7. Coding

Choi, Kwanghoon

Dept. of Software Engineering
Chonnam National University

Table of Contents

7.1 Principles of Coding

- Coding Process
- Understanding Errors in Coding
- Structured Programming
- Information Hiding
- Duplicate Avoidance
- Principles of Least Knowledge

Errors in Coding

- Memory leak
- Matching malloc/free
- NULL reference
- Alias
- Array index out of bounds
- Arithmetic Exceptions
- Off-by-one error
- User-defined (enumeration) type
- String manipulation errors
- Buffer overflow error
- Synchronization errors

7.2 Coding Style

- Naming rules
- Reference types > pointer types (in C++)
- Use more specific type, and avoid too general type such as Object
- Use { } and ( ) to make it explicit
- == vs. equals (over strings)
  - name.equals(“bob”) vs. “bob”.equals(name)
- Avoid errors by type constraints
- Conversion of malformed data into well-formed ones for input
- Documentation style

7.3 UML and Coding

- How to implement UML design
  - Class diagram (Association),
  - Sequence diagram, and
  - State diagram

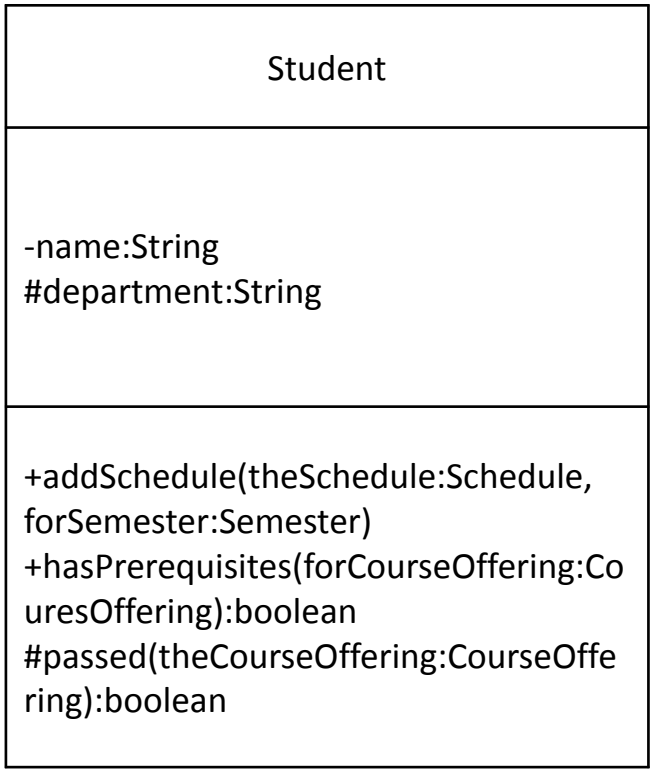
- cf. Two software development methodologies

[Ch.5 Modeling]

Software Development Methodology	Types of Elements in the Modelling
Structured Methods	Function
Object-Oriented Methods	Class/Object

Implementation: Class Diagram

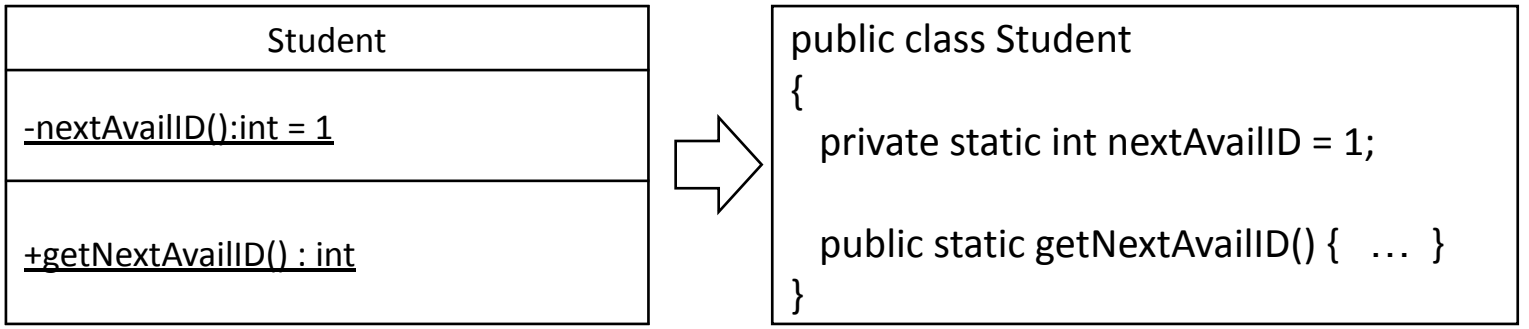
- Attribute & Operation
  - + => public, - => private, # => protected




```
public class Student
{
  private String name;
  protected String department;

  public void addSchedule(Schedule theSchedule,
                          Semester forSemester) {    …    }

  public boolean has Prerequisites(
       CourseOffering forCourseOffering)   {    …    }

  protected boolean  passed(
      CourseOffering theCourseOffering) {   …   }
}
```

