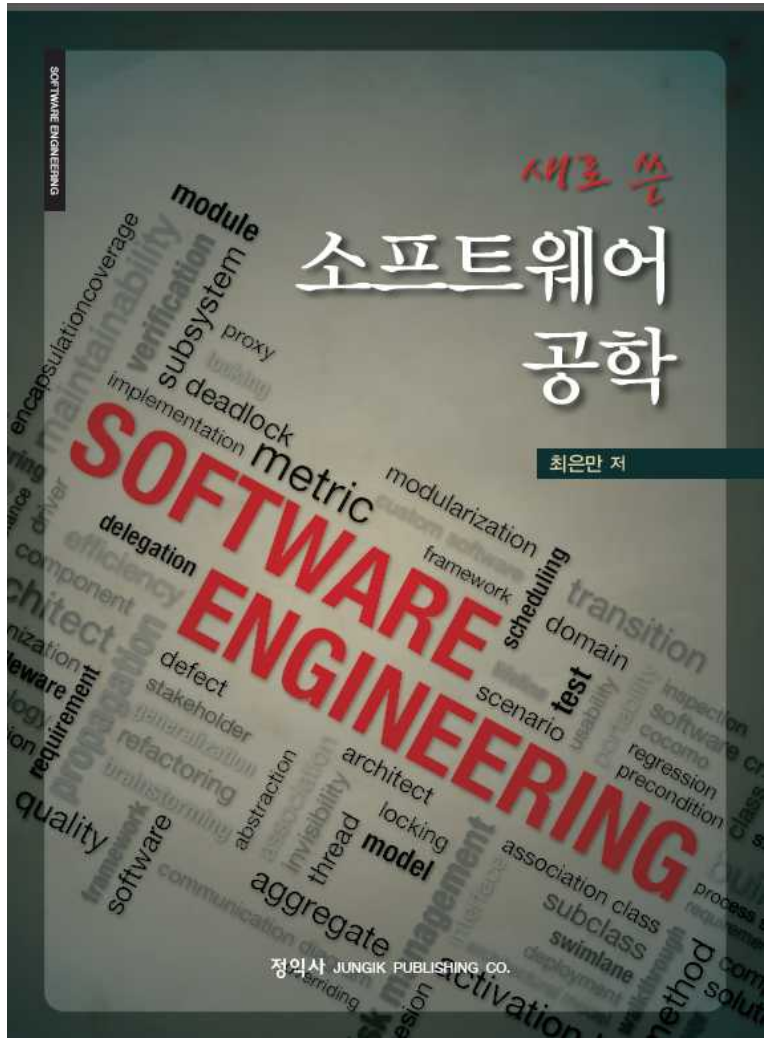
소프트웨어 공학 개론

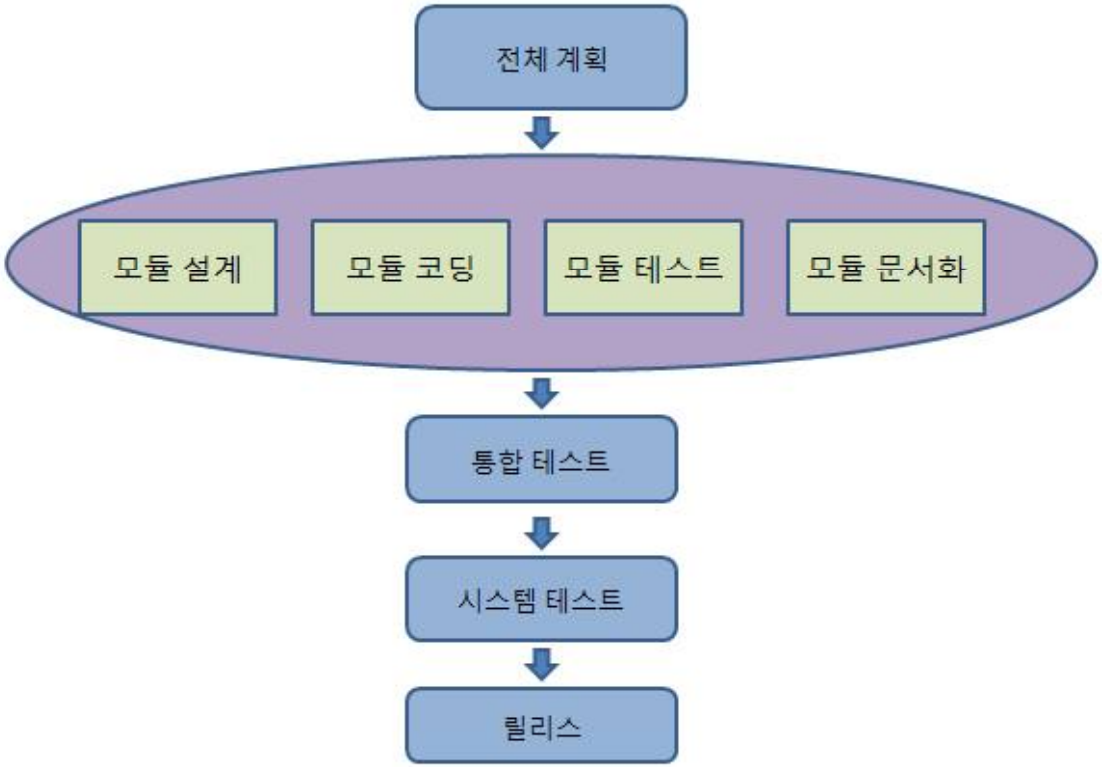


강의 11: UML 코드 매핑

최은만
동국대학교 컴퓨터공학과

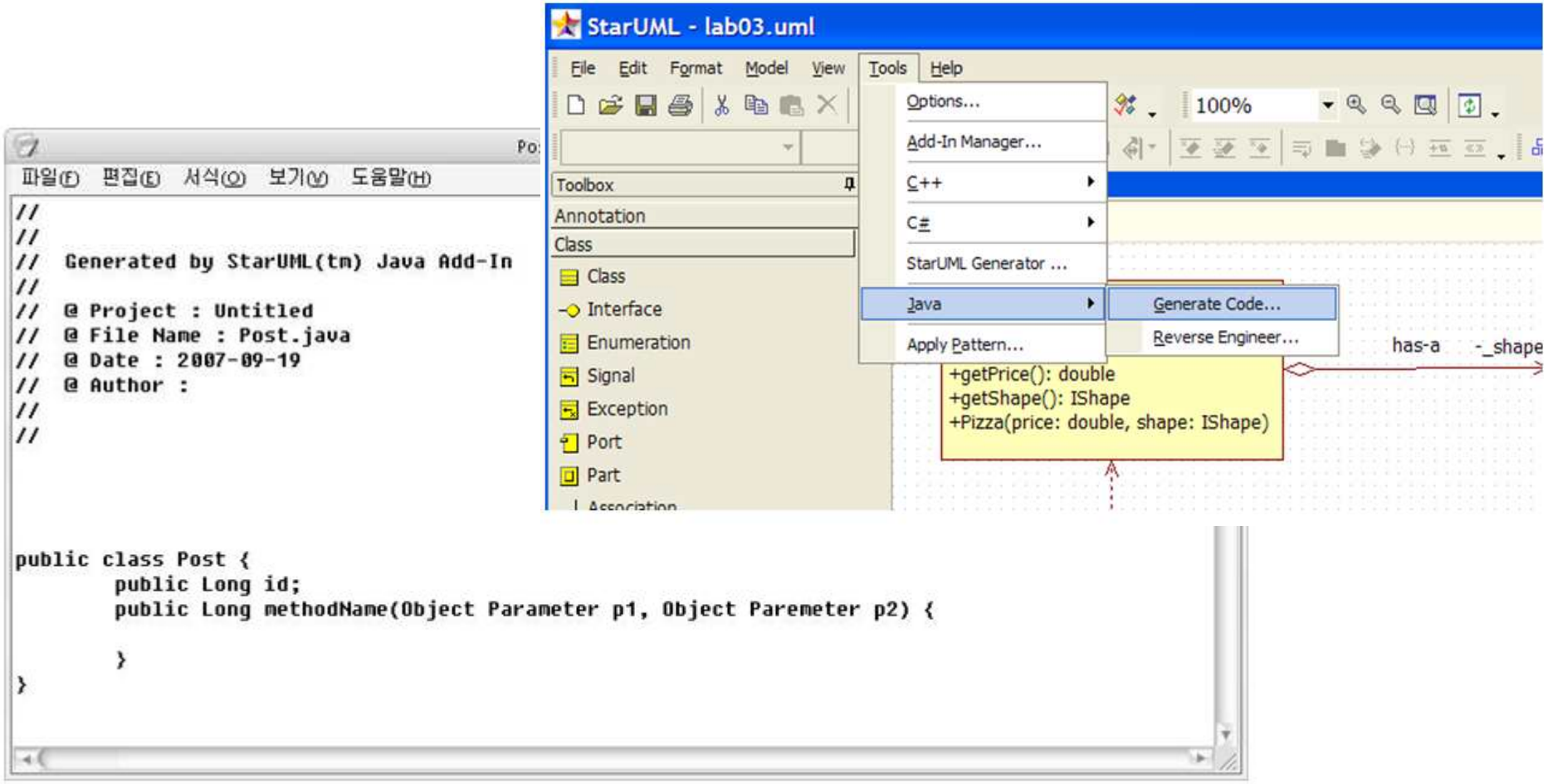
구현 작업

- 작업 이후 본격적으로 시스템을 구축하는 단계
- 프로그램, 즉 코드 모듈을 구축하는 과정



StarUML 코드 생성

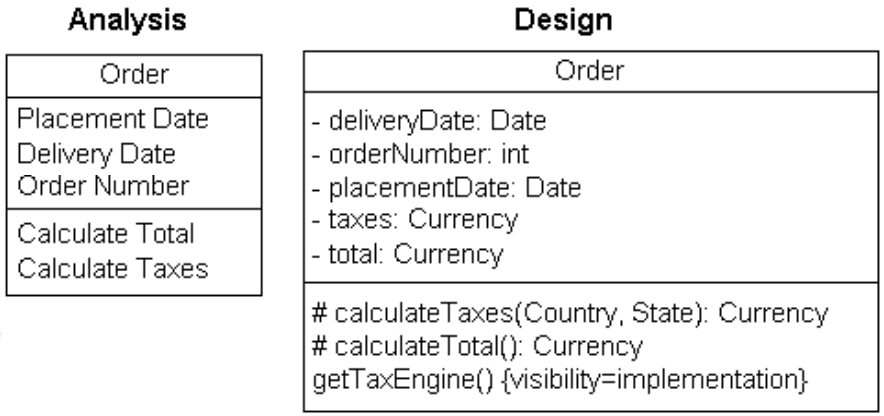
- **Tools->Java->Generate Code**



정적 모델의 구현

- 설계를 프로그램으로 매핑
  - 클래스 다이어그램과 패키지 다이어그램이 프로그램과 밀접

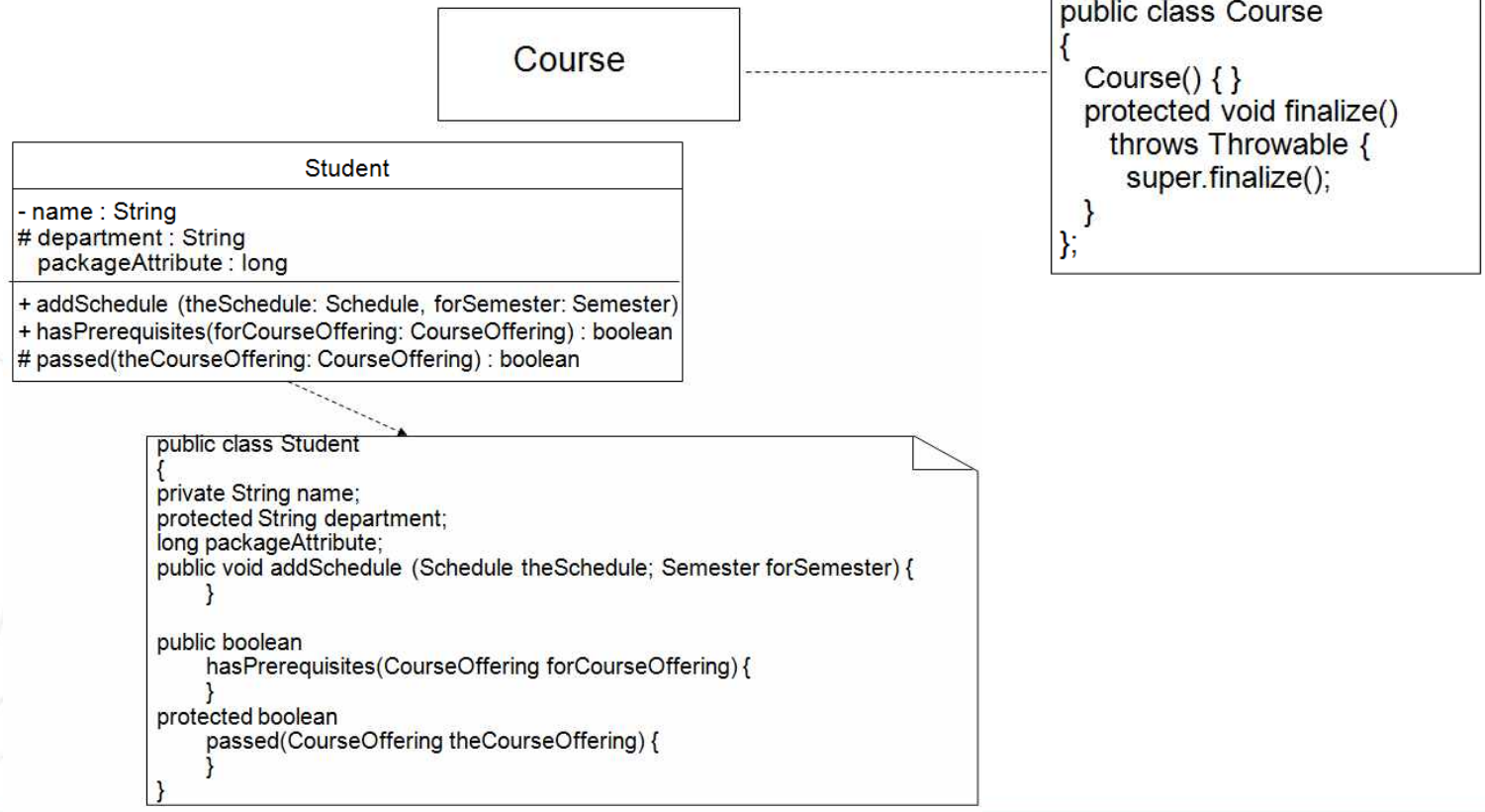
- 추상 수준에 따라 구현에 도움이 되는 정도가 다름



  - 개념 수준 – 도메인 개념
  - 명세 수준 – 인터페이스와 타입
  - 구현 수준 – 구현에 종속적인 사항을 포함

클래스의 구현

- 클래스 코드의 골격
  - 속성 – 클래스 안의 인스턴스 변수
  - 오퍼레이션 – 클래스 안의 메소드



상속

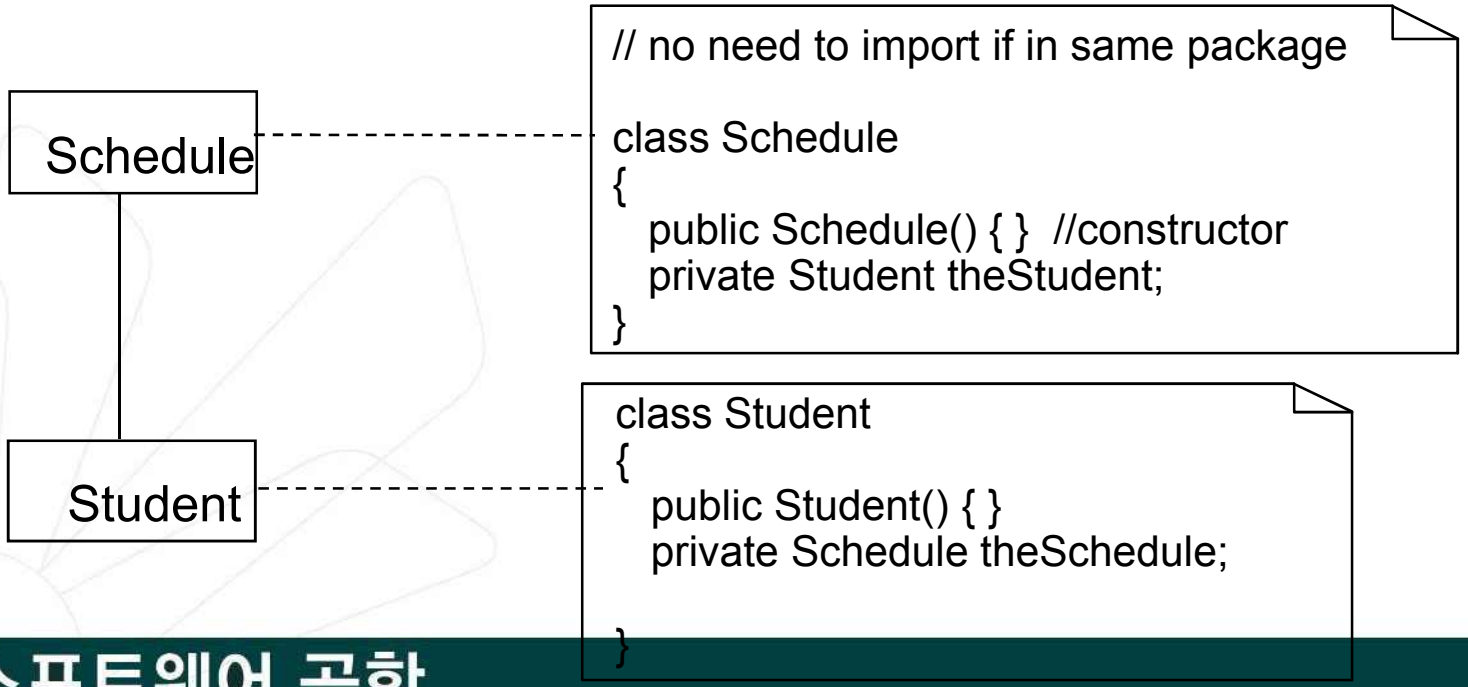
- UML 표현이 언어 문법에 1대 1 매칭


```
public class User {
    private String email;
    public String getEmail() {
            return email;
    }
    public void setEmail(String value){
            email = value;
    }
    public void notify(String msg) {
            // ....
    }
    /* Other methods omitted */
}
```

```
public class LeagueOwner extends User {
    private int maxNumLeagues;
    public int getMaxNumLeagues() {
            return maxNumLeagues;
    }
    public void setMaxNumLeagues
                    (int value) {
            maxNumLeagues = value;
    }
    /* Other methods omitted */
}
```

