Clean Code

15장 - JUnit 들여다보기 & 16장 - SerialDate 리팩터링

ComparisionCompactor

- JUnit 프레임워크 내 모듈
- 두 문자열을 받아 차이를 반환하는 모듈
  - ex. ABCDE 와 ABXDE를 입력받으면 AB[X]DE를 반환
- 테스트 커버리지가 100프로
  - 테스트 케이스가 모든 if,for문을 실행
  - 코드가 잘 분리되었다는 의미

f 접두어 제거

개선 전

```
private val fContextLength: Int
private val fExpected: String
private val fActual: String
private lateinit var fPrefix: Int
private lateinit var fSuffix: Int
```

개선 후

```
private val contextLength: Int
private val expected: String
private val actual: String
private lateinit var prefix: Int
private lateinit var suffix: Int
```

캡슐화 되지 않은 조건문 & 명확하지 않은 이름

개선 전

```
fun compact(message: String) {
  if(expected == null ||
    actual ==null ||
      areStringsEqual()){
    return Assert.format(message,expected,actual)
  }

  val expected = compactString(this.expected)
  val actual = compactString(this.actual)
}
```

캡슐화 되지 않은 조건문 & 명확하지 않은 이름

- 부정문보다는 긍정문으로
- this.expected와 expected는 의미하는 바가 다르지만 동일한 이름
  - 지역변수의 이름을 수정

개선 후

```
fun compact(message: String) {
  if(canBeCompacted()){
    findCommonPrefix()
    findCommonSuffix()
    val compactExpected = compactString(expected)
    val compactActual = compactString(actual)
    return Assert.format(message,compactExpected,compactActual)

  }
  return Assert.format(message,expected,actual)
}

fun canBeCompacted(): Boolean{
  return expected != null &&
    actual != null &&
      areStringsEqual().not()
}
```

함수 분리

개선 전

```
fun compact(message: String) {
  if(canBeCompacted()){
    findCommonPrefix()
    findCommonSuffix()
    val compactExpected = compactString(expected)
    val compactActual = compactString(actual)
    return Assert.format(message,compactExpected,compactActual)

  }
  return Assert.format(message,expected,actual)

}

fun canBeCompacted(): Boolean{
  return expected != null &&
    actual != null &&
      areStringsEqual().not()
}
```

함수 분리

- formatCompactedComparison으로 이름 변경
  - compact라는 이름이지만 canBeCompacted가 false라면 압축하지 않음
  - 형식적인 문자열을 리턴한다는 의미도 없음
- 압축하는 코드는 따로 분리
  - 문자열 형식을 만드는 것만 책임
  - 압축된 값을 저장하는 변수가 멤버변수로

개선 후

```
private var compactExpected = ""
private var compactActual = ""

fun formatCompactedComparison(message: String) {
  if(canBeCompacted()){
    compactExpecteedAndActual()
    return Assert.format(message,compactExpected,compactActual)

  }
  return Assert.format(message,expected,actual)

}

fun canBeCompacted(): Boolean{
  return expected != null &&
    actual != null &&
      areStringsEqual().not()
}
```

함수의 일관성

개선 전

```
private var compactExpected = ""
private var compactActual = ""

fun compactExpectedAndActual() {
    findCommonPrefix()
    findCommonSuffix()
    compactExpected = compactString(expected)
    compactActual = compactString(actual)

}
```

함수의 일관성

- 반환값이 있는 함수와 없는 함수가 섞여있음
  - 접두어,접미어의 인덱스를 리턴
- 함수의 시간적 결합 존재
  - findCommonPrefix -> findCommonSuffix
  - findCommonPrefixAndSuffix 내부에서 findCommonPrefix를 호출

개선 후

```
private var compactExpected = ""
private var compactActual = ""

fun compactExpectedAndActual() {
    findCommonPrefixAndSuffix()
    compactExpected = compactString(expected)
    compactActual = compactString(actual)

}

fun findCommonPrefixAndSuffix() {
  findCommonPrefix()
  //suffix 계산
}
```

명확한 이름

- suffixIndex보다 suffixLength에 더 의미가 가까움
- index는 0부터 시작하지만 length는 1부터 시작하기 때문에 +1 , -1을 해주는 수정작업

불필요한 if문 제거

- 코드 리팩터링 과정에서 if문의 의미가 없어지는 경우 존재
- if (suffixLength > 0) 은 suffixLength로 변경되면서 길이는 항상 1 이상이므로 불필요한 if문이 됨

결론

- 리팩터링은 코드가 어느 수준에 이를 때까지 시행착오를 반복하는 작업
  - 원래 코드로 다시 되돌아가는 경우도 흔하다
- 리팩터링이 불필요한 모듈은 없다.
  - 코드를 항상 깨끗하게 유지하는 책임은 모두에게 있다.

