Clean code

clean_code.pdf

▼ 6월 20 pm9:00~, 3장까지

2장

클래스이름

클래스 이름과 객체 이름은 명사 Or 명사구가 적합

ex) Customer, Account

메서드 이름

동사나 동사구가 적합

ex) save, postPayment

메서드는 인수를 설명하는 이름을 사용

```
Complex fulcrumPoint = Complex.FromRealNumber(23.0); //아래 코드보다 보기 좋음

Complex fulcrumPoint = new Complex(23.0);
```

기발한 이름은 피하라

한 개념에 한 단어를 사용하라

똑같은 메서드를 클래스마다 fetch, get 으로 다르게 부르면 안됨

독자적이고 일관적이어야 함

말장난을 하지마라

한 단어를 두 가지 목적으로 사용하면 안됨

해법 영역에서 가져온 이름을 사용하라

전산 용어, 알고리즘 이름, 패턴 이름, 수학 용어 등을 사용해도 무난

기술개념에는 기술이름 가장 적합

문제영역에서 가져온 이름을 사용하라

적절한 프로그래머 용어가 없다면 문제 영역에서 이름을 가져옴

의미있는 맥락을 추가하라

FirstName —> addrFirstName

예제에서는 함수 안에서 사용하는 세 변수를 작은 조각으로 쪼개고 클래스에 넣었음 → 맥락이 분명해짐

불필요한 맥락을 없애라

이름에 불필요한 맥락을 추가하지 말자

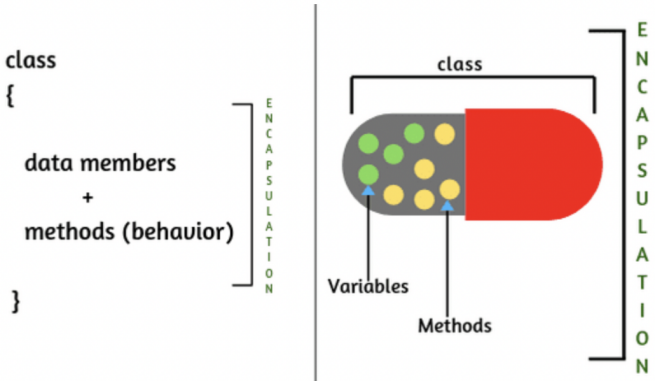
스터디 후기

- 전에 회사에서 코딩을 잘하는 분에게 코딩잘하는 법을 알려달라고 했는데 그 분이 하는말씀이 변수를 잘 써야한다 였음 → 그때는 단순히 a,b,c ..이렇게 쓰지말라는 의미로만 알았는데 이 책을 읽으니 위 처럼 변수이름을 작성해야한다는 뜻인것 같음
- 클래스와 메서드를 거의 사용하지 않아 대부분 변수이름만 고민하였음
- 변수이름조차 너무 간단하게 지어서 추후 다시볼때 내 변수이름을 보고 변수가 의미하는 뜻을 유추할 수 없었음
- 아직 의미있는 맥락으로 변수이름 지정, 함수 이름 지정이 어렵게 느껴짐 → 클래스와 메서드 연습을 많이 해야겠다라는 생각이 들었음

▼ 7월 11일

클래스 체계

- 캡슐화: **클래스 안에 서로 연관있는 속성과 기능들을 하나의 캡슐(capsule)로 만들어 데이터를 외부로부터 보호하는 것**



클래스는 작아야한다

크기: 작아야한다

크기: 더 작아야한다

맡은책임으로 크기를 센다

클래스이름이 모호하다면 클래스 책임이 크기 때문

10-2 충분히 작을까?

```
public class SuperDashboard extends JFrame implements MetaDataUser
 public Component getLastFocusedComponent()
 public void setLastFocused(Component lastFocused)
 public int getMajorVersionNumber()
 public int getMinorVersionNumber()
 public int getBuildNumber()
}

--> SuperDashboard 책임이 너무 많다
간결한 이름이 떠오르지않으면 클래스가 커서 그런것
```

- 단일 책임 원칙
  - 클래스나 모듈을 변경할 이유가 하나이여야 한다
  - 10-2를 변경해야할 이유는 버전변동성이 있다.
    - SuperDashboard는 소프트웨어 버전 정보를 추적
    - SuperDashboard는 자바스윙 컴포넌트 관리