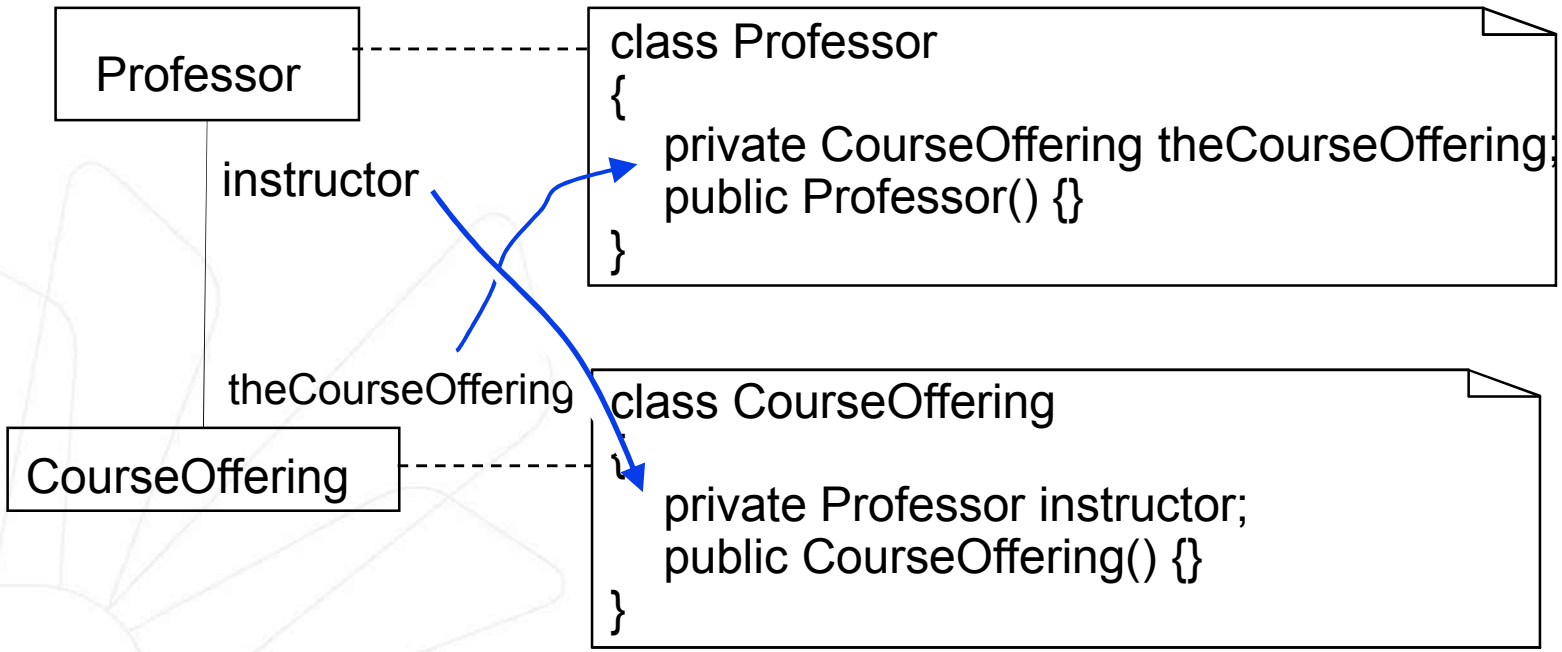
연관의 구현

- 연관
  - 두 클래스에 속한 객체들이 정보를 추적할 수 있게 하려는 것
- 양방향 연관
  - 두 클래스가 같은 패키지 안에 있어야
  - 양방향은 두 클래스가 서로 상대편 객체를 알고 있어야 하므로 서로를 알기 위한 public 메소드와 private 변수가 있어야



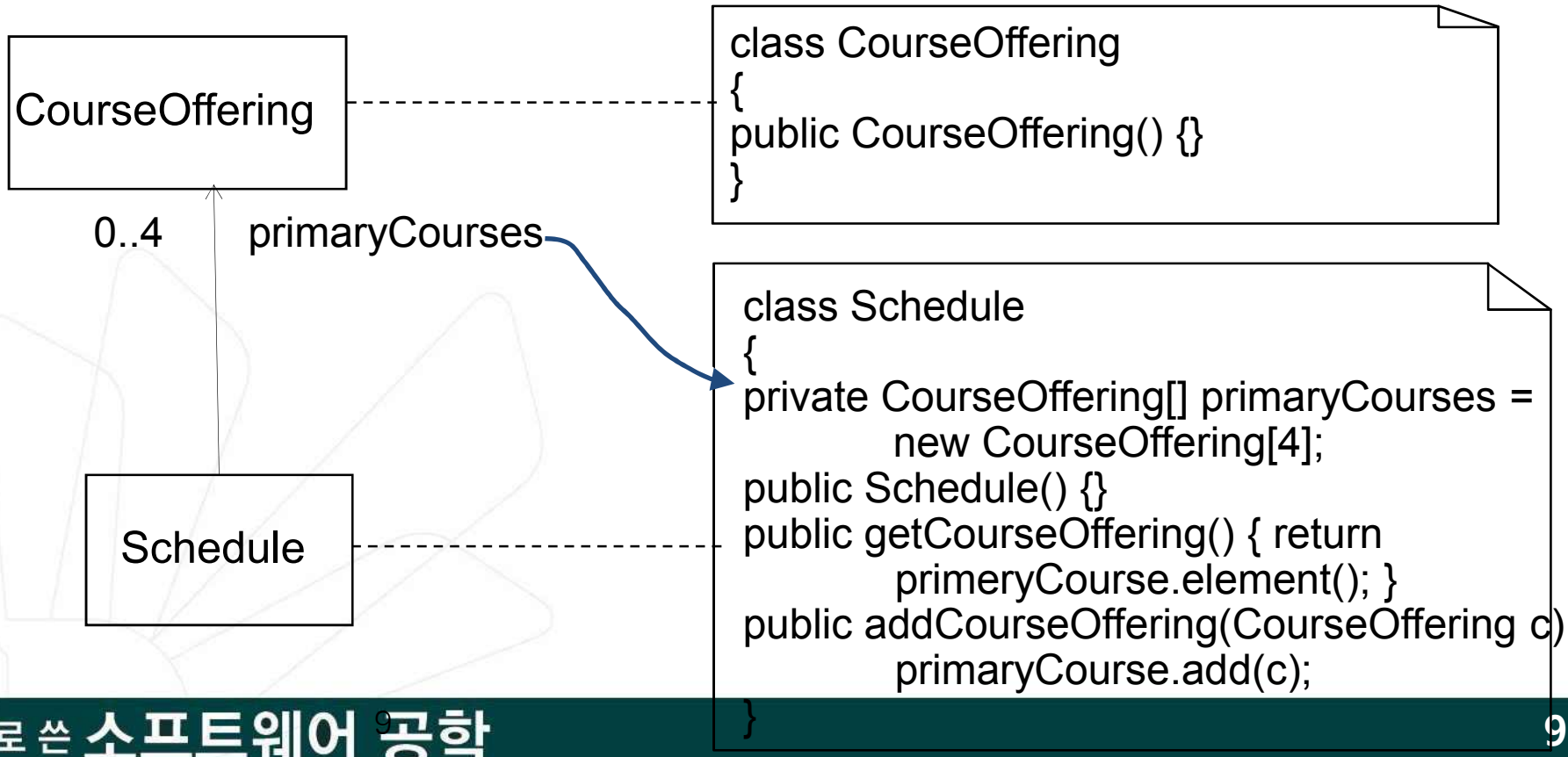
연관의 역할

- 역할의 추가
  - 각 클래스가 다른 클래스에 대하여 알고 있어야
  - 예) CourseOffering 클래스는 public 메소드 CourseOffering()가 있어야 하며 private 타입의 Professor 객체를 가지고 있어야
  - Professor 클래스는 public 메소드 Professor()와 private 타입의 CourseOffering 객체를 알고 있어야 함



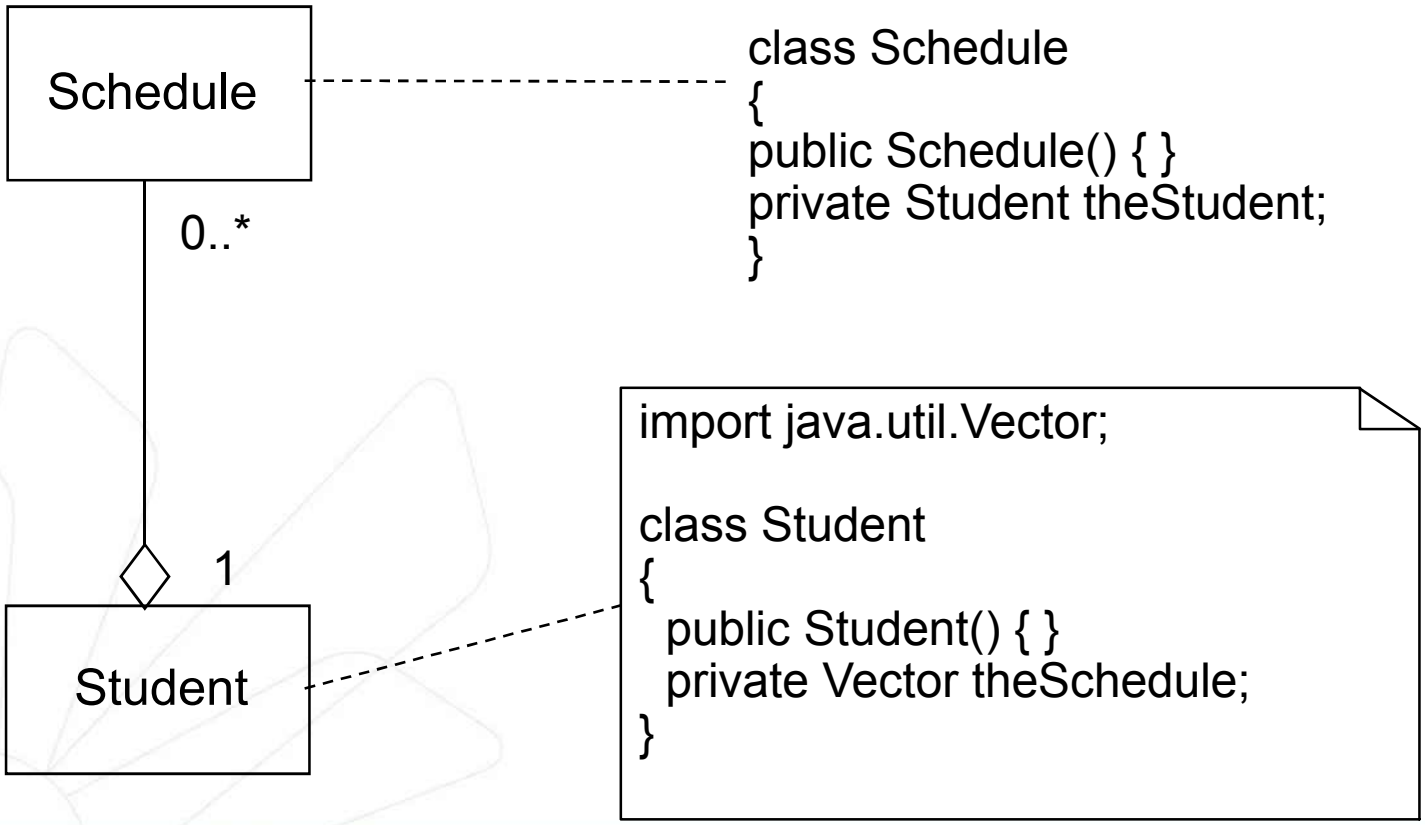
다중도의 구현

- 사례
  - CourseOffering 클래스는 public method CourseOffering()을 가짐
  - 클래스 Schedule에는 네 개의 CourseOffering 객체를 가질 수 있는 배열 또는 리스트가 있어야 함



