>

  - 튜플 객체에 두 객체가 관계를 맺고 있다는 사실과 관계 정보를 함께 저장해야 함
  - 관계를 맺는 객체는 튜플 객체들의 리스트를 유지해야 함



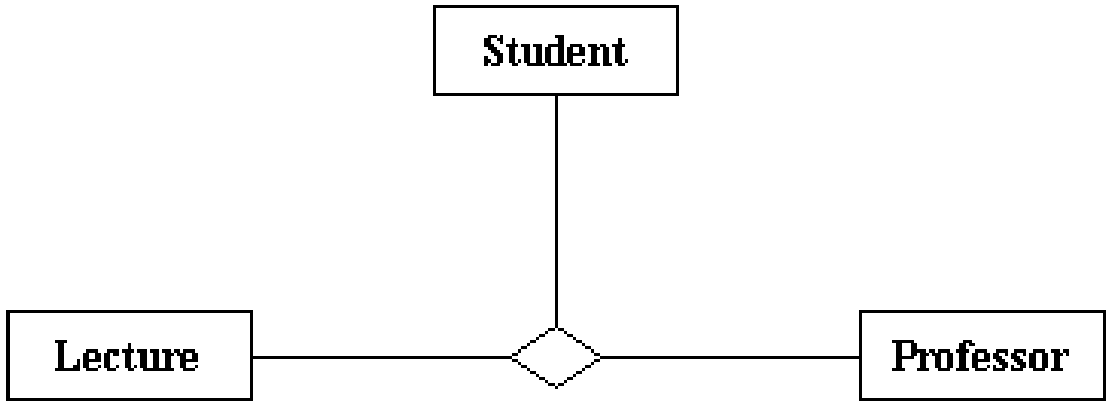
<결합 클래스를 갖는 다대다 결합관계의 변형 예>

```
class Man {
private:
    MarryInfoTupleList* _marryInfoList;  // 결혼정보 저장하는 튜플들의 리스트
    ...
public:
    Man() {
        _marryInfoList = new MarryInfoTupleList(); // 리스트가 비어있으면 충각
        ...
    }
    ~Man() {
        POSITION pos = _marryInfoList->GetHeadPosition();
        while(pos != NULL) {
            MarryInfoTuple *ptr = _marryInfoList->GetNext(pos);
            // 남자가 죽는 경우, 그 부인들로부터 링크된 결혼 정보 링크 제거
            Woman *wife = ptr->getWife();
            wife->removeMarryInfo(ptr);
        }
        delete _marryInfoList;
        ...
    }
    void marry(Woman* wife,Time when,Place where) {  // 관계 설정 시 호출
        MarryInfoTuple *ptr = new MarryInfoTuple(this,wife,when,where);
        _marryInfoList->AddTail(ptr);
        wife->addMarryInfo(ptr);
     }
```

```
    void divorce(Woman* wife) {                    // 관계 해지 시 호출
      // 여러 부인 중 특정인과 이혼하는 상황
      MarryInfoTuple *ptr = _marryInfoList->findTupleForWife(wife);
      Woman *wife = ptr->getWife();
      _marryInfoList->Remove(ptr);
      wife->removeMarryInfo(ptr);
      delete ptr;
    }
    void addMarryInfo(MarryInfoTuple* ptr)  {    // 컨테이너 객체에 추가
      _marryInfoList->AddTail(ptr);
    }
    void removeMarryInfo(MarryInfoTuple* ptr)  { // 컨테이너 객체에서 제거
      _marryInfoList->Remove(ptr);
    }
    int getNumberOfWife()  {              // 컨테이너 객체의 사용 예
      return _marryInfoList->GetCount();
    }
    ...
};
```

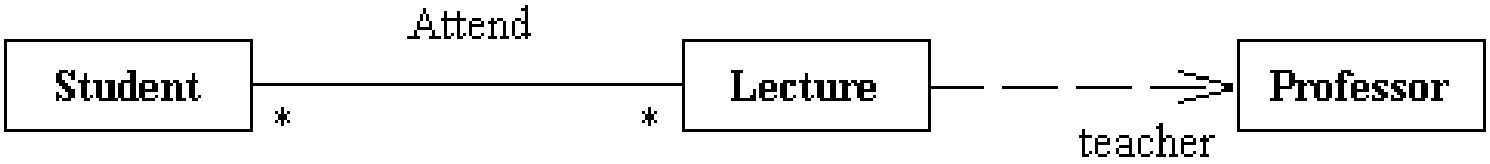
4.2.10 삼각관계

- 3개의 객체가 한꺼번에 관계를 맺는 상황
  - 삼각관계는 모든 객체가 서로 독립적이어야 한다.
  - 실제로 흔하지는 않으며, 아래의 경우를 잘못 해석한 경우 발생
    - 2개의 이진관계
    - 결합 클래스를 이용한 이진관계
  - 관계의 다중성이 다대다대다
- 사각관계 이상의 모델은 삼각관계나 이진관계로 변환하는 것이 바람직