SerialDate

- JCommon 라이브러리 내 클래스
- 날짜를 표현하는 클래스
  - Date 클래스는 너무 정밀하고 특정 날짜 하루를 표현하고 싶을 때 사용하기 위해 만든 클래스
  - ex. 2015년 1월 15일
  - 시간 날짜보다 순수 날짜를 표현

돌려보자

- 테스트 케이스를 돌려보자
- SerialDate의 기존 테스트 케이스의 테스트 커버리지는 50프로 정도였고 저자가 직접 정의한 테스트 케이스 중 실패하는 테스트 케이스도 존재
- 일단 통과하지 않는 테스트 케이스가 돌아가도록 메서드 수정
- 모든 테스트 케이스가 돌아간다면 일단 SerialDate 코드가 제대로 돈다고 믿은 후, 코드를 '올바로' 고쳐야 한다.
- 테스트 케이스도 안 돌아가는데 리팩터링하면 안됨

고쳐보자

불필요한 주석 삭제

- 법적인 주석 (ex. 저작권) 외 변경 이력, html 주석 삭제

클래스 이름 수정

- 일련 번호 (serial number)를 사용해서 클래스를 구현했기 때문에 Serial이라고 이름 붙인 이유
  - 1899년 12월 30일 기준으로 경과한 날짜수를 사용
- 일련번호보다는 상대 오프셋이라는 용어가 더 적합
  - ordinal 용어 사용
- Serial이라고 구현을 암시하는 이름을 붙였지만 실제는 추상 클래스
  - DayDate 로 이름 변경

고쳐보자

enum class

- static final 상수들을 가지고 있는 클래스를 enum class로 정의
- enum class로 정의하면서 int code가 올바른지 확인하는 코드들 제거

변수와 메서드 이동

```
final val EARLEST_DATE_ORDINAL = 2
final val LATEST_DATE_ORDINAL = 2958465
```

- EARLEST_DATE_ORDINAL이 이 클래스를 구현하는 SpreadSheetDate클래스의 날짜 표현방식을 위해 2로 표현
- DayDate와는 상관이 없으므로 SpreadSheetDate로 이동

고쳐보자

생성 클래스

- 구체적인 구현 정보에 관한 변수는 파생 클래스가 가지고 있어야 함
- createInstance 메서드에서 파생 클래스인 SpreadSheetDate 클래스를 생성하면서 구체적인 구현 정보를 DayDate가 필요로 하게 됨
    - ABSTRACT FACTORY 패턴을 이용해서 Factory클래스를 만들고 구체적인 구현 정보를 Factory 클래스로 옮김

FACTORY 패턴

하위 클래스가 어떤 클래스를 생성할 지 결정하는 패턴

FACTORY METHOD 패턴

- 실제 객체를 만드는 것은 하위 객체에 맡겨 객체 생성을 캡슐화
- 추상 클래스에서는 객체 생성을 위한 추상 메서드만 정의
- ShapeFactory가 모든 Shape의 하위 객체들을 다 알고 있던 구조
    - Shape의 하위 메서드들이 ShapeFactory를 의존하는 구조로 의존성이 뒤집어짐

```
class ShapeFactory {
    fun create(type: ShapeType): Shape {
        return when (type) {
            ShapeType.RECTANGLE -> Rectangle()
            ShapeType.CIRCLE -> Circle()
        }
    }

}

//val shapeFactory = ShapeFactory()
//shapeFactory.create(ShapeType.RECTANGLE)
```

FACTORY METHOD 패턴

하위 클래스가 어떤 클래스를 생성할 지 결정하는 패턴

```
interface ShapeFactory {
    fun create(): Shape
}

class RectangleFactory() : ShapeFactory {
    override fun create(): Shape {
        return Rectangle()
    }
}

//val factory = RectanglrFactory()
//factory.create()
```

```
interface ShapeFactory {
    fun create(): Shape
}

class Rectangle private constructor() {
    companion object Factory: ShapeFactory{
        override fun create(): Shape {
            return Retangle()
        }

    }
}

//val rect = Rectangle.create()
```

FACTORY 패턴

하위 클래스가 어떤 클래스를 생성할 지 결정하는 패턴

ABSTRACT FACTORY 패턴

- 인터페이스를 구성하여 사용할 수 있게 하는 패턴
- 다양한 구성 요소별로 객체의 집합을 생성해야 할 때 유용
  - ex. GUI

Kotlin 복사 캡션

```kotlin
interface ShapeFactory {
    fun createSize(): Size
    fun createColor(): Color
}

class Rectangle private constructor() {
    companion object Factory : ShapeFactory {
        override fun createSize(): Size {
            return Size.createSize()
        }

        override fun createColor(): Color {
            return Color.createRandomColor()
        }
    }
}

//val factory = Rectangle.Factory
//val size = factory.createSize()
//val color = factory.createColor()
//size.width = 10f
//color.paint
```

https://jusungpark.tistory.com/14
https://ko.wikipedia.org/wiki/%EC%B6%94%EC%83%81_%ED%8C%A9%ED%86%A0%EB%A6%AC_%ED%8C%A8%ED%84%B4

결론

- 불필요한 주석 삭제
- enum을 독자적인 클래스로
- 정적 변수,메서드 새 클래스(DateUtil)로 옮김
- 일부 추상메서드를 DayDate로 끌어올림
- enum class를 fromInt()나 toInt() 접근자로 index 필드에 접근
- 중복 메서드 삭제
- 숫자 1을 그대로 사용하지 않고 Month.JANUARY, Day.SUNDAY.toInt()처럼 이름을 부여줌
- 알고리즘 수정