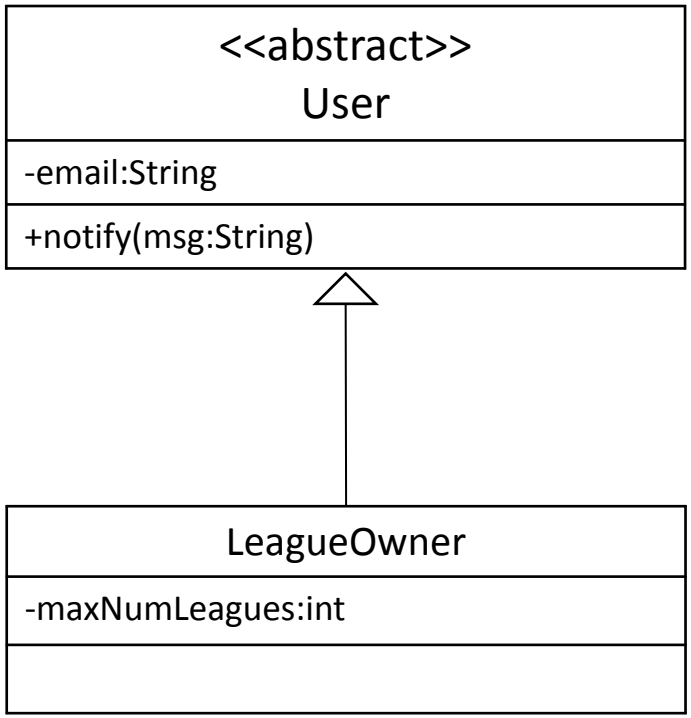
Implementation: Class Diagram

- Underlined attributes and operations => static



Implementation: Class Diagram

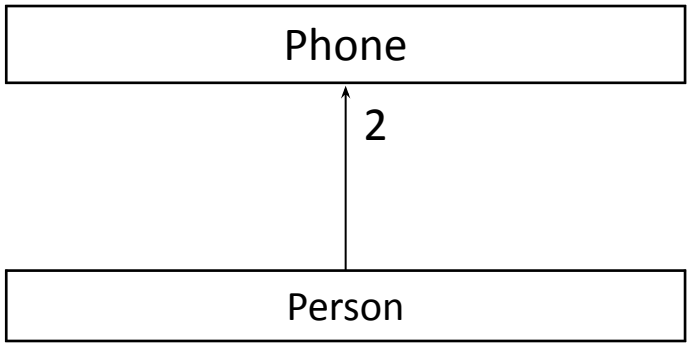
- Inheritance



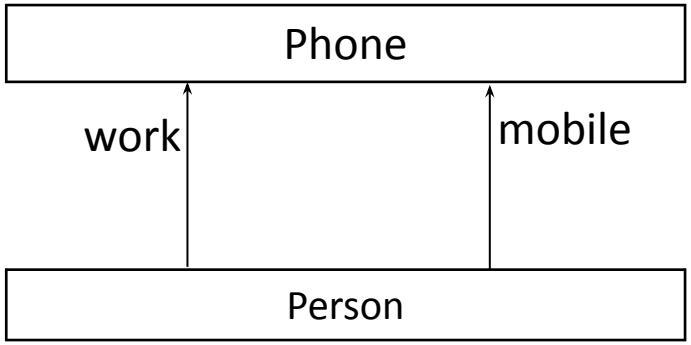

```
public abstract class User
{
   private String email;
   public void notify(String msg);
}

public class LeagueOwner extends User
{
   private int maxNumLeagues;
   …
}
```

Implementation: Class Diagram

- Association
  - One-to-one, one-to-many, many-to-many



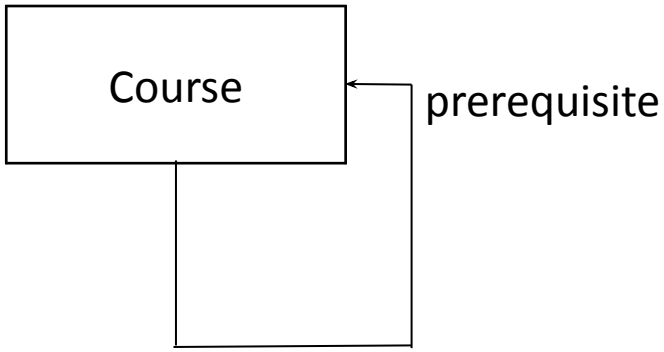

```
public class Person
{
  private Phone phone1, phone2;
  …
}
```




```
public class Person
{
  private Phone work_phone;
  private Phone mobile_phone;
  …
}
```

Implementation: Class Diagram

- Association
  - One-to-one, one-to-many, many-to-many

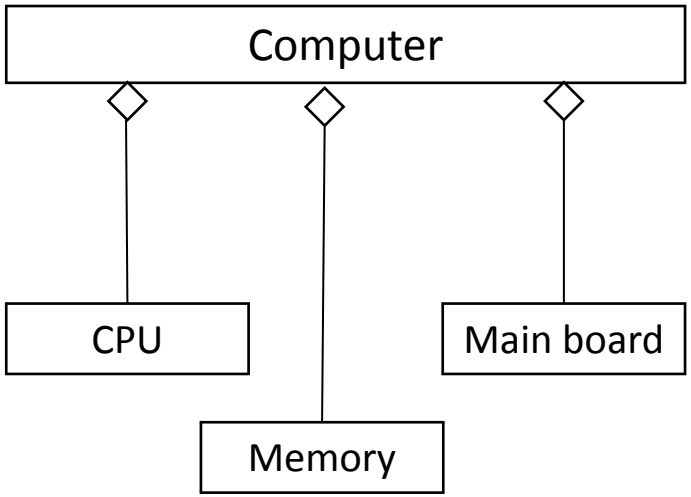



```
public class Course
{
   private Course prerequisite_course;
   …
}
```

Implementation: Class Diagram

- Aggregation and composition

[Aggregation (포함관계)]

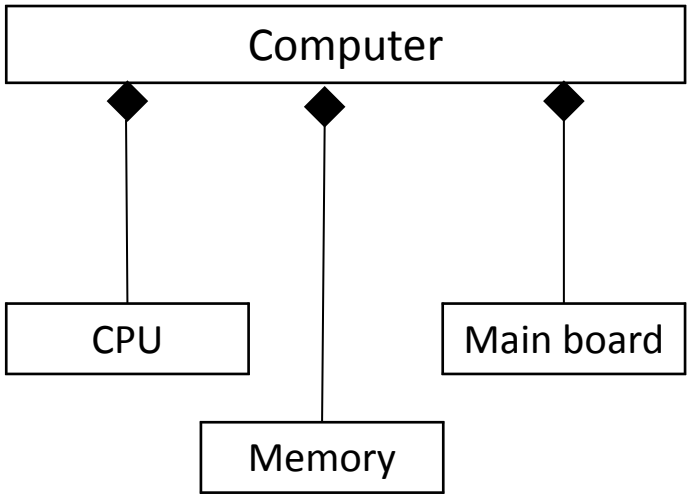



```
public class Computer
{
  private Cpu aCpu;
  private Memory aMem;
  private MainBoard aMBd

  public void Computer(Cpu aCpu,
      Memory aMem, MainBoard aMbd) {
      this.aCpu = aCpu;
      this.aMem =aMem;
      this.aMbd = aMbd;
  }
}
```