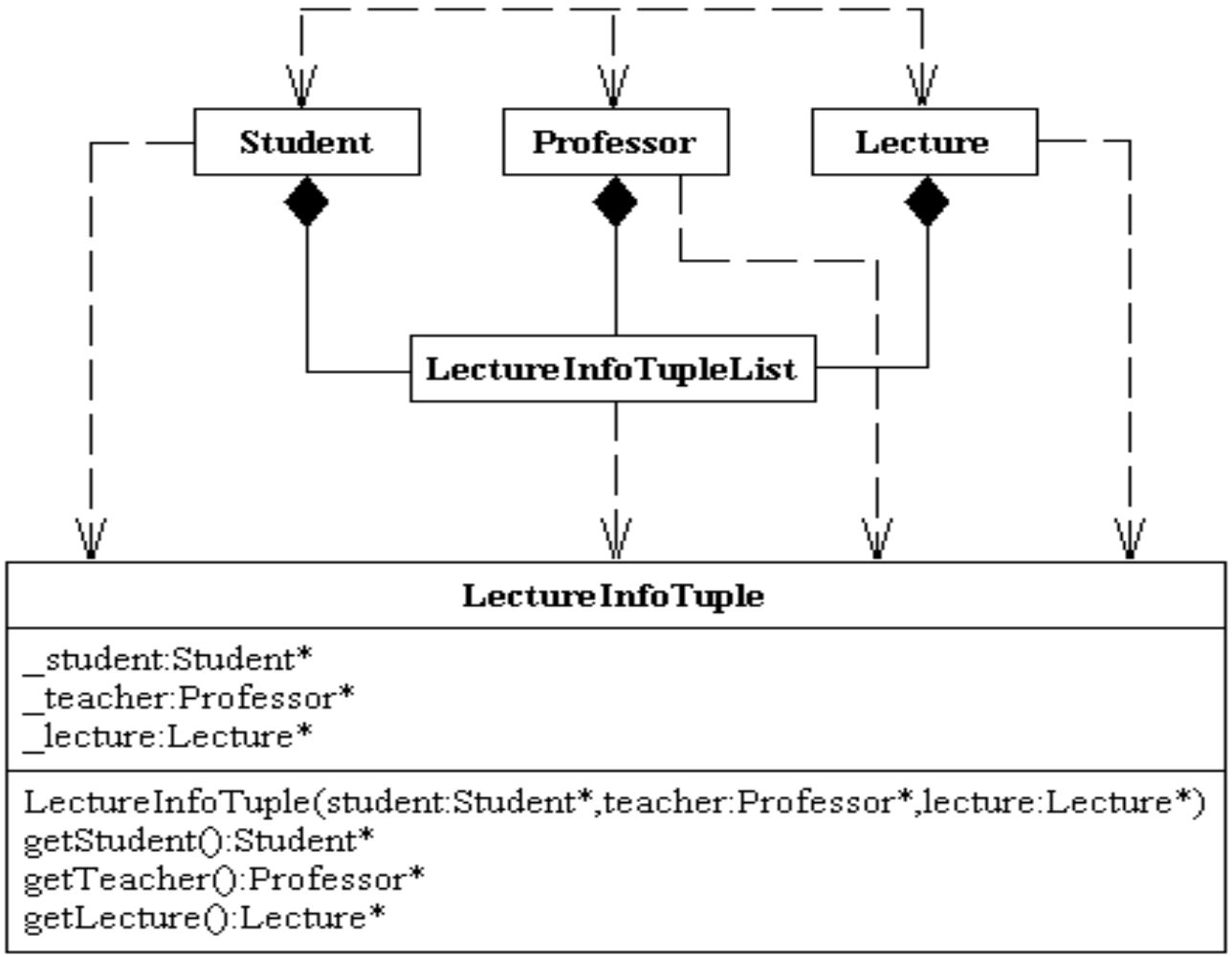


<삼각관계를 잘못 사용한 예>

- 학생과 강좌 객체는 서로 독립적
- 강좌와 교수 객체는 종속적
  - 특정 강좌에 해당 교수가 배정되기 때문
  - 팀 티칭(team teaching)의 경우는?



<삼각관계의 이진관계 환원 예>
한 강좌를 한 교수가 강의하는 경우



<삼각관계의 변형 예>

다대다 결합관계의 변환과 비슷하게 처리

Student 클래스 구현 예 : 교재 p.287~288

[요약]

- 클래스 다이어그램
  - 구체적인 클래스 모델
    - 일반화, 집합화, 의존관계
    - 기계적으로 코드 변환, 또는 간단한 방법으로 코드 변환 가능
  - 추상적인 클래스 모델
    - 결합관계, 다중성, 결합 클래스, 삼각관계
    - 코드로 바로 반영하기 어려움

4.3 모델링 팁

- Tip 1: 무엇이 가장 중요한가 ?
  - 구현된 코드
- Tip 2: UML 다이어그램은 어떠한 순서로 그려야 하는가 ?
  - 교재 p.291, 그림 4.42
- Tip 3: 클래스 다이어그램은 어떠한 순서로 그려야 하는가 ?
- Tip 4: 트리의 중요성은 무엇인가 ?
  - 그래프 형태보다 트리가 이해하는 데 도움
- Tip 5: 훌륭한 OOP의 판단 기준은 ?
  - 자료추상화, 상속, 동적 바인딩, 다형 개념이 적용

- Tip 6: 유지보수 작업의 가장 큰 문제점은 ?
  - 도큐먼트 부재와 소스 코드 중복
- Tip 7: 기능분해는 더 이상 사용되지 않는가 ?
  - 구조적 기법이지만 집합화와 합성화 모델링 시 사용
- Tip 8: 객체지향 구현의 추가 적용은 바람직한가?
  - 구조적 시스템에 적용하면 유지보수가 더 어려워짐
- Tip 9: 상속의 깊이는 ?
  - 재사용 극대화할 수 있으나, 너무 깊으면 구현이 어려움
- Tip 10: 객체를 재사용할 것인가 클래스를 재사용할 것인가 ?
  - 객체 재사용: 쉽게 구현, 데이터 멤버/생성자 복잡
  - 클래스 재사용: 상속 깊이 깊어짐, 재사용도 증가
- Tip 11: CASE 도구는 ?