Objects and Classes

Part 10 – Object-Oriented Design: Case Study

Chapter 4, Core Java, Volume I

Contents

- Object-Oriented Analysis and Design
- Example: Card Game (A Simplified Version)
- Design Hints

Object-Oriented Analysis and Design

- 객체지향 분석 (Object-Oriented Analysis)
  - 주어진 문제가 어떤 문제를 해결하고자 하는지 파악한다 (What)
  - 문제에 내재된 성질을 객체지향적인 방법으로 파악한다.
- 객체지향 설계 (Object-Oriented Design)
  - 주어진 문제에 대한 Software Solution을 찾는다 (How)
  - 문제를 객체지향적인 방법으로 해결하는 방안을 찾는다.

- OOAD Steps
  - Identifying Classes ➔ class diagram
  - Identifying Attributes ➔ class diagram
  - Identifyng Methods ➔ class diagram, activity diagram, sequence diagram, etc.
  - Designing Alogrithms (for some methods)

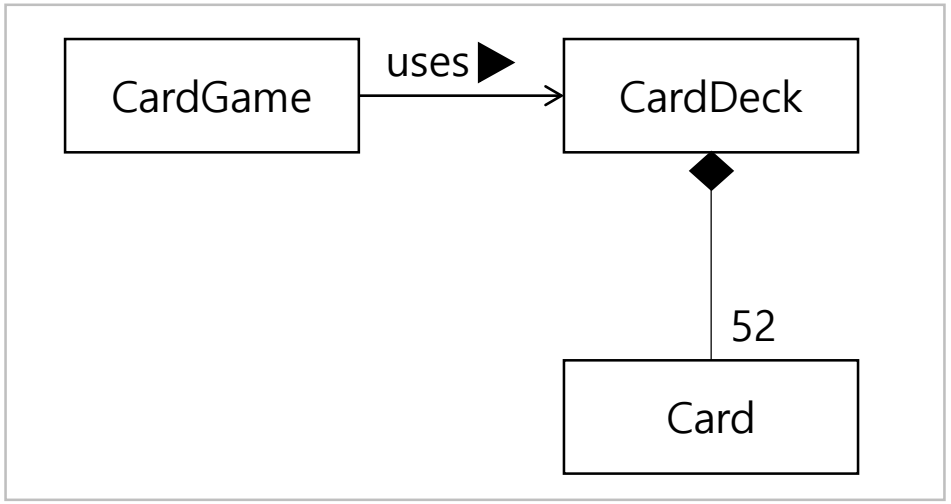
Developing a Card Game (Simplified Version)

- 문제 정의
  - Poker 카드 52장(Card Deck)을 가지고 카드 게임을 진행한다.
  - 게임을 시작하기 전에 카드를 작 섞어야 한다.
  - 한번에 한장의 카드를 뽑아 8보다 크면 이기고 8보다 작으면 진다. 8이면 비긴다.
  - 게임은 종료를 원하거나 카드가 더 이상 없을 때까지 계속해서 진행한다.
  - 게임이 종료되면 총 게임수와 이긴 횟수, 비긴 횟수, 진 횟수를 출력한다.

Identifying Classes

- 문제 정의나 요구사항 분석 문서에서 주요 클래스(물질적 개념, 추상적개념)를 찾는다.
- 클래스들의 관계(relationship)를 파악한다.
- 클래스들의 관계를 UML class diagram으로 표현한다.