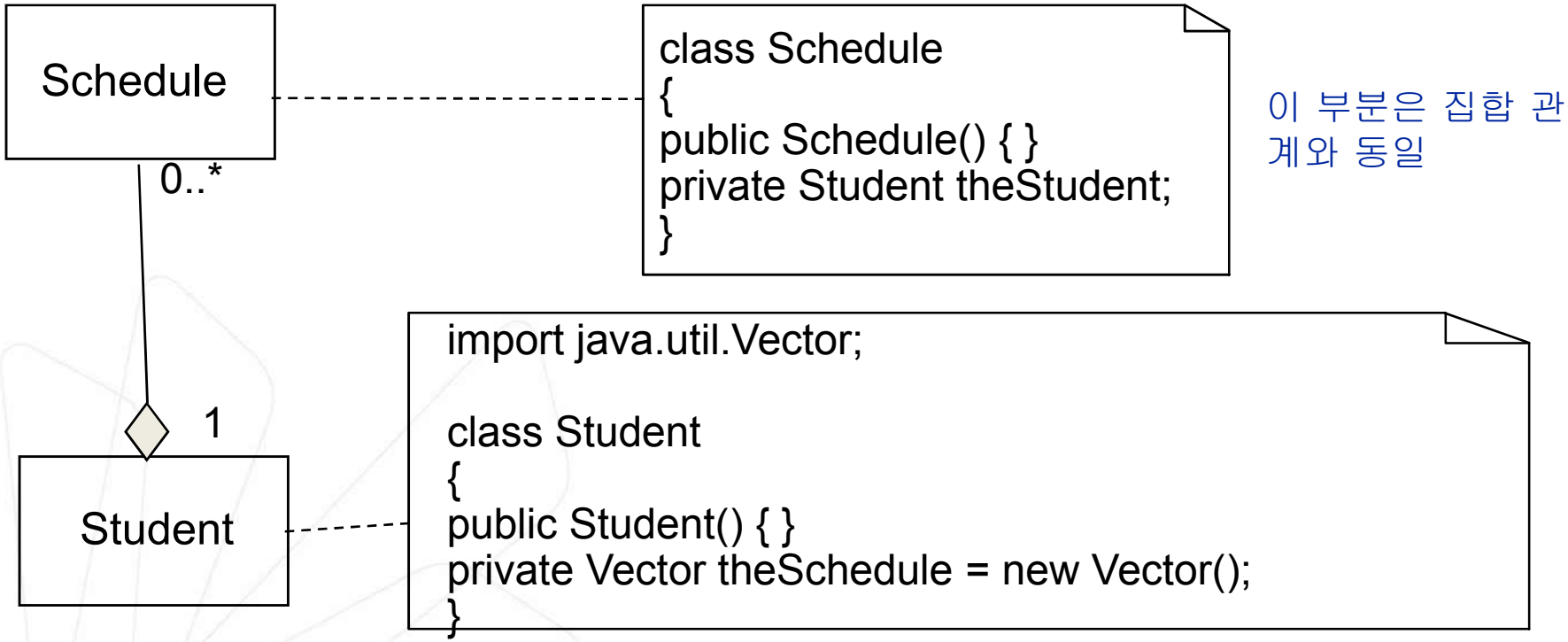
집합 관계의 구현

- 'Student'클래스는 Vector 리스트 타입의 클래스를 선언
- 집합을 위한 생성자는 따로 없음. 객체의 배열로 취급
- 집합 관계의 구현은 연관 관계와 차이가 없음



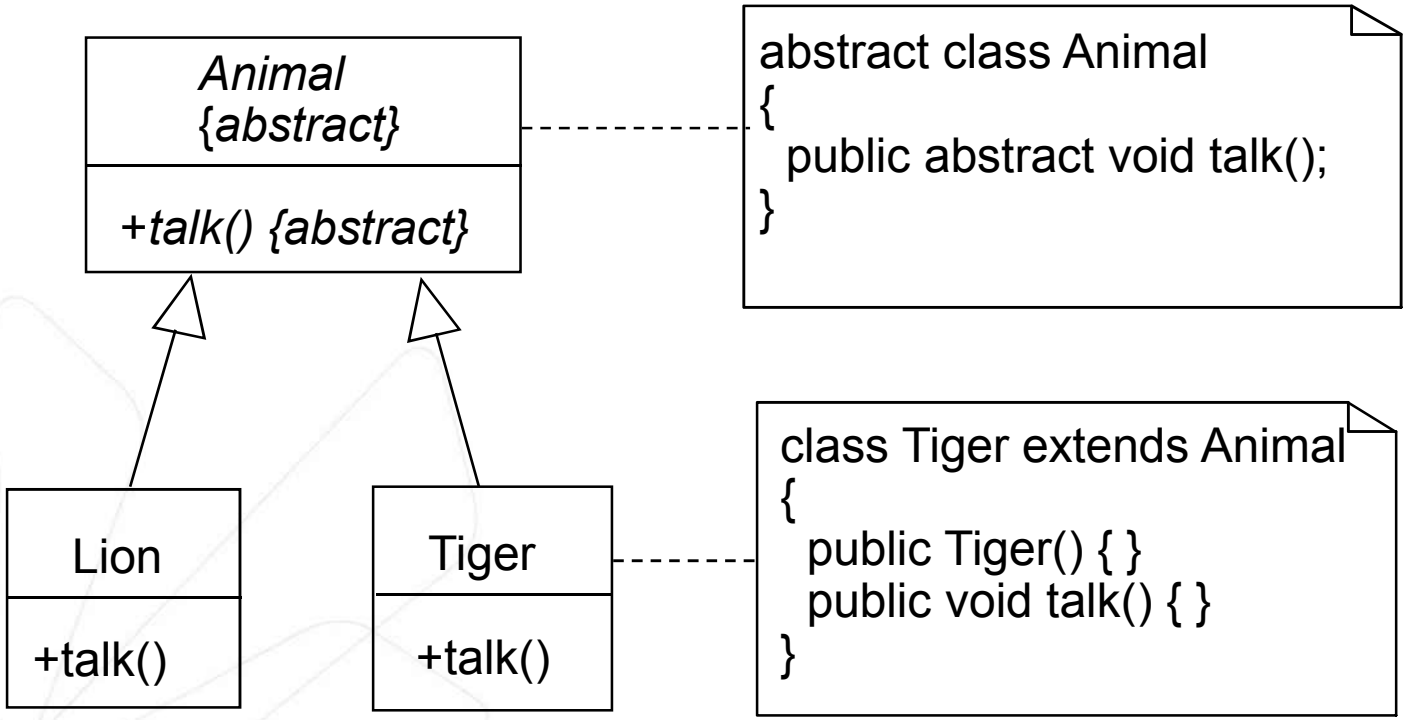
합성 관계

- 객체의 존폐에 대한 제어가 있는 집합 관계
  - 집합 개념의 클래스가 요소 개념의 객체의 생성과 소멸을 제어



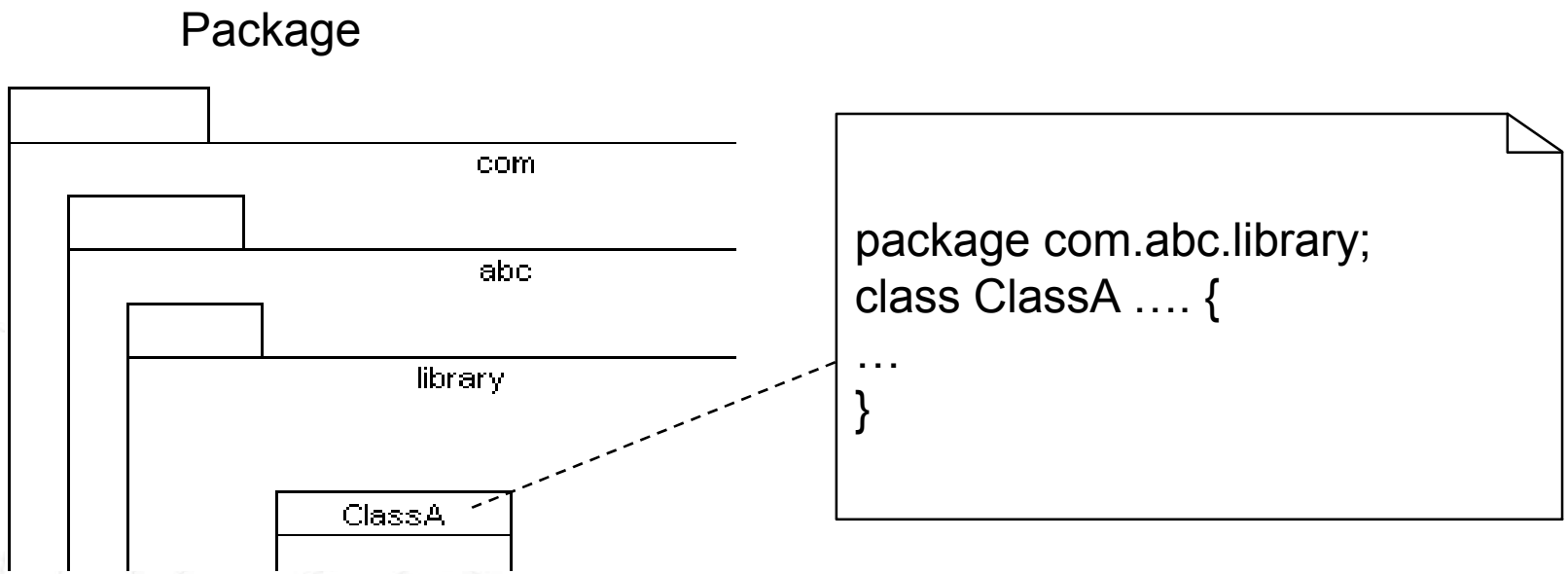
추상 클래스의 구현

- 객체가 생성되는 클래스가 아니라 상속된 클래스가 구체적으로 구현하도록 메소드와 변수의 선언만 있는 템플릿
- <<Abstract>> 프로토타입 표기