Implementation: Class Diagram

- Aggregation and composition

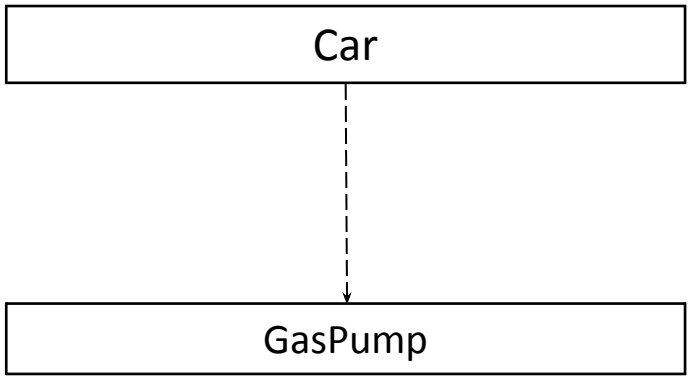
[Composition (합성 관계)]




```
public class Computer
{
  private Cpu aCpu;
  private Memory aMem;
  private MainBoard aMBd

  public void Computer() {
      aCpu = new Cpu();
      aMem = new Memory();
      aMbd = new MainBoard();
  }
}
```

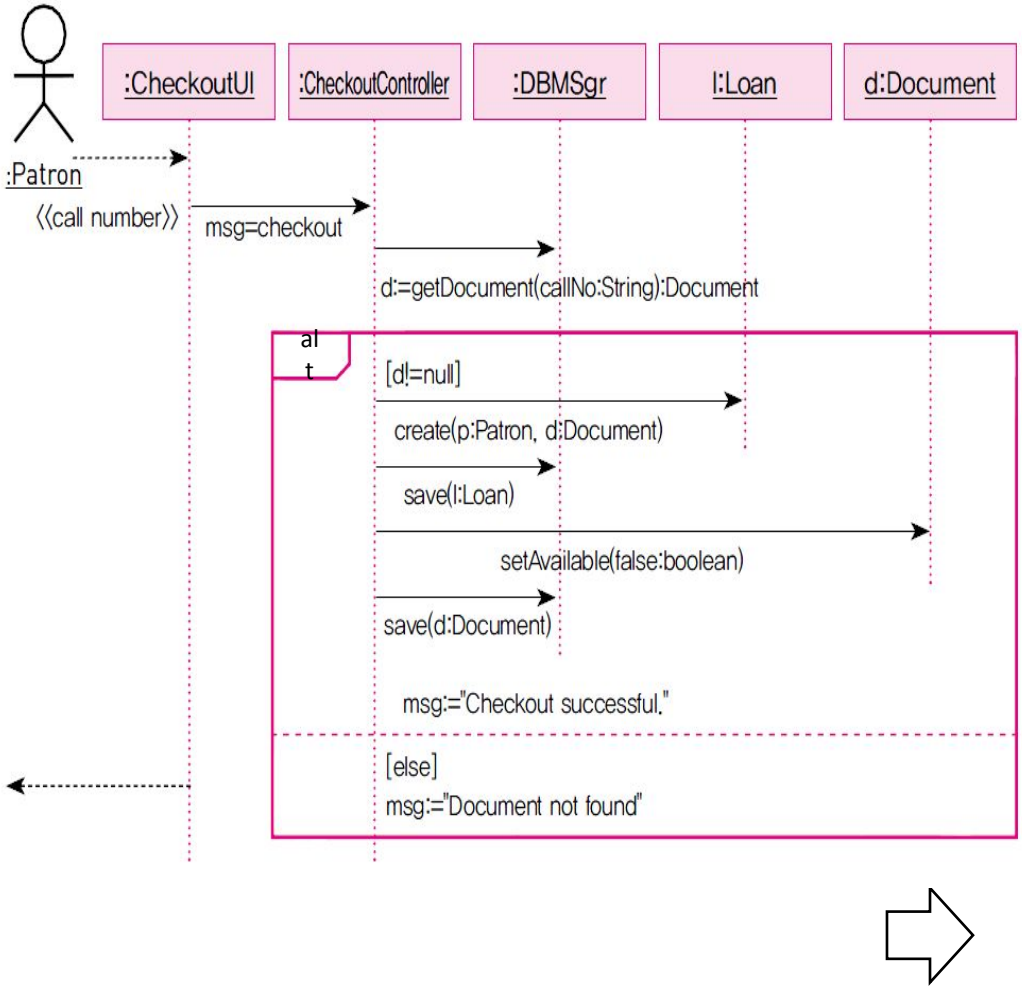
Implementation: Class Diagram

- Dependence




```
public class Car {
  …
  public void fillGas(GasPump p) {
     p.getGas(amount);
  }
}
```