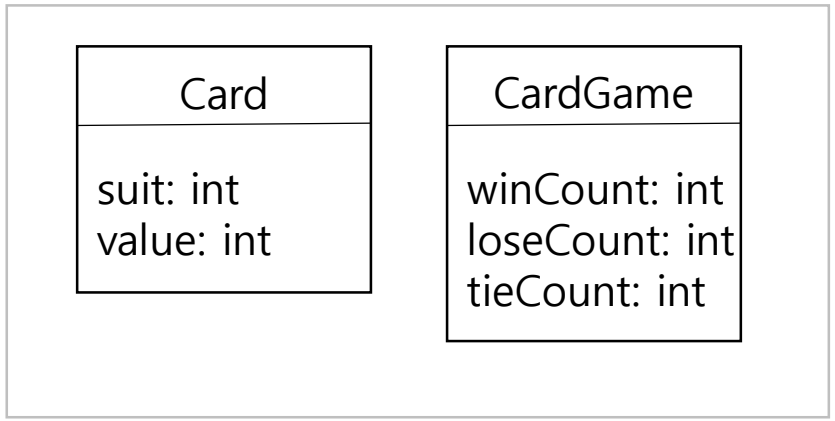
- 카드게임 분석
  - 주요 클래스
    - Card, CardDeck, CardGame
  - Class Diagram

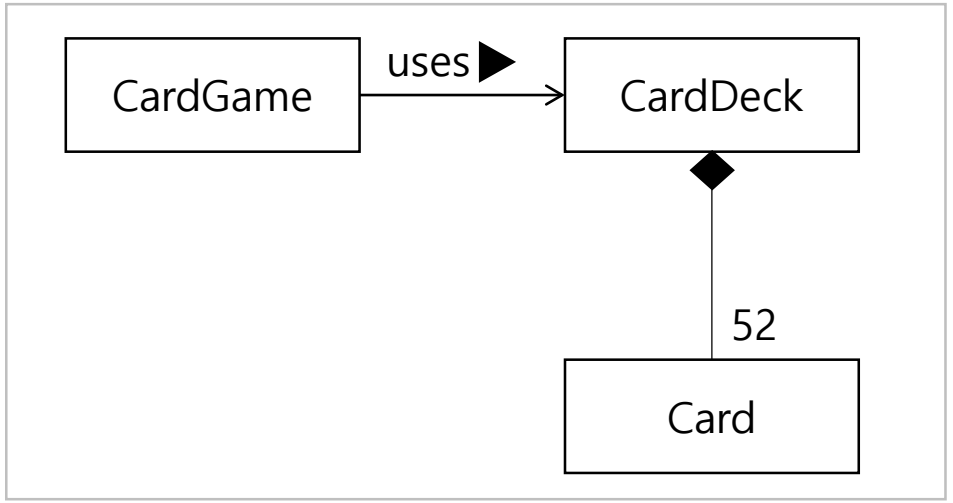


Identifying Attributes

- 전단계에서 찾은 클래스들의 주요 속성 정보를 찾는다.
- 속성 정보는 주로 primitive type이나 간단한 클래스 타입(String, Date 등)을 가진다.
- 속성 정보를 UML class diagram에 추가한다.

- 카드게임 분석
  - 주요 속성
    - Card : suit, value
    - CardGame: win count, lose count, tie count
  - Class Diagram

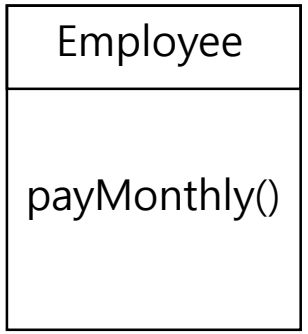




Identifying Methods

- 식별 방법
  - 프로그램의 시작클래스가 담당해야할 태스크(main method)를 위한 work flow를 파악한다
    - activity diagram(혹은 flow chart)으로 표현
  - Work flow에서 파악된 sub-task들을 누가 담당해야할지 결정한다(책임 분배).
  - .각 sub task 또한 앞 단계와 비슷한 방법으로 설계해나간다.