패키지의 구현

- 패키지 다이어그램
  - 클래스의 모임인 패키지의 구성과 의존 관계 표시
  - import 또는 include 사용
- 패키지 선언



동적 모델링의 구현

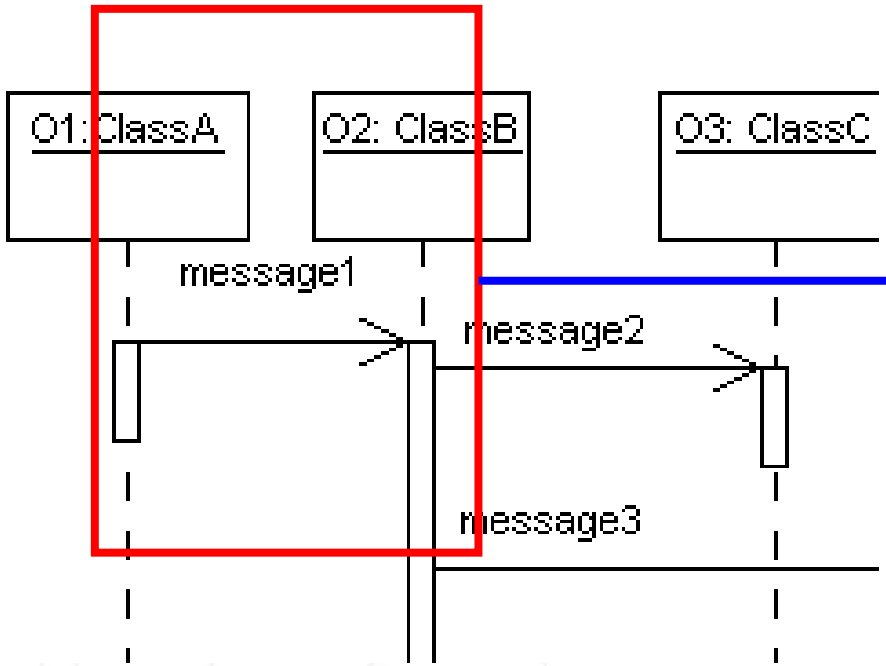
- 순서 다이어그램
  - 메시지의 호출을 표현
- 상태 다이어그램
  - 상태를 표현하는 법
  - 상태의 변환을 메소드로 구현하는 법
- 액티티비 다이어그램
  - 제어흐름을 메소드에 구현하는 법

순서 다이어그램의 구현

- 순서 다이어그램
  - 협력하는 객체들 사이의 메시지 교환을 나타낸 것
  - 메시지는 화살표 나가는 객체에서 들어오는 객체로 메소드 호출
  - 메시지를 받는 객체는 제 3의 객체에게 하나 이상의 메시지를 호출할 수 있음

- 순서 다이어그램을 코딩하는 방법
  - 메시지는 메소드의 호출로 코딩. 객체의 생성은 생성자(constructor)를 호출
  - 메시지를 받는 객체의 클래스 안에 메소드 구현
  - 분기구조는 if-else 문장과 같은 조건문으로 구현
  - 병렬구조는 thread로 구현

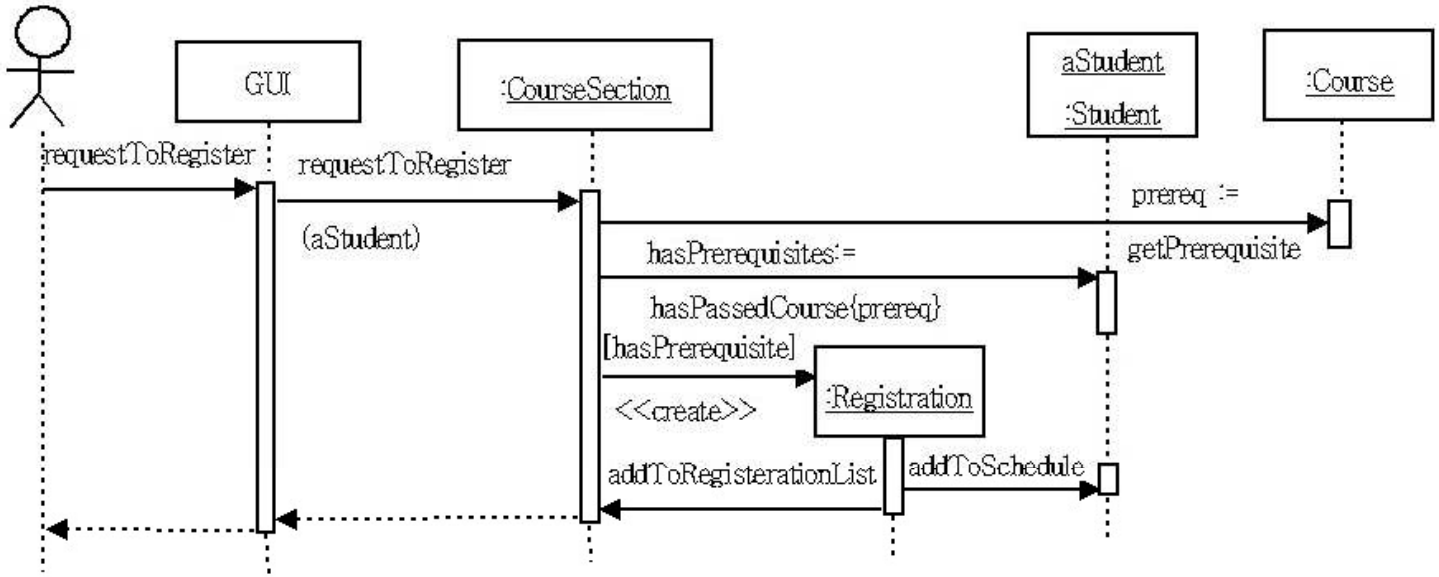
순서 다이어그램의 구현




```
ClassB
{
  ClassC o3;
  ClassD o4;
  method1(…)
  {
    o3.method2(..);
    o4.method3(..);
  }
}
```

- 메시지가 호출 당하는 클래스 안에 메소드 구현

사례 – 수강 신청 과정


GUI
:CourseSection
aStudent
:Student
:Course
requestToRegister
requestToRegister
(aStudent)
prereq :=
getPrerequisite
hasPrerequisites:=
hasPassedCourse(prereq)
[hasPrerequisite]
<<create>>
:Registration
addToRegisterationList
addToSchedule

구현 된 코드

```
public class CourseSection
{
    // The many-1 abstraction-occurence association
  private Course course;

  // The 1-many association to class Registration
  private List registationList;

  // The following are present only to determine
  // the state
  // The initial state is 'Planned'
  private boolean open = false;
  private boolean closedOrCancelled = false;
 ...
}

public CourseSection(Course course)
{
  this.course = course;
  RegistrationList = new LinkedList();
}
```

구현된 코드

```
public void cancel()
{
  // to ‘Cancelled’ state
  open = false;
  closedOrCancelled = true;
  unregisterStudents();
}
public void openRegistration()
{
  if(!closedOrCancelled)
  // must be in ‘Planned’ state
  {
    open = true;
    // to 'OpenNotEnoughStudents' state
  }
}
```

```
public void closeRegistration()
  {
    // to 'Cancelled' or 'Closed' state
    open = false;
    closedOrCancelled = true;
    if (registrationList.size() <
      course.getMinimum())
    {
      unregisterStudents();
      // to ‘Cancelled’ state
    }
  }
```

구현된 코드

```
public void requestToRegister(Student student)
{
    if (open)  // must be in one of the two 'Open' states
    {
     // The interaction specified in the sequence diagram
     Course prereq = course.getPrerequisite();
     if (student.hasPassedCourse(prereq))
     {
        // Indirectly calls addToRegistrationList
        new Registration(this, student);
     }

     // Check for automatic transition to 'Closed' state
     if (registrationList.size() >= course.getMaximum())
     {
      // to ‘Closed’ state
      open = false;
      closedOrCancelled = true;
     }
    }
}
```

구현된 코드

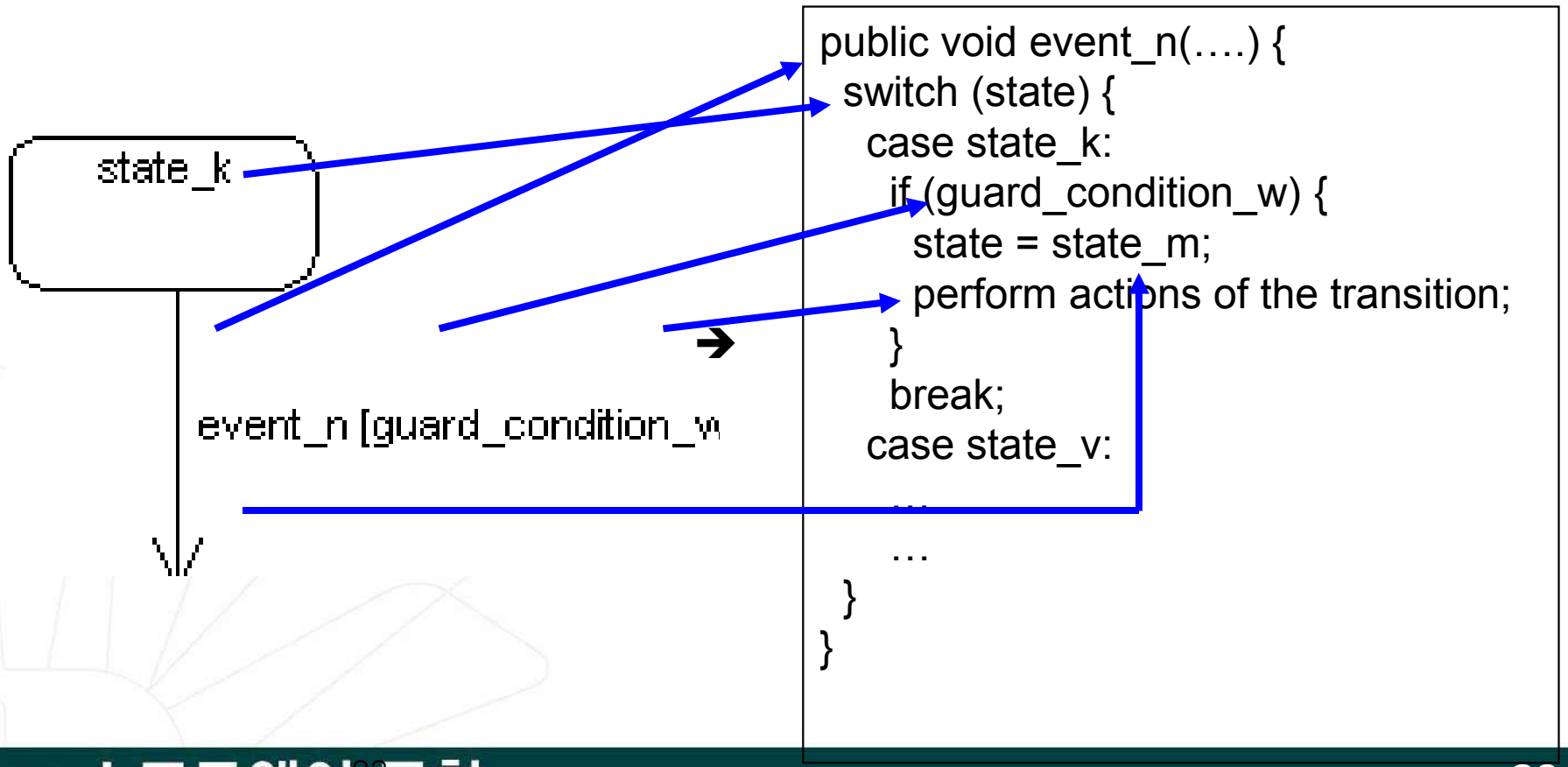
```
// Activity associated with 'Cancelled' state.
private void unregisterStudents()
{
  Iterator it = registrationList.iterator();
  while (it.hasNext())
  {
    Registration r = (Registration)it.next();
    r.unregisterStudent();
    it.remove();
  }
}

// Called within this package only, by the
// constructor of Registration
  void addToRegistrationList(
    Registration newRegistration)
  {
    registrationList.add(newRegistration);
  }
}
```

