Clean Code

10. 클래스

알파서클 연구원
Luna

클래스 체계

- (JAVA) 클래스를 정의하는 순서는 아래와 같다.
  - static public constant
  - static private variable
  - private variable
  - public variable (거의 필요하지 않음)
  - public method
    - private method (자신을 호출하는 공개 메소드 직후)
- 변수와 유틸리티 함수는 가능한 공개하지 않는다. (테스트 제외)
- 캡슐화를 푸는 것은 언제나 최후의 수단이다.

클래스의 크기는 작아야 한다

- 클래스의 크기를 계산하는 척도는 맡은 책임이다.
- 클래스의 이름으로 책임을 기술해야 한다.
  - 간결한 이름이 떠오르지 않거나 모호하다면 클래스의 책임이 많기 때문이다. (Processor, Manger, Super 등)
  - if, and, or, but을 사용하지 않고 25단어 내외여야한다.

```
public class SuperDashboard extends JFrame implements MetaDataUser {
  public Project getProject();
  public String getCurrentDir();
  public Class[] getDataBaseClasses();
  public int getMajorVersionNumber();
  ... // 공개 메서드의 수만 해도 약 70개
}
```

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

}
```

70개의 메서드를 가진 클래스에서 주요 기능 5개만 뽑았지만 오른쪽도 책임이 너무 많다.

버전 정보 추적, JFrame 컴포넌트 관리

응집도

- 클래스는 인스턴스 변수가 적어야 한다.
- 각 클래스 메서드는 클래스 인스턴스 변수를 하나 이상 사용해야 한다.
- 메서드가 변수를 많이 사용할수록 메서드와 클래스의 응집도가 높아진다.
- 몇몇 메서드에서만 사용하는 인스턴스 변수가 많아진다면 새로운 클래스로 쪼개야 한다.
- 응집도가 높아지도록 변수와 메서드를 적절히 분리해 새로운 클래스로 쪼개줘야 한다.

응집도가 높은 Stack 클래스

size()를 제외한 두 메서드는 클래스에 선언된 두 인스턴스 변수를 모두 사용한다.

```
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;
  }
}
```

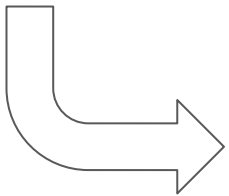
단일 책임 원칙 (SRP: Single Responsibillity Principle)

- 클래스나 모듈은 하나의 역할만을 해야한다.
- 클래스를 변경할 이유(책임)는 단 하나여야 한다.
- 큰 클래스 몇 개가 있는 것보다 작은 클래스 여럿으로 이뤄진 시스템이 더 바람직하다.

응집도를 유지하면 작은 클래스 여럿이 나온다

- 큰 함수를 작은 함수 여럿으로 나누기만 해도 클래스는 많아진다.
  - 나누는 과정중 큰 함수에서 정의된 변수를 사용한다면 인스턴스 변수로 승격한다
  - **=>** 인수가 필요 없어지며 함수로 쪼개기 쉬워진다.
  - **=>** 클래스가 응집력을 잃으며 그만큼 쪼개기 더 쉬워진다.

```
public class PrintPrims {
  public static void main(String[] args) {
    // 소수목록을 생성하며 주어진 행과 열에 맞춰 페이지를 출력하는 함수
  }
}
```



```
public class PrimePrinter {
  // 실행 환경을 책임진다.
  // 출력하는 곳을 바꾼다면 이쪽을 수정하면 된다.
}

public class RowColumnPagePrinter {
  // 숫자 목록을 주어진 형식에 맞춰준다.
  // 출력 형식을 바꾼다면 이쪽을 수정하면 된다.
}

public class PrimeGenerator {
  // 소수 목록을 생성한다.
  // 소수를 구하는 알고리즘을 바꾼다면 이쪽을 수정하면 된다.
}
```

변경하기 쉬운 클래스

- 어떤 시스템이든 지속적인 변경이 생기며 클래스에 손대면 다른 코드를 망가뜨릴 잠정적인 위험이 존재한다.
- 깨끗한 시스템은 클래스를 체계적으로 정리해 변경에 수반하는 위험을 낮춰야 한다.