- 책임 분배 (assignment of responsibility)
  - 각 태스크를 어떤 클래스가 담당할 지 결정한다.
  - 태스크 수행에 필요한 정보를 가장 많이 가지고 있는 클래스가 담당
  - 예: Employee 문제에서 월급을 누가 계산하여 지불 하는게 좋은가?
    - Employee가 적당 (EmployeeTest, Account 클래스와 비교)
  - 클래스에 method 추가

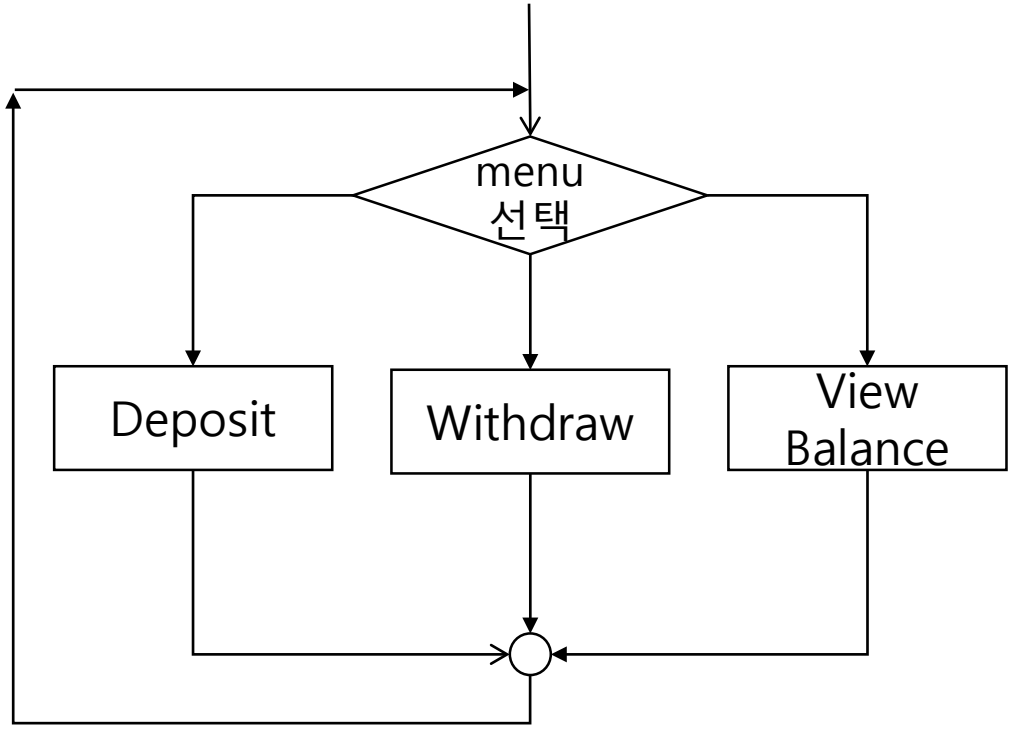


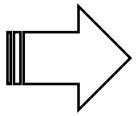
Identifying Methods

- Domain Class나 범용 자료구조 클래스
  - Work flow의 task를 판별하지 않더라도 일반적으로 전체 또는 일부 method 식별이 가능한 경우가 있음 (잘 알려진 behavior에 기반해서)
    - Stack : push, pop
    - Queue : enqueuer, dequeuer
    - Account : deposit, withdraw, get balance
    - Card : get value, get suit,
  - 일반화된 자료구조 클래스는 라이브러리로 제공
    - Java Collection Framework