상태 다이어그램의 구현

- Inactive 객체의 상태 다이어그램을 구현하는 방법
- 상태 다이어그램을 클래스로 매핑
- 상태정보를 저장하기 위한 속성 추가
- 이벤트는 메소드로 상태 변화나 이벤트의 액션은 메소드 안에 탑재

상태 다이어그램의 구현

- 모든 상태는 상태를 나타내는 속성의 값
- 상태 변화는 클래스의 메소드
- 가드는 메소드 안의 조건 체크



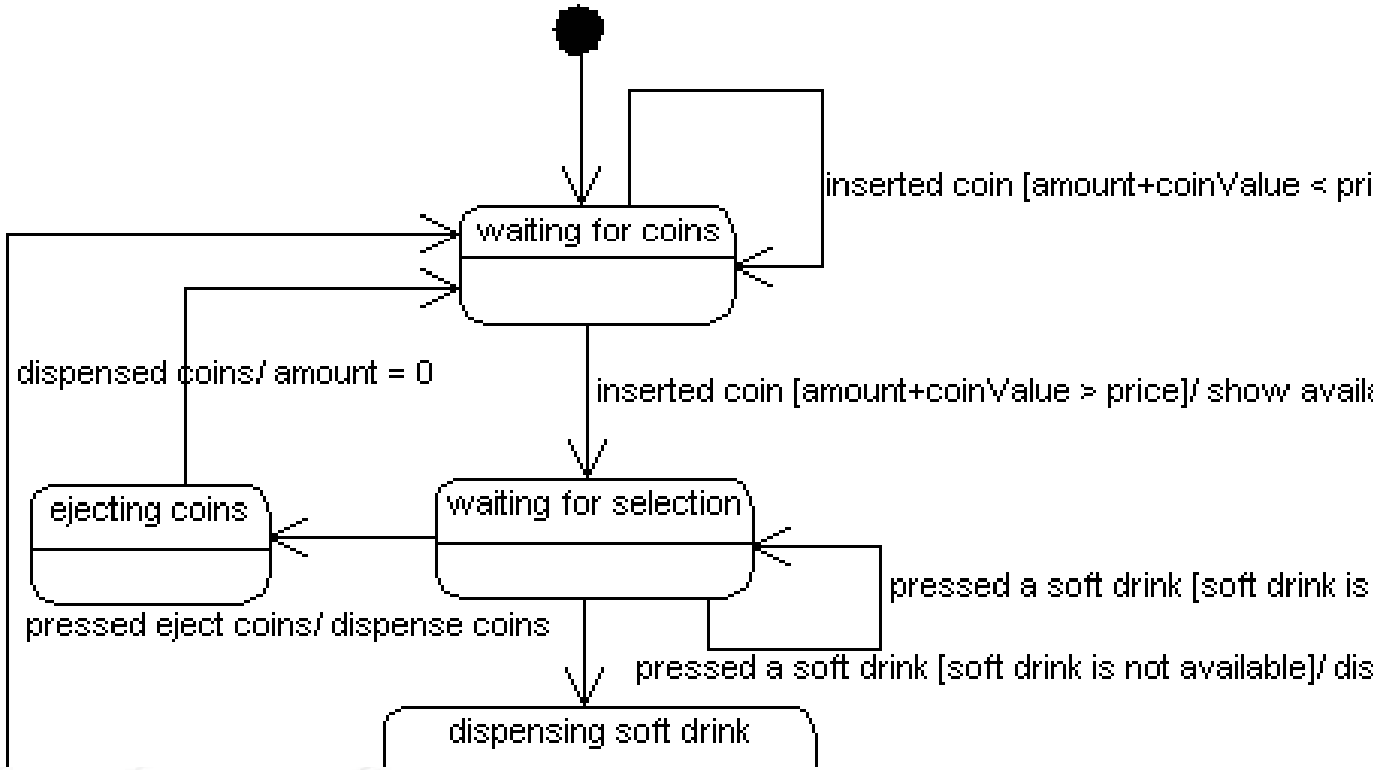
내장된 상태다이어그램의 구현

- 순차 서브상태를 가진 복합 상태(composite state)
  - 서브 상태를 구현하기 위한 내장 클래스 생성
  - 부모 상태 머신에서 내부 상태 다이어그램의 상태 변화를 제어하기 위한 메소드(내부 상태 클래스 안의) 호출
  - 다른 구현 방법은 복합 상태를 제거하기 위하여 내장 상태 다이어그램을 펼쳐서 구현

- 병렬 서브상태를 가진 복합 상태
  - 각 서브 상태를 구현하기 위한 내장 클래스 생성
  - 내장 상태 머신의 구현과 유사
  - 모든 병렬 서브 상태가 종료되면 복합 상태를 빠져 나오도록 코딩

사례 – 자판기 제어

- 자판기 제어 객체의 상태 변화
- 내장 상태(dispense soft drink)를 가진 복합 객체



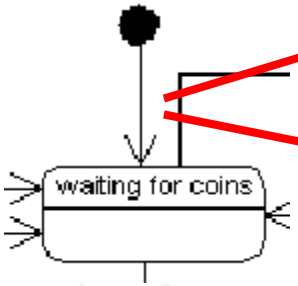
자판기 제어 객체

- 상태 다이어그램을 하나의 클래스로 구현
- 상태 정보를 저장하는 _state 속성 추가

```
class VendingMachineControl
{
  int _state;
  float _amount, _price;
  static final int WaitingCoin = 1;
  static final int WaitingSelection = 2;
  static final int DispensingSoftDrink = 3;
  static final int DispensingChange = 4;
  static final int EjectingCoins = 5;
```

상태 초기화

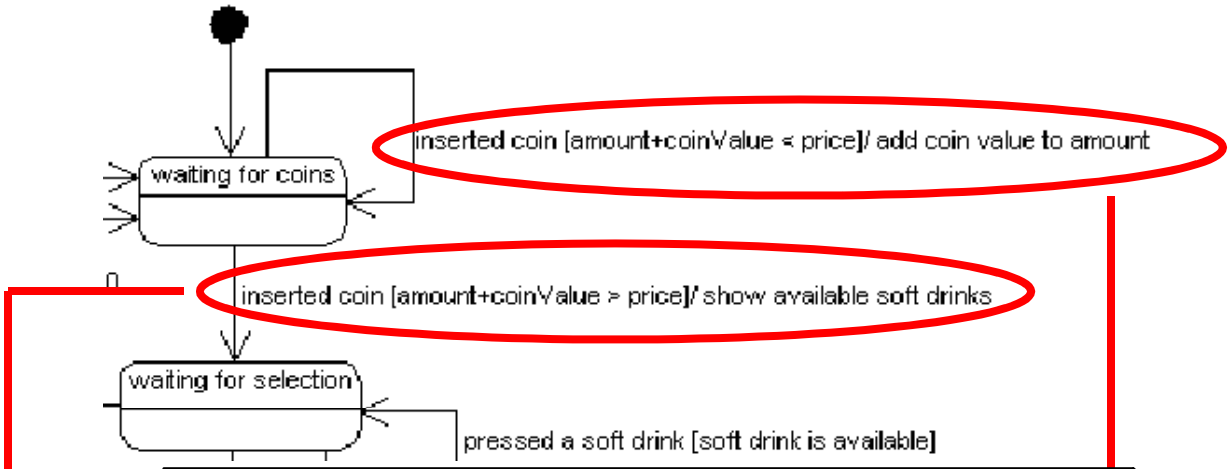
- 생성자 안에서 상태 변수 및 기타 변수 초기화




```
public VendingMachineControl(float pric
e)
  {
    _amount = 0;
    _state = WaitingCoin;
    _price = price;
  }
```

이벤트는 메소드로

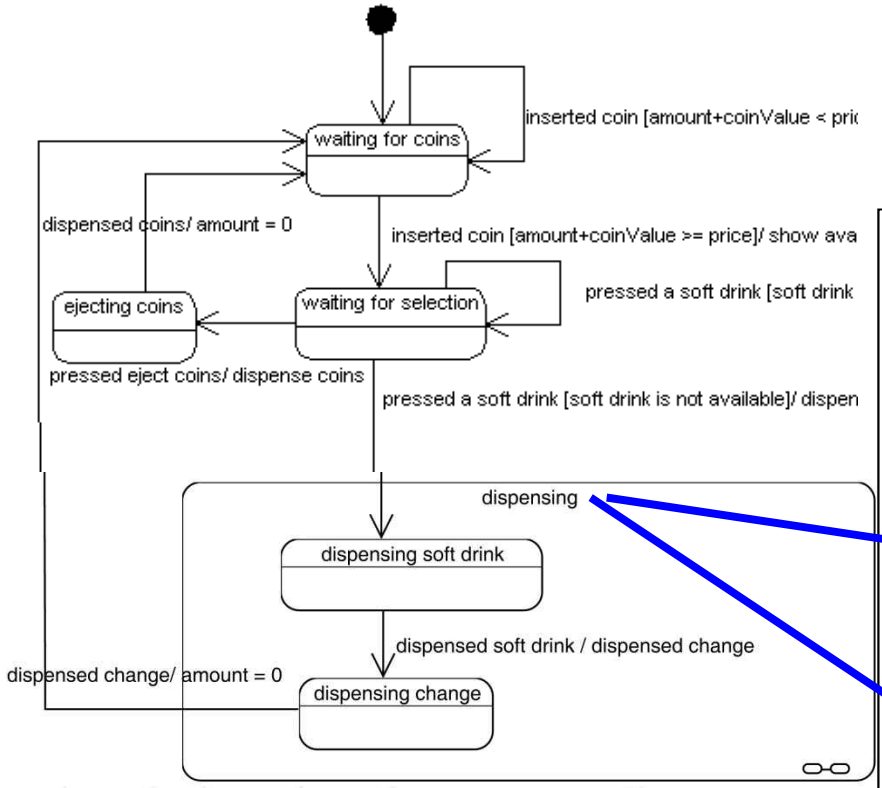
- 상태천이와 이벤트의 액션은 메소드 안에 구현




```
public void insertedCoin(float coinValue)
 {
   if (state == WaitingCoin)
   {
     amount += coinValue;
     if (amount >= price) {  // fire transition
       state = WaitingSelection;
       show available soft drinks;
     }
   }
 } // insertedCoin
```