    10-3 단일책임 클래스 : version이라는 독자적인 클래스를 만들자

    ```
    public class Version {
     public int getMajorVersionNumber()
     public int getMinorVersionNumber()
     public int getBuildNumber()
    }
    ```

  - **프로그램이 돌아가면 일이 끝났다고 여긴다. —> 다음 관심사로 전환하지 않음**
  - 깨끗하고 체계적인 소프트웨어 → 클래스 몇개가 아니라 작은 클래스 여럿으로 이뤄진 시스템이 더 바람직하다
- 응집도
  - 응집도가 높아지도록 변수와 메소드를 적절히 분리해 새로운 클래스 두세개로 쪼개준다

```
public class Stack {
 private int topOfStack = 0;
 List<Integer> elements = new LinkedList<Integer>();
 public int size() {
 return topOfStack;
 }
 public void push(int element) {
 topOfStack++;
 elements.add(element);
 }
 public int pop() throws PoppedWhenEmpty {
 if (topOfStack == 0)
 throw new PoppedWhenEmpty();
 int element = elements.get(--topOfStack);
 elements.remove(topOfStack);
```

```
 return element;
 }
}
--> 응집도가 높은 클래스
```

```
class Stack:
    def __init__(self):
        self.top_of_stack = 0
        self.elements = []

    def size(self):
        return self.top_of_stack

    def push(self, element):
        self.top_of_stack += 1
        self.elements.append(element)

    def pop(self):
        if self.top_of_stack == 0:
            raise PoppedWhenEmpty
        self.top_of_stack -= 1
        return self.elements.pop(self.top_of_stack)
--> 응집도가 높은 클래스 python으로 바꿈
```

또 다른 예제

```
class Circle:
    def __init__(self, radius):
        self.radius = radius

    def calculate_area(self):
        return 3.14 * self.radius ** 2

    def calculate_circumference(self):
        return 2 * 3.14 * self.radius

    def print_info(self):
        area = self.calculate_area()
        circumference = self.calculate_circumference()
        print("반지름:", self.radius)
        print("넓이:", area)
        print("둘레:", circumference)

class Cylinder:
    def __init__(self, radius, height):
        self.radius = radius
        self.height = height

    def calculate_volume(self):
        return 3.14 * self.radius ** 2 * self.height

    def print_info(self):
        volume = self.calculate_volume()
        print("반지름:", self.radius)
        print("높이:", self.height)
        print("부피:", volume)

# 예시 사용
circle = Circle(5)
cylinder = Cylinder(3, 10)

circle.print_info()
print()
cylinder.print_info()
```

- 응집도를 유지하면 작은 클래스 여럿이 나온다

```
package literatePrimes;

public class PrintPrimes {
    public static void main(String[] args) {
        final int M = 1000;
        final int RR = 50;
        final int CC = 4;
        final int WW = 10;
        final int ORDMAX = 30;
        int P[] = new int[M + 1];
        int PAGENUMBER;
        int PAGEOFFSET;
        int ROWOFFSET;
        int C;
        int J;
        int K;
        boolean JPRIME;
        int ORD;
        int SQUARE;
        int N;
        int MULT[] = new int[ORDMAX + 1];
        J = 1;
        K = 1;
        P[1] = 2;
        ORD = 2;
        SQUARE = 9;
        while (K < M) {
            do {
                J = J + 2;
                if (J == SQUARE) {
                    ORD = ORD + 1;
                    SQUARE = P[ORD] * P[ORD];
                    MULT[ORD - 1] = J;
                }
                N = 2;
                JPRIME = true;
                while (N < ORD && JPRIME) {
                    while (MULT[N] < J)
                        MULT[N] = MULT[N] + P[N] + P[N];
                    if (MULT[N] == J)
                        JPRIME = false;
                    N = N + 1;
                }
            } while (!JPRIME);
            K = K + 1;
            P[K] = J;
        }
        {
            PAGENUMBER = 1;
            PAGEOFFSET = 1;
            while (PAGEOFFSET <= M) {
                System.out.println("The First " + M +
                                   " Prime Numbers --- Page " + PAGENUMBER);
                System.out.println("");
                for (ROWOFFSET = PAGEOFFSET; ROWOFFSET < PAGEOFFSET + RR; ROWOFFSET++) {
                    for (C = 0; C < CC; C++)
                        if (ROWOFFSET + C * RR <= M)
                            System.out.format("%10d", P[ROWOFFSET + C * RR]);
                    System.out.println("");
                }
                System.out.println("\f");
                PAGENUMBER = PAGENUMBER + 1;
                PAGEOFFSET = PAGEOFFSET + RR * CC;
            }
        }
    }
}
```