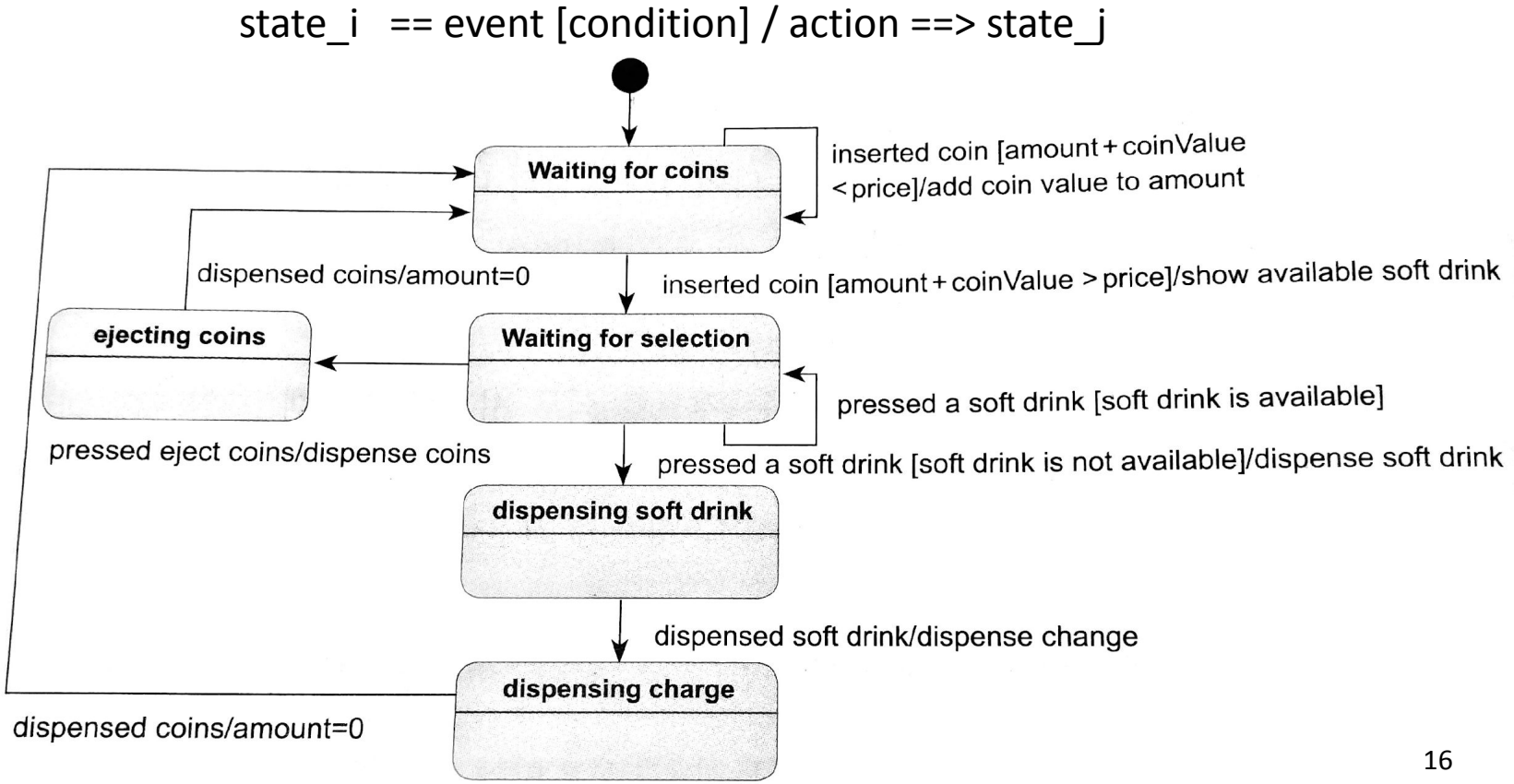
Implementation: Sequence Diagram




```
public class CheckoutController {
   Patron p;

   public String checkout(String callNo) {
    DMBSgr dbm = new DMBSgr();
    Document d = dbm.getDocument(callNo);
    String msg ;
   if (d != null) {
      Loan l = new Loan(p, d);
      dbm.save(l);
      d.setAvailable(false);
      dbm.save(d);
      msg = “Checkout successful.”;
   }
   else {
      msg = “Document not found.”;
   }
   return msg;
  }
}
```

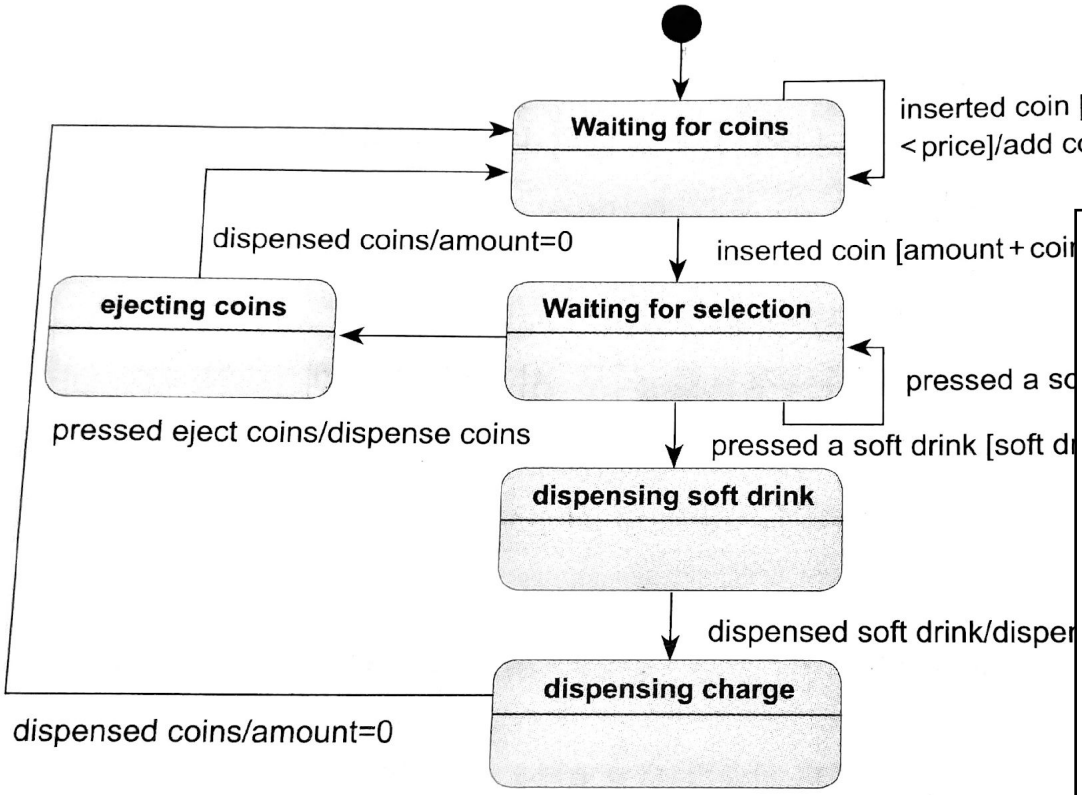
Implementation: State Diagram

- States => constants, Events => methods, Conditions => if statements in the methods, Actions => statements in the methods