Identifying Methods

- Banking Transactions

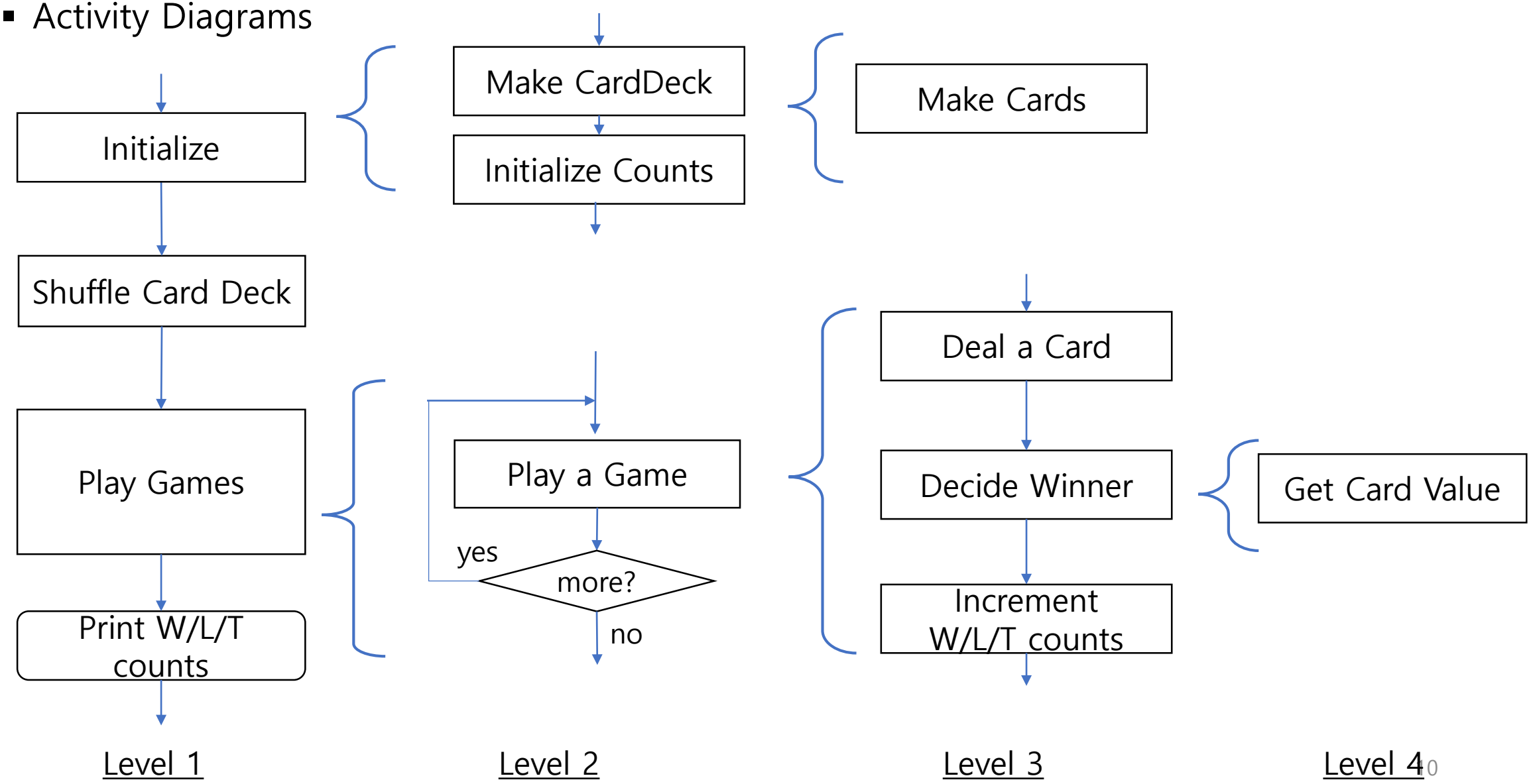




Account
balance : double
deposit(): void withdraw(): void getBalance(): double

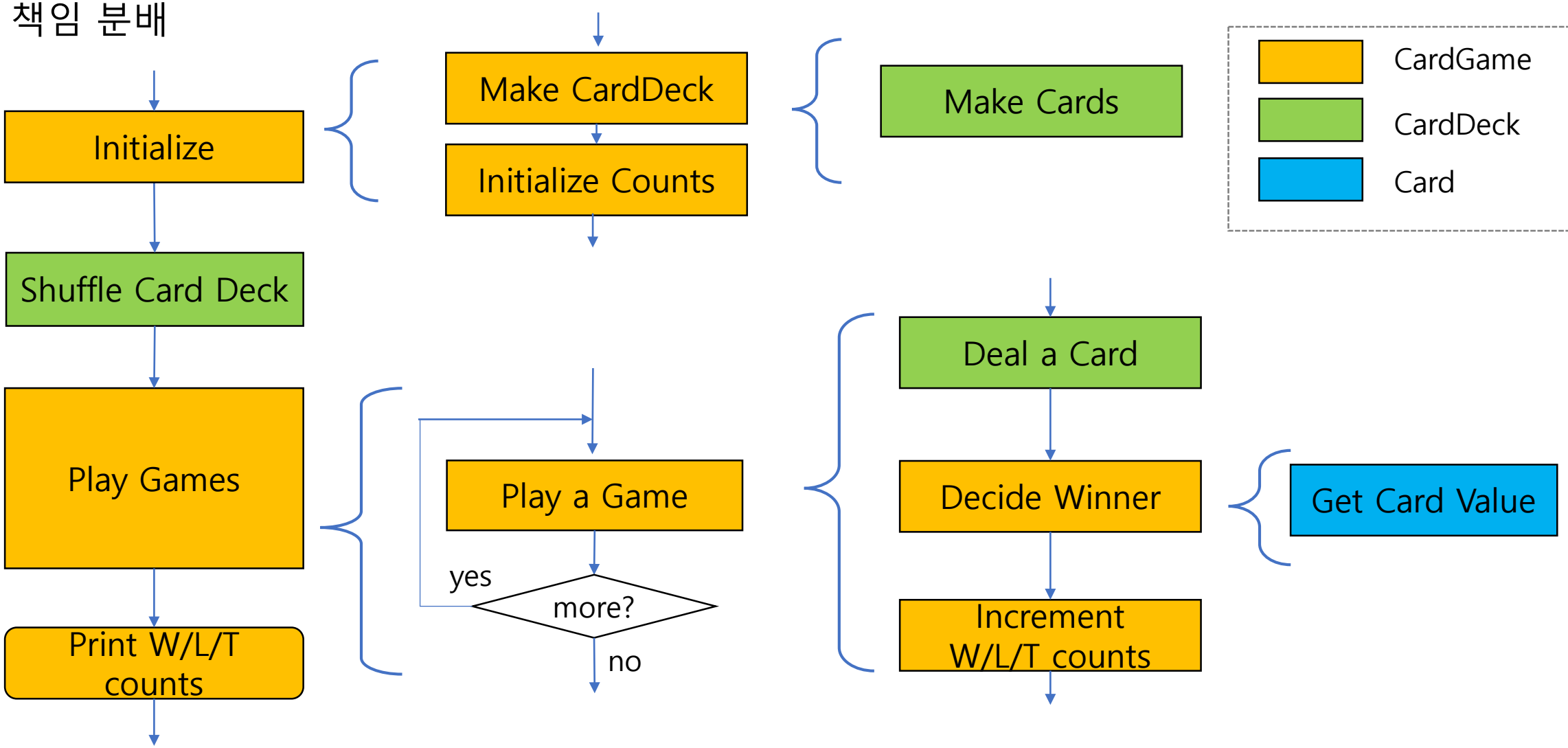
Identifying Methods

- Activity Diagrams



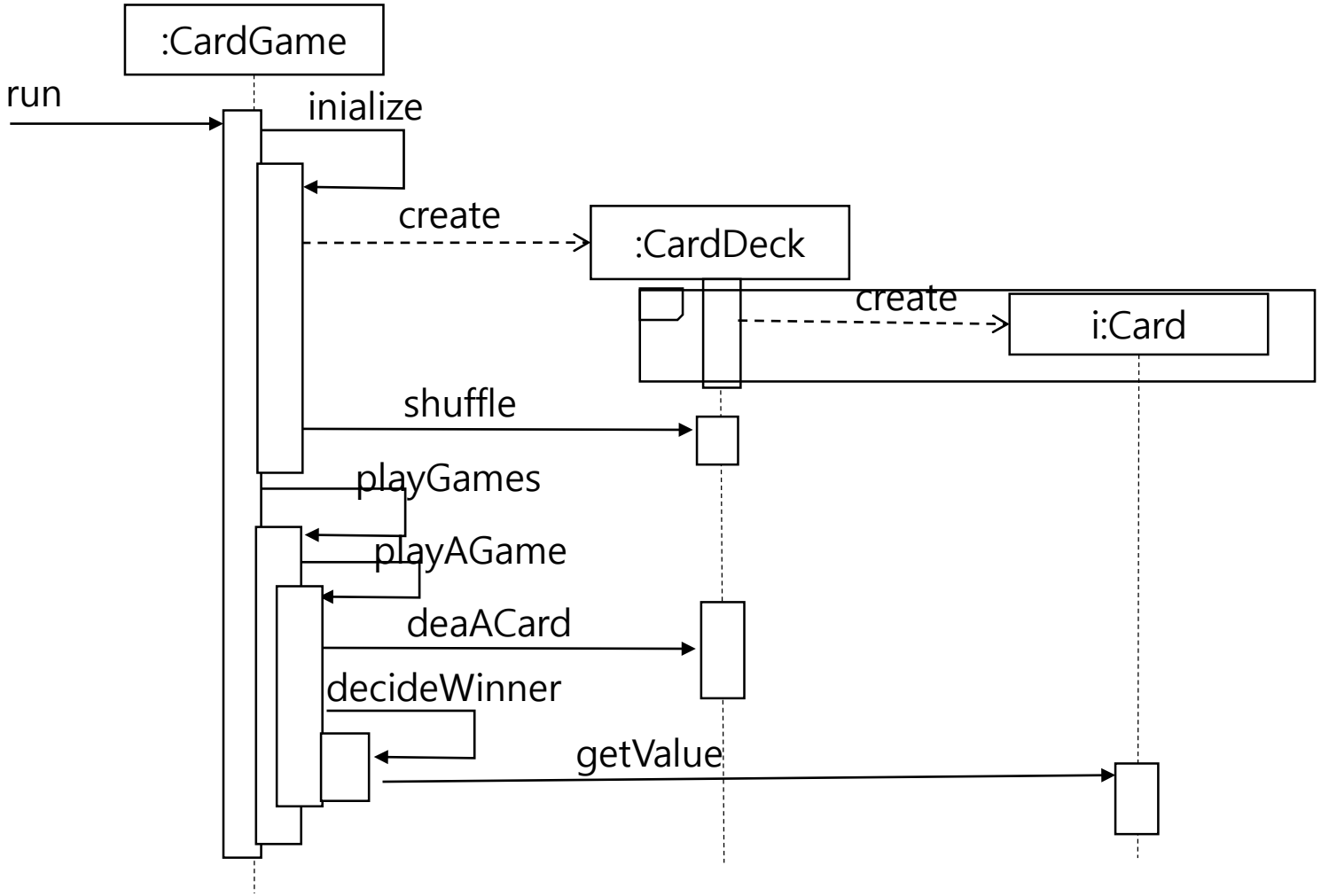
Identifying Methods

- 책임 분배



Identifying Methods

- Sequence Diagram
  - Describing collaboration of objects w.r.t. message sending



Identifying Methods

- (Design) Class Diagram

CardGame
winCount: int loseCount: int tieCount: int
initialize():void playGames():void playAGame():void decideWinner():int printCounts():void

CardDeck
currentIdx: int
shuffle():void dealACard():Card

Card
suit: int value: int
getValue() :int

Designing Algorithms

- 주요 메소드의 알고리즘 작성
  - Input 및 Ouput (return 값) 명시
  - Algorithm (pseudo code 또는 flow chart) 작성
  - 예외 조건 파악

- 카드게임의 주요 메소드 알고리즘
  - shuffle
  - dealACard
  - decideWinner
  - etc.