서브 상태의 구현

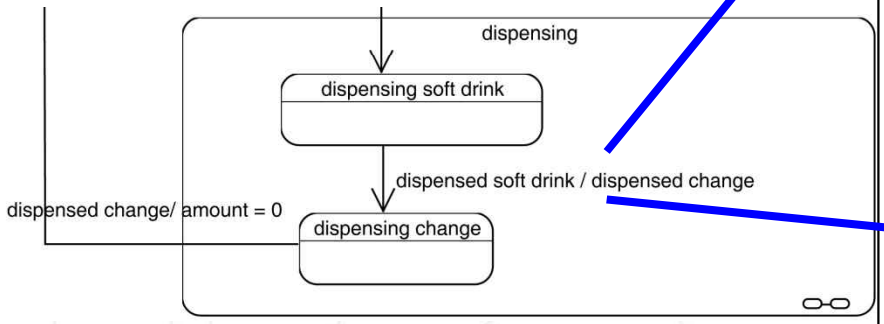
- 서브 상태는 별도의 클래스로 정의




```
class DispenseControl {
  int _state;
  static final int DispensingSoftDrink = 1;
  static final int DispensingChange = 2;
  static final int Complete = 3;
  public dispenseControl()
  {
    _state = DispensingSoftDrink;
  }
```

서브 상태의 구현

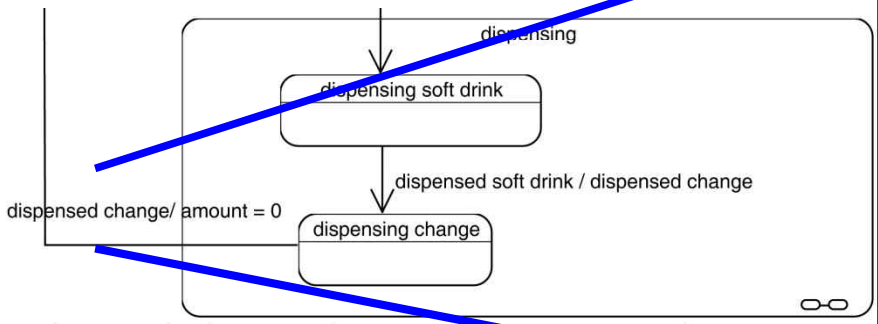
- 서브 상태 안의 상태변화는 서브 상태를 나타내는 클래스의 메소드로




```
public boolean dispensedSoftDrink()
    {
      if (_state == DispensingSoftDrink) {
        _state = DispensingChange;
        dispense change;
      }
      return false;
    }
```

서브 상태의 구현

- 서브 상태 안의 상태변화는 서브 상태를 나타내는 클래스의 메소드로




```
public boolean dispensedChange()
    {
      if (_state == DispensingChange) {
        _state = Complete;
        return true;
      }
      return false;
    }
```

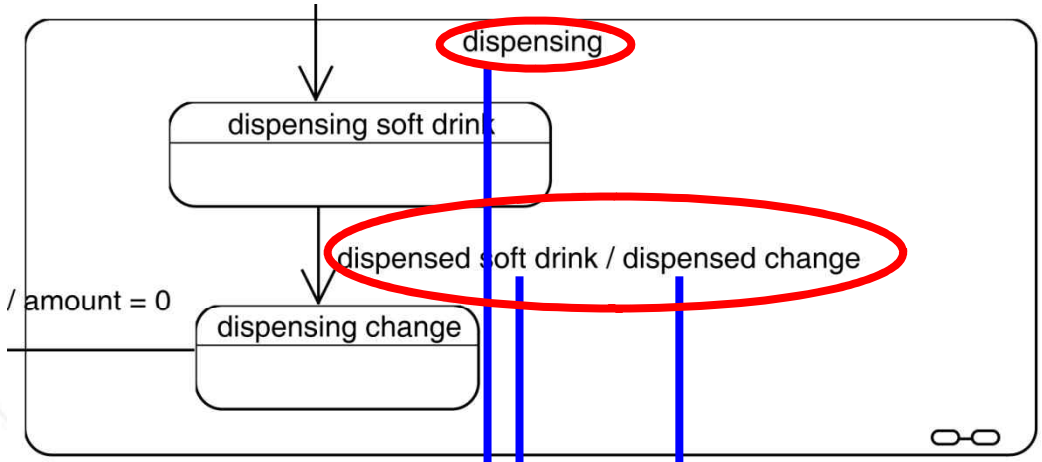
서브 상태의 초기화

- 서브 상태를 나타내는 클래스 정의
- 서브 상태 객체 생성 및 초기화

```
class VendingMachineControl
{
..declaration of state attribute, constants, other attributes;
  declaration of inner class dispenseControl;
..public VendingMachineControl(float price)
  {
    _amount = 0;
    _state = WaitingCoin;
    _price = price;
    _substate = new DispenseControl();
  }
```

서브 상태의 구현

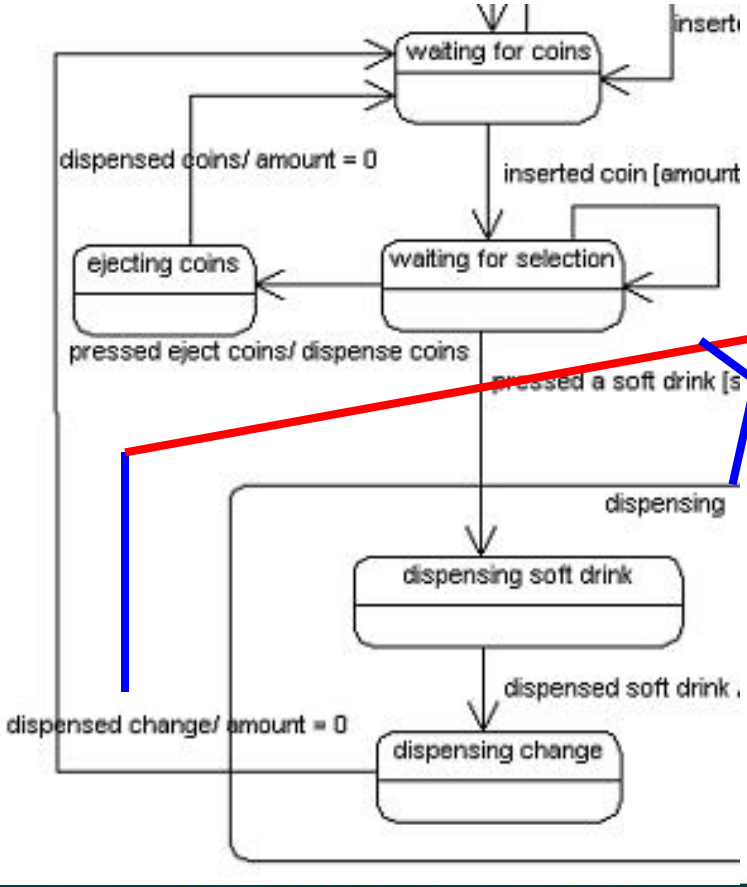
- 서브 상태 안의 상태 천이는 서브 상태는 나타내는 객체의 메소드 호출




```
public void dispensedSoftDrink() // VendingMachineControl
 {
   if (_state == Dispensing) {
     boolean isComplete = _substate.dispensedSoftDrink();
   }
 }
```