- 코드는 주어진 범위 내에서 소수를 찾아 출력하는 프로그램
- `M`: 출력할 소수의 개수
- `RR`: 한 페이지에 출력할 행(row)의 개수
- `CC`: 한 행(row)에 출력할 열(column)의 개수
- `WW`: 소수 출력 시 사용할 공간의 너비(width)
- `ORDMAX`: 다른 변수들을 계산하는데 사용되는 최대값

주요 로직:

- `P`: 소수를 저장하는 배열
- `J`, `K`, `ORD`, `SQUARE`, `N`, `MULT`: 소수를 찾기 위한 변수들

주요 로직은 다음과 같습니다:

1. 초기화: `J`, `K`, `ORD`, `SQUARE`, `N`, `MULT` 등을 초기화합니다.
2. 소수 찾기: `K`가 `M`보다 작을 때까지 반복하여 소수를 찾습니다.
   - `J`를 2씩 증가시키며 홀수를 찾습니다.
   - `J`가 `SQUARE`와 같을 경우, `ORD`를 증가시키고 `SQUARE`을 업데이트합니다.
   - `N`을 2부터 `ORD`까지 반복하면서 `MULT` 배열을 업데이트합니다.
   - `MULT[N]`이 `J`와 같을 경우, `J`는 소수가 아니므로 `JPRIME`을 `false`로 설정합니다.
   - 반복문이 종료된 후 `J`가 소수이면 `P` 배열에 저장합니다.
3. 소수 출력: 페이지 단위로 소수를 출력합니다.
   - 페이지 번호(`PAGENUMBER`)와 페이지 오프셋(`PAGEOFFSET`)을 설정합니다.
   - `PAGEOFFSET`이 `M`보다 작거나 같을 때까지 반복합니다.
   - 페이지 헤더를 출력합니다.
   - 각 행(row)에 대해 `ROWOFFSET`부터 `ROWOFFSET + RR`까지 반복하면서 열(column)에 대한 반복을 수행하고 소수를 출력합니다.
   - 페이지 구분을 위해 개행 문자(`\f`)를 출력합니다.
   - 페이지 번호와 페이지 오프셋을 업데이트합니다.

10-6. 리팩토링한 버전

```
package literatePrimes;

public class PrimePrinter {
    public static void main(String[] args) {
        final int NUMBER_OF_PRIMES = 1000;
        int[] primes = PrimeGenerator.generate(NUMBER_OF_PRIMES);

        final int ROWS_PER_PAGE = 50;
        final int COLUMNS_PER_PAGE = 4;
        RowColumnPagePrinter tablePrinter =
            new RowColumnPagePrinter(ROWS_PER_PAGE,
                                     COLUMNS_PER_PAGE,
```

```
                                    "The First " + NUMBER_OF_PRIMES +
                                    " Prime Numbers");
        tablePrinter.print(primes);
    }
}
```

1. 모듈 임포트: `PrimeGenerator` 모듈을 임포트하여 소수 생성 기능을 사용합니다.
2. 클래스 분리: `PrimePrinter`와 `RowColumnPagePrinter` 클래스가 분리되어 있습니다. `PrimePrinter` 클래스는 `main` 메서드를 가지고 있으며, 소수 생성 및 출력을 담당합니다. `RowColumnPagePrinter` 클래스는 행(row), 열(column), 페이지(page) 단위로 출력을 관리하는 역할을 합니다.
3. 생성자 및 인스턴스 메서드: `RowColumnPagePrinter` 클래스는 `__init__` 생성자를 가지며, `print` 인스턴스 메서드를 사용하여 출력 작업을 수행합니다. 이를 통해 클래스의 인스턴스를 생성하고 메서드를 호출하여 출력을 관리할 수 있습니다.
4. 변수명 변경: 일부 변수명이 변경되었습니다. 예를 들어, `NUMBER_OF_PRIMES` 대신에 `primes`로 변수명이 변경되었습니다.
5. 문자열 포맷팅: 출력 시 `print("{:10d}".format(primes[index]), end="")`와 같이 문자열 포맷팅을 사용하여 출력 너비를 조정하고 있습니다.