Method : dealACard
Input : currentIdx, deck[52]
Ouput: a Card or null
Algorithm:

```
Card c;
if(currentIdx < 52)
    c = deck[currentIdx];
 else
    c = null;
currendIdx++;
return c;
```

End.

Implementation

- 클래스의 구현
  - Class Diagram을 이용하여 Java class의 틀을 만든다
  - Class Diagram을 이용하여 method의 틀과 instatnce variable을 추가한다.
  - Class Diagram의 association, aggregation, composition등을 객체에 대한 reference로 추가한다.
  - Activity Diagram과 세부 알고리즘을 이용하여 method를 구현한다.
- 클래스 구현 순서
  - Dependency가 가장 작은 클래스부터 구현
  - 이미 구현된 클래스를 이용하여 구현하기 때문에 테스팅이 용이
- Card Game 클래스 구현
  - Card -> CardDeck -> CardGame 순으로 구현

Implementation

- CardGame의 구현
  - Class Templates with Fields and Method Signatures

```
class Card
{
    private int suit;
    private int value;

    public Card();
    public int getValue() { }
}
```

```
class CardDeck
{
    private int currentIdx;
    private Card[] deck;

    public CardDeck()
    public void initialize() { }
    public void shuffle() { }
    public Card dealACard() { }
}
```

```
class CardGame
{
    private int winCnt;
    private int loseCnt;
    private int tieCnt;
    private CardDeck deck;

    public CardGame();
    void initialize() { }
    void playGames() { }
    void playAGame() { }
    int decideWinner(Card) { }
    void incrementCounts(int) { }
}
```

How to Start a Card Game

- Solution-1 : CardGame의 메소드를 static으로 구현
  - JVM calls main
  - main calls other static methods
- Solution-2 : CardGame 메소드를 non-static 으로 구현
  - main에서 CardGame 객체를 생성
  - CardGame을 시작하는 메소드(가령, run) 호출
  - 시작 메소드가 game의 flow를 진행

```
class CardGame
{
   CardDeck deck;
   int WinCount;
   int LoseCount;
   int TieCount;
   public static void main(String[] args)
   {
      CardGame aGame = new CardGame();
      aGame.run();
   }
   public CardGame() { …}
   public void run()
   {
      initialize();
      playGames();
      printCounts();
   }
   …
}
```

Class Design Hints

- Always keep data private.
- Always initialize data.
- Don't use too many basic types in a class:

```
public class Customer
{
   private String street;
   private String city;
   private String state;
   private int zip;
   ...
}
```

⇨

```
public class Customer
{
   private Address shippingAddress;
   ...
}
```

- Not all fields need field accessors and mutators:

```
public class Address
{
   ...
   private int zip;
   public void setZip(int newZip) { zip = newZip; }
   ...
}
```

```
public class Stack
{
   ...
   private int top = -1;
   public int getTop() { return top; }
   public void setTop(int t) { top = t; }
   ...
}
```

Class Design Hints

- Break up classes that have too many responsibilities:

```
public class CardDeck // bad design
{
   private int[] value;
   private int[] suit;
   ...
   public void shuffle() { . . . }
   public int getTopValue() { . . . }
   public int getTopSuit() { . . . }
}
```

```
public class CardDeck
{
   private Card[] cards;
   ...
   public void shuffle() { . . . }
   public Card draw() { . . . }
}
public class Card
{
   private int value;
   private int suit;
   ...
   public int getValue() { . . . }
   public int getSuit() { . . . }
}
```

- Make the names of your classes and methods reflect their responsibilities.
- Prefer immutable classes.