Implementation: State Diagram

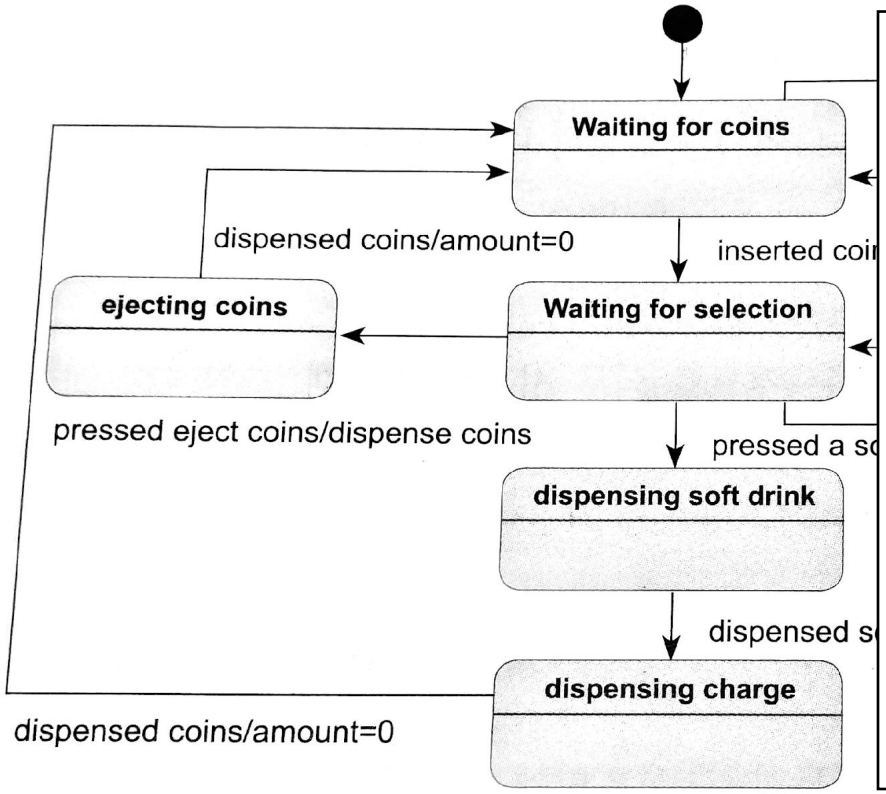
- Five states and two variables are used in the conditions




```
public class VendingMachineControl {
  private int state;

  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;

  private float amount;
  private float price;
  …
}
```

Implementation: State Diagram

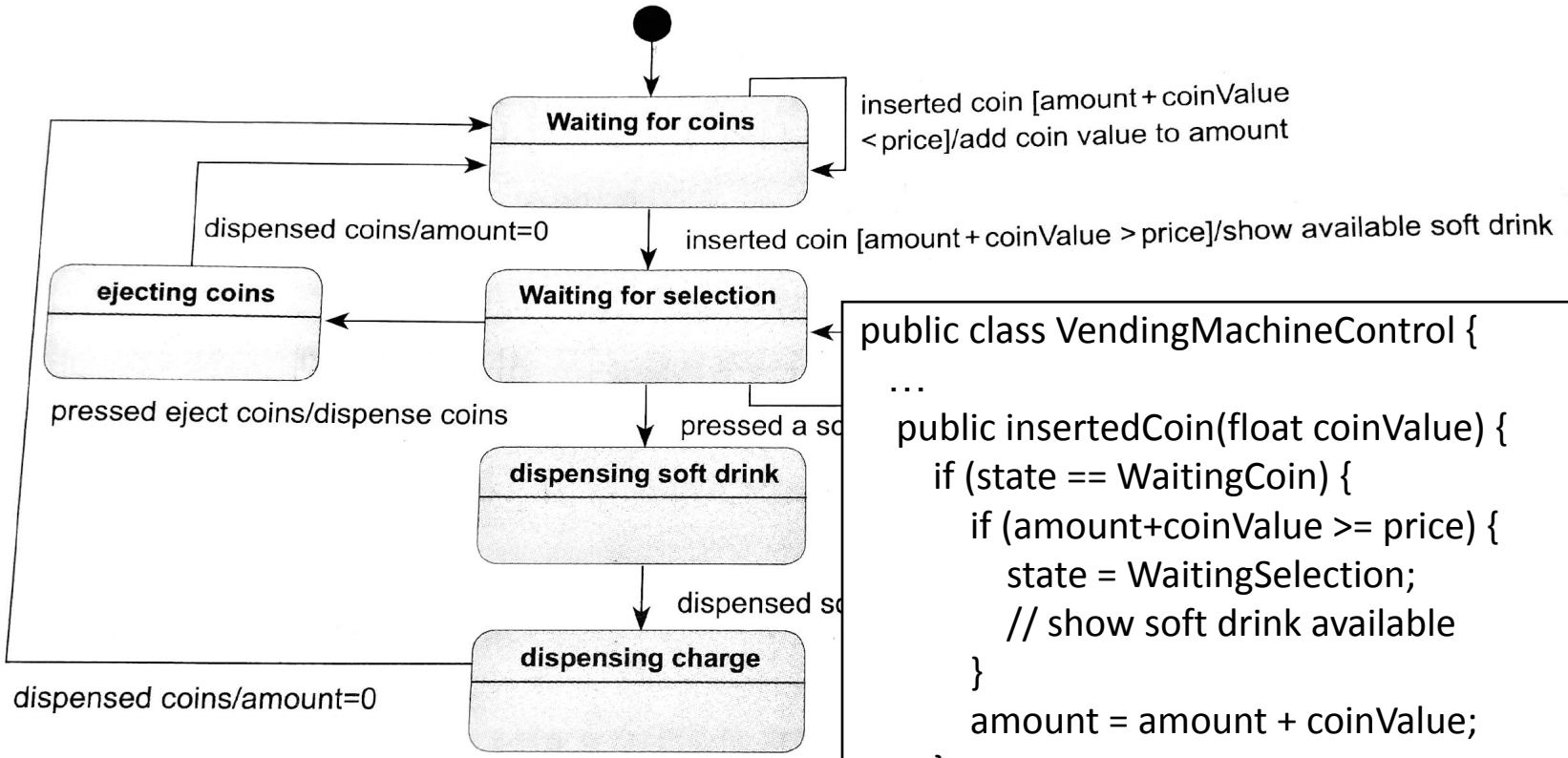
- Initialization




```
public class VendingMachineControl {
  …
   public VendingMachineControl(float price) {
      this.amount = 0;
      this.state = WaitingCoin;
      this.price = price;
    }
  …
}
```