위의 변경 사항을 통해 코드의 구조와 가독성이 개선되었으며, 클래스의 역할과 책임이 분리되어 코드를 유지보수하기 쉬워졌습니다. 또한, `RowColumnPagePrinter` 클래스를 통해 출력에 관련된 기능을 캡슐화하여 재사용성을 높일 수 있게 되었습니다.

10.7

```
package literatePrimes;

import java.io.PrintStream;

public class RowColumnPagePrinter {
    private int rowsPerPage;
    private int columnsPerPage;
    private int numbersPerPage;
    private String pageHeader;
    private PrintStream printStream;

    public RowColumnPagePrinter(int rowsPerPage, int columnsPerPage, String pageHeader) {
        this.rowsPerPage = rowsPerPage;
        this.columnsPerPage = columnsPerPage;
        this.pageHeader = pageHeader;
        numbersPerPage = rowsPerPage * columnsPerPage;
        printStream = System.out;
    }

    public void print(int data[]) {
        int pageNumber = 1;
        for (int firstIndexOnPage = 0; firstIndexOnPage < data.length; firstIndexOnPage += numbersPerPage) {
            int lastIndexOnPage = Math.min(firstIndexOnPage + numbersPerPage - 1, data.length - 1);
            printPageHeader(pageHeader, pageNumber);
            printPage(firstIndexOnPage, lastIndexOnPage, data);
            printStream.println("\f");
            pageNumber++;
        }
    }

    private void printPage(int firstIndexOnPage, int lastIndexOnPage, int[] data) {
        int firstIndexOfLastRowOnPage = firstIndexOnPage + rowsPerPage - 1;
        for (int firstIndexInRow = firstIndexOnPage; firstIndexInRow <= firstIndexOfLastRowOnPage; firstIndexInRow++) {
```

```
            printRow(firstIndexInRow, lastIndexOnPage, data);
            printStream.println();
        }
    }

    private void printRow(int firstIndexInRow, int lastIndexOnPage, int[] data) {
        for (int column = 0; column < columnsPerPage; column++) {
            int index = firstIndexInRow + column * rowsPerPage;
            if (index <= lastIndexOnPage) {
                printStream.format("%10d", data[index]);
            }
        }
    }

    private void printPageHeader(String pageHeader, int pageNumber) {
        printStream.println(pageHeader + " --- Page " + pageNumber);
        printStream.println();
    }

    public void setOutput(PrintStream printStream) {
        this.printStream = printStream;
    }
}
```

1. `import` 문 추가: `java.io.PrintStream` 클래스를 임포트하여 출력에 사용합니다.
2. 멤버 변수 및 생성자 변경: `numbersPerPage` 멤버 변수를 추가로 선언하고, 생성자에서 초기화합니다. `printStream` 멤버 변수를 `System.out`으로 초기화합니다. 또한, 생성자의 매개변수로 `pageHeader`를 전달받도록 변경합니다.
3. `print` 메서드 변경: `data` 배열을 인자로 받는 `print` 메서드를 추가합니다. 이 메서드는 페이지별로 데이터를 출력합니다. `pageNumber` 변수를 사용하여 페이지 번호를 관리하며, `firstIndexOnPage`를 기준으로 페이지 내 첫 번째와 마지막 인덱스를 계산하여 해당 범위의 데이터를 출력합니다.
4. `printPage` 메서드 변경: 페이지 내 행(row)을 출력하는 `printPage` 메서드가 변경되었습니다. `firstIndexOfLastRowOnPage` 변수를 사용하여 페이지 내 마지막 행 인덱스를 계산하고, 해당 범위 내의 데이터를 출력합니다.
5. `printRow` 메서드 변경: `printRow` 메서드에서는 각 행(row)별로 데이터를 출력합니다. `column` 변수를 사용하여 열(column)을 반복하고, `index` 변수를 계산하여 해당 인덱스의 데이터를 출력합니다.
6. `printPageHeader` 메서드 변경: 페이지 헤더를 출력하는 `printPageHeader` 메서드가 변경되었습니다. `pageHeader`와 `pageNumber`를 인자로 받아 출력합니다.
7. `setOutput` 메서드 추가: `setOutput` 메서드를 추가하여 출력을 변경할 수 있도록 합니다.