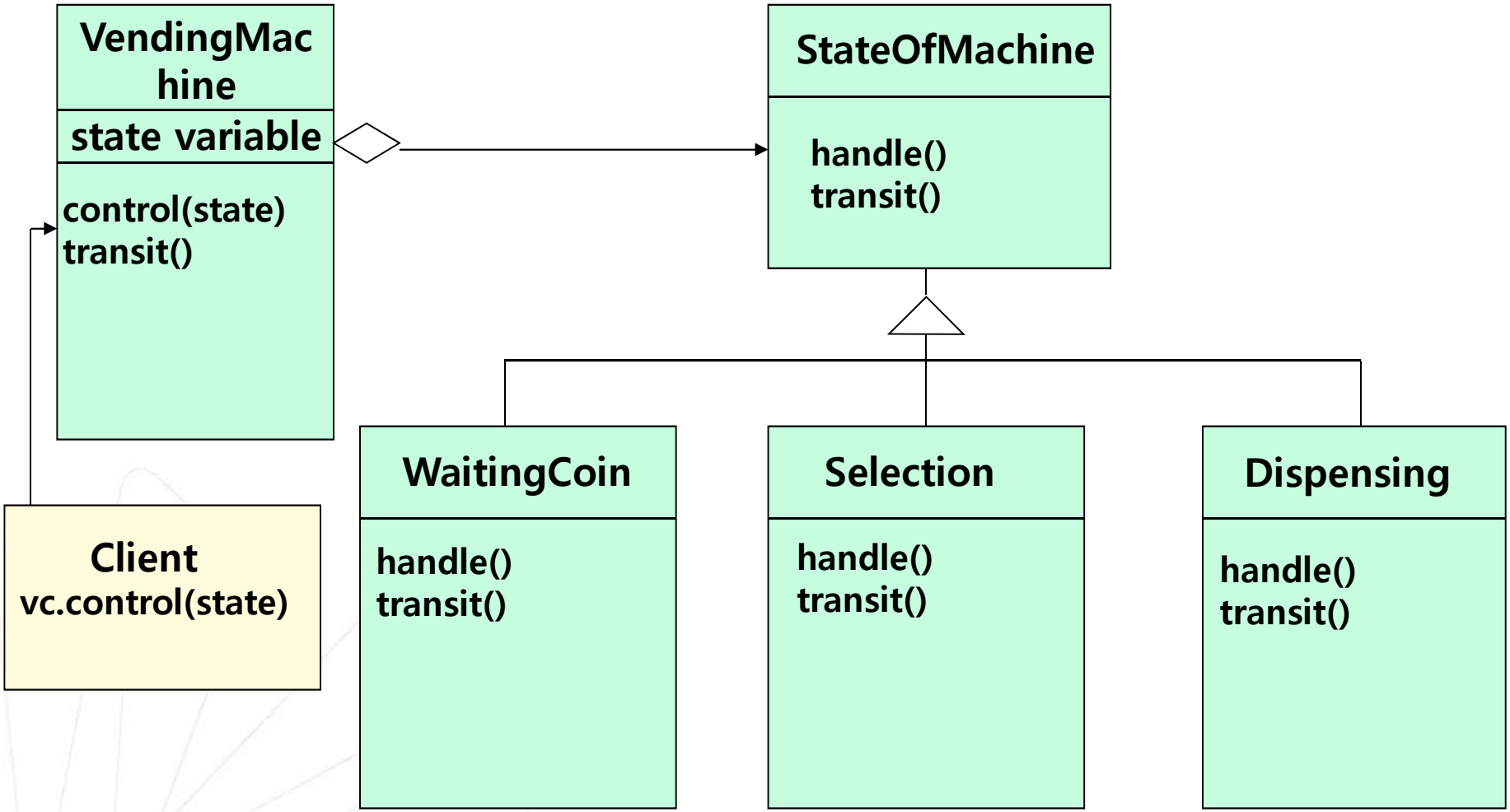
메인과 서브 상태의 연결

- 서브 상태의 종료조건 체크 후 메인 상태로 전환




```
// VendingMachineControl
public boolean dispensedChange()
 {
   if (_state == Dispensing) {
     boolean isComplete =
_substate.dispensedChange();
     if (isComplete) {
       amount = 0;
       _state = WaitingCoin;
     }
   }
 }
```

상태 패턴의 적용

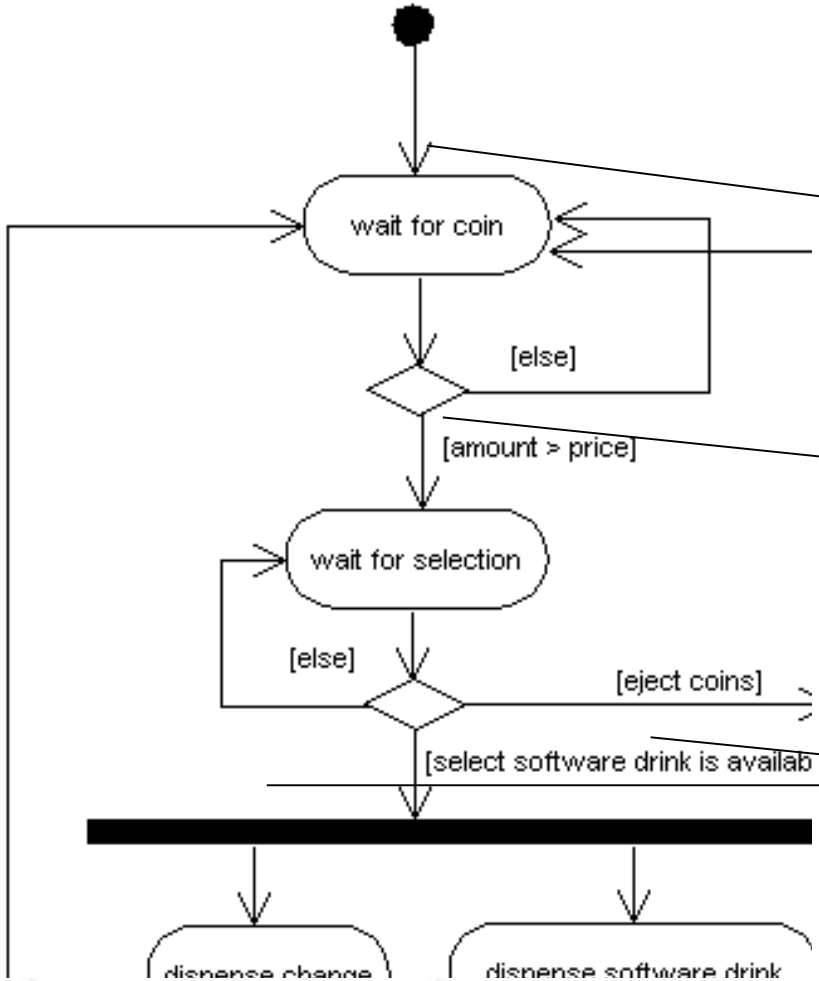

VendingMac hine
state variable
control(state)
transit()
StateOfMachine
handle()
transit()
Client
vc.control(state)
WaitingCoin
handle()
transit()
Selection
handle()
transit()
Dispensing
handle()
transit()

액티비티 다이어그램의 구현

- 액티비티나 프로시저 안에서 수행되어야 할 액션들의 순서를 나타냄
  - 제어 객체나 서브시스템의 알고리즘이나 제어 흐름을 나타냄
  - 프로그램의 위치, 즉 프로그램의 제어문, 반복문 으로 액티비티 다이어그램의 제어 흐름을 구현

- 액티비티 다이어그램을 코딩 하는 일반적인 규칙
  - 액션 상태는 메소드 호출이나 일반 계산문장으로 구현
  - 제어 노드는 if-then-else 문장으로 구현
  - 병렬 노드는 스레드로 구현
  - 반복 구조는 while 루프로 구현

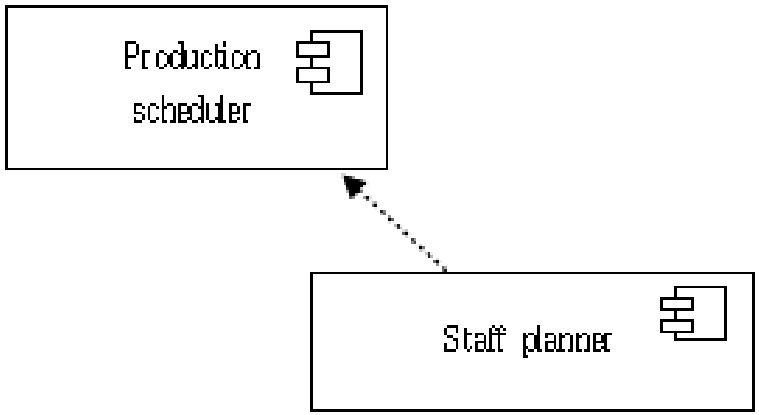
액티비티 다이어그램의 구현



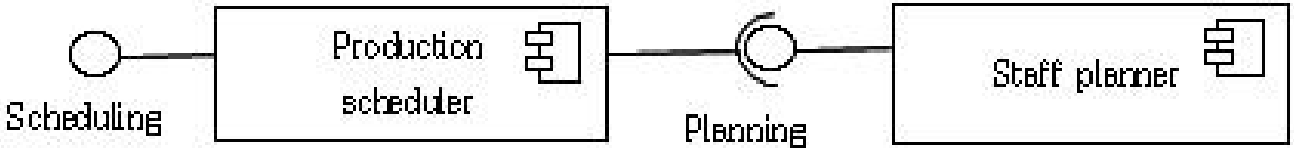

```
while (true) {
  amount = 0.0;
  while (amount < price) {
    wait for a coin;
    add coin value to amount;
  }
  show all available soft drink;
  while (selection is not done) {
    wait for selection from user;
    if selection is “eject coins” {
      dispense coins;
      set selection to “done”;
    }
    else if selection is a valid soft drink {
      dispense change & soft drink concurrently;
      set selection to “done”
    }
  }
}
```

구현 단계의 UML 다이어그램

- 컴포넌트 다이어그램
  - 원시코드의 단위가 되는 실행 컴포넌트 사이의 관계

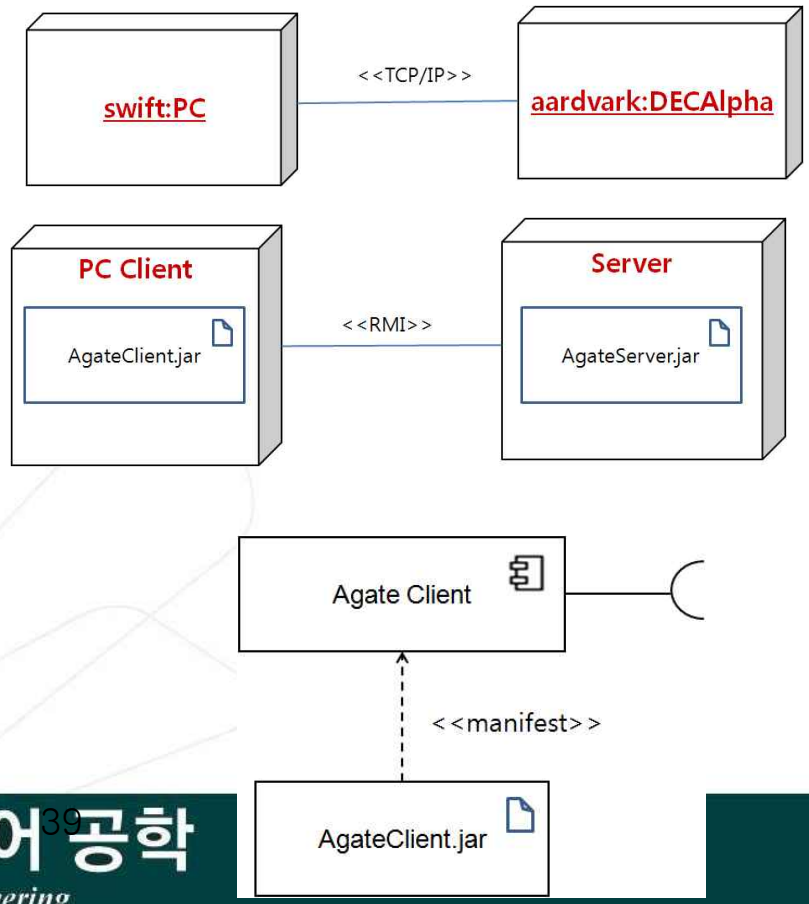


  - 컴포넌트의 인터페이스



배치 다이어그램

- 배치 다이어그램
  - 런 타임 처리 요소의 형상과 소프트웨어 컴포넌트, 결과물, 프로세스가 어디에 위치하는지를 나타냄
  - 노드와 커뮤니케이션 경로를 나타냄



Questions?