Implementation: State Diagram

- Events => methods



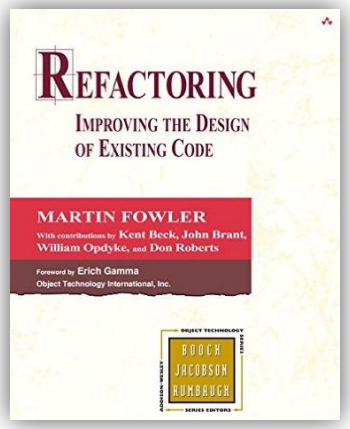

```
public class VendingMachineControl {
  …
   public insertedCoin(float coinValue) {
      if (state == WaitingCoin) {
         if (amount+coinValue >= price) {
            state = WaitingSelection;
            // show soft drink available
         }
         amount = amount + coinValue;
      }
   }
  …
}
```

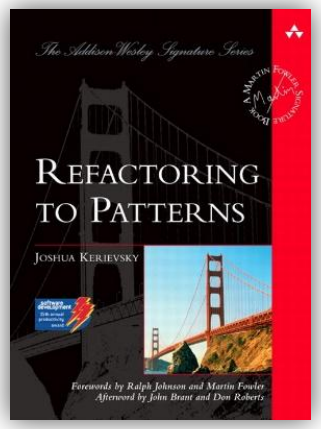
7.4 Refactoring

- Behavior-preserving transformation
  - Removal of duplication, the simplification of complex logic, and the clarification of unclear code
- Why?
  - Make it easier to add new code (requirements change)
  - Improve the design of existing code
  - Gain a better understanding of code
- Increasingly popular due to agile development

7.4 Refactoring

- How can we “guarantee” behavior-preserving?
  - Test the code before and after refactoring to see if the results are the same
- Fowler’s book
  - Catalog of refactorings
  - List of code smells
  - Guidelines on when applying refactoring
  - Example of code before and after





7.4 Refactoring

- Catalogue

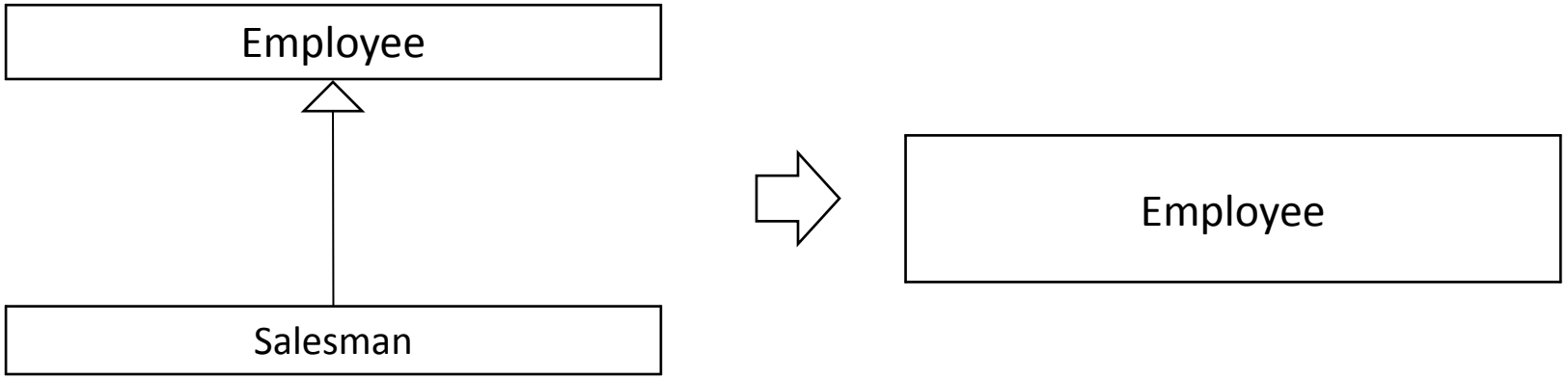
- Add parameter
- Change association
- Reference to value
- Value to reference
- Collapse hierarchy
- Consolidate conditionals
- Procedures to objects
- Decompose conditionals
- Encapsulate collection

- Encapsulate downcast
- Encapsulate field
- Extract method
- Extract class
- Inline class
- Form template method
- Hide delegate
- Hide method
- Inline temp

참고) Udacity

Refactoring: Collapse hierarchy

- A superclass and a subclass are too similar
  => merge them

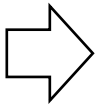


참고) Udacity

Refactoring: Consolidate conditional expr.