위의 변경 사항을 통해 `RowColumnPagePrinter` 클래스의 구조와 기능이 개선되었습니다. 코드의 가독성이 향상되고, 출력 관련 기능을 캡슐화하여 재사용성이 높아졌습니다. 또한, 출력을 변경하기 위해 `setOutput` 메서드를 사용할 수 있게 되었습니다.

10-8

```
package literatePrimes;

import java.util.ArrayList;

public class PrimeGenerator {
    private static int[] primes;
    private static ArrayList<Integer> multiplesOfPrimeFactors;

    protected static int[] generate(int n) {
```

```
        primes = new int[n];
        multiplesOfPrimeFactors = new ArrayList<Integer>();
        set2AsFirstPrime();
        checkOddNumbersForSubsequentPrimes();
        return primes;
    }

    private static void set2AsFirstPrime() {
        primes[0] = 2;
        multiplesOfPrimeFactors.add(2);
    }

    private static void checkOddNumbersForSubsequentPrimes() {
        int primeIndex = 1;
        for (int candidate = 3; primeIndex < primes.length; candidate += 2) {
            if (isPrime(candidate))
                primes[primeIndex++] = candidate;
        }
    }

    private static boolean isPrime(int candidate) {
        if (isLeastRelevantMultipleOfNextLargerPrimeFactor(candidate)) {
            multiplesOfPrimeFactors.add(candidate);
            return false;
        }
        return isNotMultipleOfAnyPreviousPrimeFactor(candidate);
    }

    private static boolean isLeastRelevantMultipleOfNextLargerPrimeFactor(int candidate) {
        int nextLargerPrimeFactor = primes[multiplesOfPrimeFactors.size()];
        int leastRelevantMultiple = nextLargerPrimeFactor * nextLargerPrimeFactor;
        return candidate == leastRelevantMultiple;
    }

    private static boolean isNotMultipleOfAnyPreviousPrimeFactor(int candidate) {
        for (int n = 1; n < multiplesOfPrimeFactors.size(); n++) {
            if (isMultipleOfNthPrimeFactor(candidate, n))
                return false;
        }
        return true;
    }

    private static boolean isMultipleOfNthPrimeFactor(int candidate, int n) {
        return candidate == smallestOddNthMultipleNotLessThanCandidate(candidate, n);
    }

    private static int smallestOddNthMultipleNotLessThanCandidate(int candidate, int n) {
        int multiple = multiplesOfPrimeFactors.get(n);
        while (multiple < candidate)
            multiple += 2 * primes[n];
        multiplesOfPrimeFactors.set(n, multiple);
        return multiple;
    }
}
```

1. `ArrayList` 추가: `multiplesOfPrimeFactors`라는 `ArrayList<Integer>`를 추가하여 소인수의 배수를 저장합니다.
2. `generate` 메서드 변경: `generate` 메서드는 소수 배열 `primes`를 생성하는 역할을 합니다. 기존 코드에서 `multiplesOfPrimeFactors` 리스트를 초기화하고, `2`를 첫 번째 소수로 설정합니다.
3. `isPrime` 메서드 변경: `isPrime` 메서드는 주어진 숫자가 소수인지 확인합니다. 이를 위해 `isLeastRelevantMultipleOfNextLargerPrimeFactor` 메서드와 `isNotMultipleOfAnyPreviousPrimeFactor` 메서드를 사용합니다. `isLeastRelevantMultipleOfNextLargerPrimeFactor`는 다음으로 큰 소인수의 배수인지 확인하고, `isNotMultipleOfAnyPreviousPrimeFactor`는 이전 소인수의 배수가 아닌지 확인합니다.
4. `smallestOddNthMultipleNotLessThanCandidate` 메서드 변경: `smallestOddNthMultipleNotLessThanCandidate` 메서드는 주어진 숫자의 n번째 소인수의 배수를 반환합니다. 기존 코드와 비교하여 몇 가지 변경이 있습니다.

위의 변경 사항을 통해 `PrimeGenerator` 클래스가 소수를 생성하는 데 사용되며, 소인수와 배수를 관리하는 추가적인 기능이 포함되어 있음을 알 수 있습니다.

- 리팩토링한 프로그램은 좀 더 길고 서술적인 변수 이름을 사용함
- 리팩토링한 프로그램은 코드에 주석을 추가하는 수단으로 함수선언과 클래스 선언을 활용
- 가독성을 높이고자 공백을 추가하고 형식을 맞춤