```
public class Sql {
  public Sql(String table, Column[] columns);
  public String create();
  public String insert(Object[] fields);
  public String select(Column column, String pattern);
  public String select(Criteria criteria);
  // select에서만 사용 (SRP 위반)
  private String selectWithCriteria(String criteria);

  // 새로운 SQL문을 추가하면 위험이 생길 수 있으며 테스트도 다시 해야함
  // select문에 내장된 다른 select 문을 지원하려면 Sql클래스를 고쳐야함 (SRP 위반)
```

```
abstract public class Sql {
      public Sql(String table, Column[] columns);
      abstract public String generate();
}

public class CreateSql extends Sql {
  public CreateSql(String table, Column[] columns);
  @Override public String generate();
}

public class SelectSql extends Sql {
  public SelectSql(String table, Column[] columns);
  @Override public String generate();
}

public class InsertSql extends Sql {
  public InsertSql(String table, Column[] columns, Object[] fields);
  @Override public String generate();
  private String valuesList(Object[] fields, final Column[] columns);
}

// 새로운 SQL문을 추가하려면 그저 Sql을 상속받는 새 클래스를 만들면 됨
// 개방 폐쇄 원칙(OCP: Open-Closed Principle)을 지원
```

변경으로부터 격리

- 시스템의 결합도를 낮추면 유연성과 재사용성도 더욱 높아진다.
  - 결합도가 낮다는 뜻은 각 시스템 요소가 다른 요소로부터, 변경으로부터 격리가 잘 되어있다는 의미이다.
  - 결합도를 최소로 줄이면 의존 역전 원칙(DIP: Dependency Inversion Principle)을 따르게 된다. (클래스가 상세한 구현에 의존하지 않고 추상화에 의존해야함)

인터페이스를 사용하는 Portfolio 클래스

- 외부 TokyoStockExchangeAPI(상세한 구현 클래스)를 사용하는 Portfolio 클래스
  - 테스트 코드가 시세 변화에 영향받음
- StockExchange Interface(추상적)를 사용하는 Portfolio 클래스
  - :실제 주가를 얻어오는 출처나 얻어오는 방식 등을 숨김
  - => 다른 API를 사용하게 되더라도 재사용 가능
  - => 해당 인터페이스를 구현하는 테스트용 클래스 생성 가능

```
// 주식 기호를 받아 현재 주식 가격을 반환하는 인터페이스
// 이 인터페이스를 구현하는 TokyoStockExchange 클래스를 구현해서 사용
public interface StockExchange {
  Money currentPrice(String symbol);
}

public Portfolio {
  private StockExchange exchange;
  public Portfolio(StockExchange exchange) {
    this.exchange = exchange;
  }
  // ...
}
```

```
//언제나 100불을 반환하는 클래스
public class PortfolioTest {
  private FixedStockExchangeStub exchange;
  private Portfolio portfolio;

  @Before
  protected void setUp() throws Exception {
    exchange = new FixedStockExchangeStub();
    exchange.fix("MSFT", 100);
    portfolio = new Portfolio(exchange);
  }

  @Test
  public void GivenFiveMSFTTotalShouldBe500() throws Exception {
    portfolio.add(5, "MSFT");
    Assert.assertEquals(500, portfolio.value());
  }
}
```

OOP 5원칙 SOLID

- 단일 책임 원칙 (SRP: Single Responsibillity Principle)
- 개방 폐쇄 원칙 (OCP: Open Close Principle)
  - 확장에 대해서는 열려있고, 변경에 대해서는 닫혀있게 만들어야 한다.
    - 즉 새로운 기능 추가는 쉽지만 변경의 영향이 제한적이여야 한다.
- 리스코프 치환 원칙 (LSP: Liskov Substitution Principle)
  - 같은 형을 가지는 객체로 대체되어 사용할 수 있어야 한다.
  - 동일한 클래스를 상속하는 경우, 상위 클래스를 접근하는 Client 코드들은 상속받은 클래스로 대체하더라도 변경이 없어야 한다.
- 인터페이스 분리 원칙 (ISP: Interface Segregation Principle)
  - 필요한 인터페이스 각각에 대해서 따로 정의해서 사용하는 것이 좋다.
  - 즉, 각각의 인터페이스를 공용화해서 한 인터페이스가 너무 많은 기능을 제공하도록 만드는 것은 확장성 및 변경의 영향을 제어하지 못하게 만든다.
  - 만능보다는 한가지 일에 특화된 것이 좋다.
- 의존 역전 원칙 (DIP: Dependency Inversion Principle)
  - 고수준 모듈은 저수준 모듈의 구현에 의존해서는 안 된다.
  - 저수준 모듈은 고수준 모듈에서 정의한 추상 타입에 의존해야 한다.
  - 구체적인 것에 의존하지 말고 추상화에 의존해라.

https://mangkyu.tistory.com/194