- Set of conditionals with the same result
  => combine and extract them

```
Double disabilityAmount() {
   if (seniority < 2 )
       return 0;
   if (monthsDisabled > 12)
       return 0;
   if (isPartTime)
       return 0;
   // compute disability amount
}
```



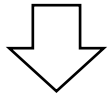
```
Double disabilityAmount() {
   if ( notEligibleForDisability() )
       return 0;
   // compute disability amount
}
```

참고) Udacity

Refactoring: Decompose conditionals

- A conditional statement is particularly complex
  => extract methods from conditions modify THEN and ELSE part of the conditional

```
If ( date.before(SUMMER_START) || date.after(SUMMER_END))
    charge = quantity * winterRate + winterServiceCharge;
Else
    charge = quantity * summerRate;
```



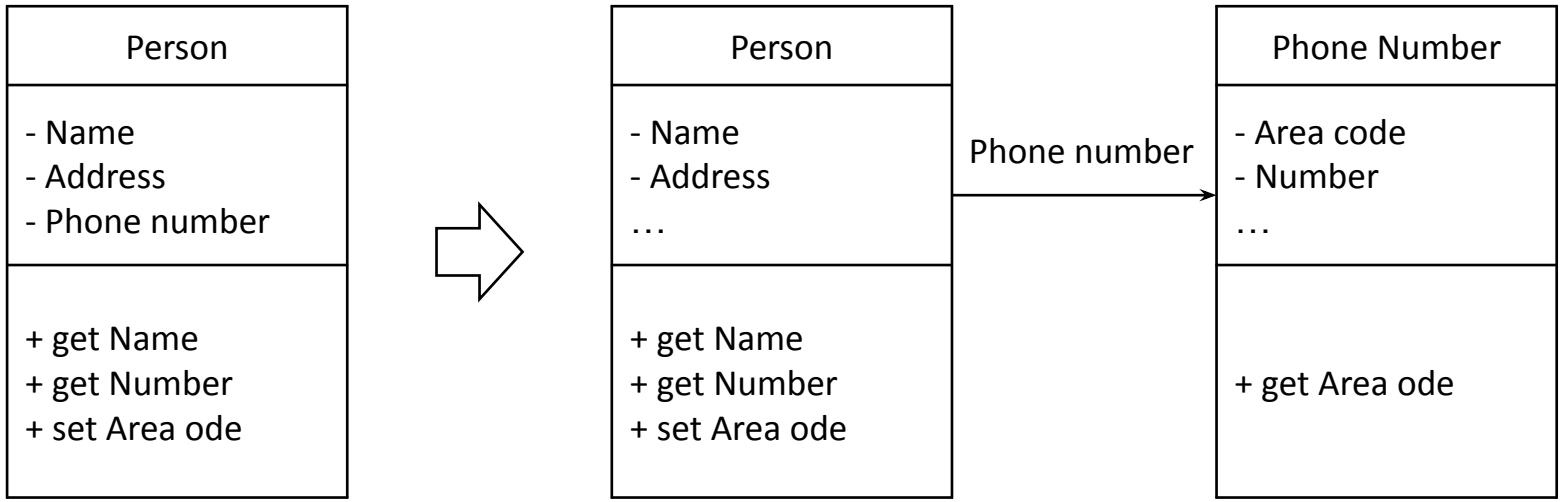
```
If ( not_Summer(date) )
   charge = winterCharge ( quantity );
else
    charge = summerCharge( quantity );
```

참고) Udacity

Refactoring:Extract class

- A class is doing the work of two classes

  => create a new class and move there relevant fields/methods

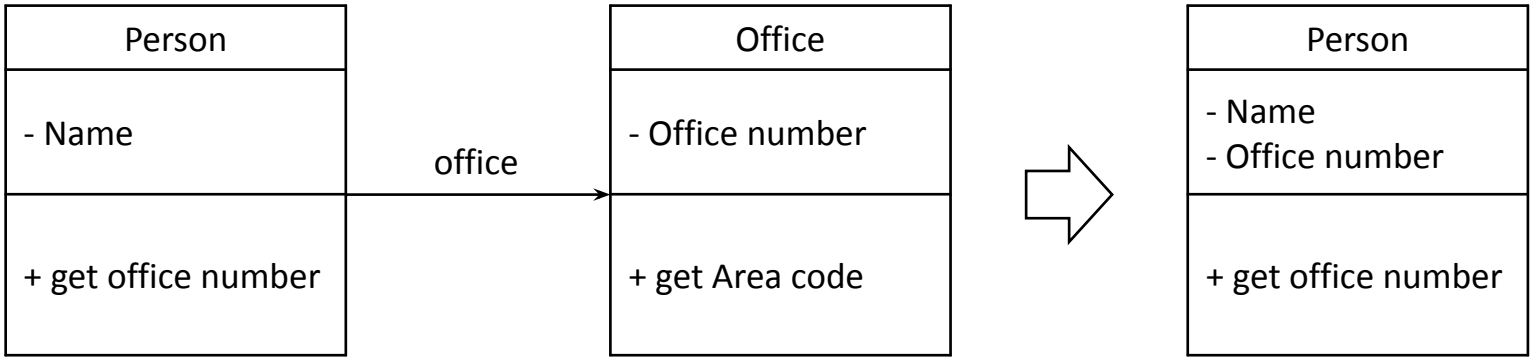


참고) Udacity

Refactoring: Inline class

- A class is not doing much

  => move its features into another class and delete this one

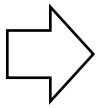


참고) Udacity

Refactoring: Extract method

- One of the most common refactoring methods
- Cohesive code fragment in a large method
  => create a method using that code fragment

```
void printOwing() {
 …
 System.out.println(
   “name” + name + “address” + address);
 …
 System.out.println(
   “amount owned” + amount);
 …
}
```



```
void printOwing() {
 …
 printDetails();
 …
}

void printDetails() {
 System.out.println(
   “name” + name + “address” + address);
 …
 System.out.println(
   “amount owned” + amount);
}
```

참고) Udacity

Standard Refactoring in Eclipse

- Rename
- Move
- Change Method Signature
- Extract Method
- Extract Local Variable
- Extract Constant
- Inline
- Introduce Parameter

http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Freference%2Fref-menu-refactor.htm

Refactoring: Extract method using Eclipse

```
public class Invoice {
  private MyCollection _orders = new MyCollection();
  private String _name = “Customer Name”;

  void printOwing() {
    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    // print banner
    System.out.println(“**************************”);
    System.out.println(“*****  Customer Owes  *****”);
    System.out.println(“**************************”);

    // calculate outstanding
    while (e.hasMoreElements()) {
      Order each = (Order) e.nextElement();
      outstanding += each.getAmount();
    }

    // print details
    System.out.println(“name: ” + _name);
    System.out.println(“amount: ” + outstanding);
}
```

Q. Try to refactor the example using Extract Method in Eclipse so that we can have three new methods:

- printBanner
- calculateOutstanding
- printDetails

Q. Discuss how the three extracted methods are different.

참고) Udacity

```
Public class AnotherInvoice {
  private MyCollection _orders = new MyCollection();
  private String _name = “Customer Name”;

  void printOwing() {
    Enumeration e = _orders.elements();
    double outstanding = 0.0;
    int count = 0;

    // print banner
    System.out.println(“**************************”);
    System.out.println(“*****  Customer Owes  *****”);
    System.out.println(“**************************”);

    // calculate outstanding
    while (e.hasMoreElements()) {
      Order each = (Order) e.nextElement();
      outstanding += each.getAmount();
       count += 1;
    }

    // print details
    System.out.println(“name: ” + _name);
    System.out.println(“amount: ” + outstanding);
    System.out.println(“count: ” + count);
}
```

We have slightly changed the example by introducing a new variable count.

Q. Refactor it again and discuss what problem happens and what is your solution?

- Hint: calculateOutstanding

참고) Udacity

Refactoring Risks

- Powerful tool, but …
  - May introduce subtle faults
  - Should not be abused
  - Should be used carefully on systems in production

Cost of Refactoring

- Manual work
- Test development and maintenance
  - Should check ‘behavior preservation’ before and after refactoring (typically by unit tests)
- Documentation maintenance
  - Should update the document as well accordingly

When not to refactor

- When code is broken
- When a deadline is close
- When there is no reason to do it

Code Smell and Refactoring

- Code smell is an indication that there might be something wrong with the code

Code smell	Description	Refactoring
Duplicated code (중복된 코드)	기능이나 데이터 코드가 중복된다	Extract method
Long method(긴 메소드)	메소드의 내부가 너무 길다	Extract method, Decompose conditional, …
Large class(큰 클래스)	한 클래스에 너무 많은 속성과 메소드가 존재한다	Extract class (or subclass)
Long parameter list(너무 많은 파라미터 리스트)	메소드의 파라미터 개수가 너무 많다	파라미터의 개수를 줄인다
Divergent Class (두 가지 이상의 이유로 수정되는 클래스)	한 클래스의 메소드가 2가지 이상의 이유로 수정되면, 그 클래스는 한 가지 종류의 책임만을 수행하는 것이 아니다	Extract class (or subclass) (한 가지 종류만을 담당하도록 수정)

참고) 최은만 교재

Code Smell and Refactoring

-

Code smell	Description	Refactoring
Shotgun Surgery (여러 클래스를 동시에 수정)	특정 클래스를 수정하면 그때마다 관련된 여러 클래스들 내에서 자잘한 변경을 해야 한다	Move method/field, Inline class, …
Feature Envy (다른 클래스를 지나치게 애용)	빈번히 다른 클래스로부터 데이터를 얻어 와서 기능을 수행한다	Extract method, Move method
Data Clumps (유사 데이터들의 그룹 중복)	3개 이상의 데이터 항목이 여러 곳에 중복되어 나타난다	Extract class (해당 데이터들은 독립된 클래스로 정의)
Primitive Obsession(기본 데이터 타입에만 집착)	객체 형태 그룹을 만들지 않고, 기본 데이터 타입만 사용한다	같은 작업을 수행하는 기본 데이터의 그룹을 별도의 클래스로 만든다

참고) 최은만 교재

Code Smell and Refactoring

•

Code smell	Description	Refactoring
Switch, If statements	switch 문장이 지나치게 많은 case를 포함한다	다형성으로 바꾼다(같은 메소드를 가진 여러 개의 클래스를 구현한다)
Parallel Inheritance Hierarchies (병렬 상속 계층도)	[Shotgun Surgery]의 특별한 형태로서, 비슷한 클래스 계층도가 지나치게 많이 생겨 중복을 유발한다	호출하는 쪽의 계층도는 그대로 유지하고 호출당하는 쪽을 변경한다

참고) 최은만 교재

https://refactoring.guru/refactoring/smells

7.5 Code Quality Improvement

- Code Inspection https://en.wikipedia.org/wiki/Fagan_inspection
  - Manually reading and verifying source programs
  - Tools: PMD (*구문-syntax* 기반 코드 패턴 점검), SonarQube(sonarqube.org)

- Static analysis
  - Systematic methods to find defects from source programs
  - Tools: FindBugs, Infer http://fbinfer.com/
    (*의미-semantics 기반* 코드 점검)

- Dynamic analysis
  - testing, monitoring, debugging

7.5 Code Quality Improvement

- SW 품질(보안)을 자동 점검하는 소프트웨어
  - 세상의 SW 규모가 사람의 수동 검사 규모를 능가
    => 프로그래밍언어론과 컴파일러 기술로
    SW 자동 점검 소프트웨어를 만들어야 함!

- 구문 분석 기반 점검
  - Lexical analysis (토큰 분석)
  - Syntax analysis (구문 분석, 파싱) :
    ex) PMD, SonarQube

7.5 Code Quality Improvement

- 의미 분석 기반 점검

  - Static analysis (정적 분석)
    - ex) FBInfer https://fbinfer.com/ (Java/Android/C/C++/Obj-C 프로그램)

  - Type checking (타입 검사)
    - ex) TypeScript (자바스크립트 JavaScript의 타입 체킹)
    - ex) Pyright https://github.com/microsoft/pyright (파이썬 타입 체킹)

  - Symbolic execution (기호 실행)
    - ex) VeriSmart https://github.com/kupl/VeriSmart-public
      (이더리움 블록체인 토큰 스마트계약 프로그램)

  - Fuzzing (퍼징) :
    - ex) 퍼징 A-Z https://www.fuzzingbook.org (파이썬 기반 실습)