# T.C.

# SAKARYA ÜNİVERSİTESİ

## BİLGİSAYAR VE BİLİŞİM BİLİMLERİ FAKÜLTESİ

## BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ

YAZILIM TESTİ ÖDEV RAPORU

# Ödev1
# Java Dosyası Okuma , Operatör, Operand ve Fonksiyon Analizi Yapan Bir Kütüphane Hazırlama ve Yazılım Testini Gerçekleme

**B191210028-Saadet Elif Tokuoğlu**

**SAKARYA**

**Nisan, 2023**

# Java Dosyası Okuma ve Operatör, Operand ve Fonksiyon Analizi Yapan Bir Kütüphane Hazırlama ve Yazılım Testini Gerçekleme

Saadet Elif Tokuoğlu

*B191210028 / BSM462 / A /1.Öğretim*

*Saadet.tokuoglu@ogr.sakarya.edu.tr*

Bu ödevde bizden Java dilini kullanmamız isteniyor. Ödevde bir *.java dosyasının okunması ve dosyanın içindeki , tekli , ikili ,sayısal , ilişkisel, mantıksal operatörlerin sayısını ve toplam fonksiyon , toplam operand sayılarını da verebilecek özelliklerin olacağı bir kütüphane yazılması ve bu kütüphaneni mock , faker , parameterized ,repeated özelliklerinde birim testlerinin ve beş adet entegrasyon testinin yazılması isteniyor.

Anahtar Kelimeler: Regex , Birim ve Entegrasyon Testi , Operand Analizi , Dosya Okuma

---

## 1. Yazılımın Geliştirilme Amacı

Bu yazılımın geliştirilme amacı bir projedeki operandların ve fonksiyonların analizinin oluşturulan bir kütüphane yardımıyla ekstra kod yazılmasına ihtiyaç duymadan yapılabilmesini sağlamak ve bu kütüphanenin birim ve entegrasyon testlerini gerçekleştirmektir.

## 2.Yazılımın Geliştirilmesi

Ödeve öncelikle üç ayrı sınıf oluşturarak başladım. Bu sınıflara; KodOku, KodTemizleme ve OperatorBul isimlerini verdim.

**KodOku Sınıfı:**

KodOku adındaki sınıf kendisine verilen dosya yolunu alıyor ve dosya içeriğini bir stringe dönüştürüyor. KodOku sınıfında dosya okuma fonksiyonu şu şekilde gerçekleştiriliyor;

```
String kod= kodOku.dosyaOku("deneme.java");
```

KodOku sınıfı aldığı dosya yolunu Scanner kullanarak okuyor ve her satır sonrasına \n ile bir newline ekliyor (Kod Temizleme sınıfını anlatırken açıklayacağım) ve okuduğu veriyi döndürüyor.

**KodTemizleme:**

KodTemizleme sınıfı yorumSil adlı bir fonksiyona sahip bu sınıf /* ifadesi ile */ ifadesi arasında bulunan bütün karakterleri ve // ifadesinden sonra satır sonuna kadar bulunan bütün ifadeleri kod dosyası içinden kaldırıyor. KodOku sınıfında her satırın sonuna \n eklememizin nedeni burada ortaya çıkıyor. Eğer dosyayı bütün olarak okuduğumuzda tek bir satır içine atamış olsaydık bu regex ifadesi // karakterlerinden sonrasında kalan bütün karakterleri yorum varsayacak ve kaldıracaktı bu yüzden önce yorumları sılıp ardından yine bu sınıfta regex ifadesi ile \r ve \n karakterlerini kaldırıyoruz.

KodTemizleme sınıfındaki yorum silme işlemi ise şu şekilde çağırılıyor;

```
String temizKod=kodTemizle.yorumSil(kod);
```

**OperatorBul:**

OperatorBul sınıfında regex ifadelerini tanımladım bu bölümde regex ifadelerini sayımlarda daha kolay hale gelecek şekilde ayarladım. regexMantiksal değişkeninde || , && ve ! işaretini içeren bütün parçaları seçen bir regex ifadesi yazdım bu durumda bu ifade != işaretini içeren parçayı da seçeceği için regexİliskiselTek ifadesini yazdım bu ifade sadece != operatorunu işaretlemek için yazılmıştı.

Bu işlemler bitince regexMantiksal ifadesinden elde edilen sonuçtan regexİliskiselTek sonucunu çıkardım ki böylelikle sonucum doğru olsun. Burada hesaplama ise regex operandları operatör ile birlikte alınacak şekilde yazıldı böylelikle birden fazla operatör ard arda kullanıldığında bunlar da

doğru bir şekilde hesaplanacaktı. Örneğin;

a+b+c+d ifadesi olduğunu düşünelim kodda bulunan ilk while ile a+b sayılıp RegexVariable ifadesi ile değiştirilecek ardından c+d sayılıp regexVariable ifadesi ile değiştirilecek ve sonrasında RegexVariable+RegexVariable ifadesi sayılacak ve bir bütün kabul edilecektir. Aşağıda OperatorBul sınıfının örnek kullanımı verilmiştir;

```
OperatorBul operatorBul = new OperatorBul();
System.out.println("Toplam Sayisal : "+operatorBul.sayisalOperatorSayisi(temizKod));
System.out.println("İkili Operator Sayisi : "+operatorBul.ikiliOperatorSayisi(temizKod));
System.out.println("Tek Sayisal : "+operatorBul.tekliOperatorSayisi(temizKod));
System.out.println("Toplam İliskisel : "+operatorBul.iliskiselOperatorSayisi(temizKod));
System.out.println("Toplam Mantiksal : "+operatorBul.mantiksalOperatorSayisi(temizKod));
System.out.println("Toplam Fonksiyon Sayisi : "+operatorBul.fonksiyonSayisi(temizKod));
System.out.println("Operand : "+operatorBul.operandSayisi(temizKod));
```

Bu kısımda bittikten sonra yazılımı direk düzenlemek yerine testlerini yazmaya başladım böylelikle testlerde hata aldıkça yazılımda sıkıntı çıkaran bölümleri bilecek ve orada test edecektim.

## 3.Yazılımın test edilmesi.

Öncelikle **JUnit** birim testleriyle başlayacaktım ama mantığını ve kullanım nedenini bilsem de test yazarken sorun yaşamamak için SABİS üzerinde paylaşılmış youtube videosunu izledim. Ardından KodTemizleme sınıfı için birim testi(bu sonradan mock nesnesi kullanılan bir teste döndü) yazmaya başladım. İlk olarak sürekli olarak string ifadeleri kopyalayıp testi karıştırmamak için TestStrings.java isimli bir dosyada Stringleri tuttum ki testlerde oradan string değerlerini çekebileyim. Test verisi olarak ilk önce geçen sene PDP ödevinde örnek verilen ve test edilen dosyaları kullandım. Ardından testler gerektirdikçe ekstradan test verileri ekledim örneğin ilişkisel ve mantıksal ünlemin ayrımı doğru yapılıyor mu anlamak için bir test verisi ekledim.

Ardından **parametreli** testlere başladım. Parametreli testleri yazarken dışarıdan çekilen bir String kullanmakta sorun yaşadım ve araştırma yaptım bu araştırmanın üstüne argümanları kendim oluşturarak parametreli testi gerçekledim. Bu testlerde parametre olarak TestString dosyasına verdiğim yorumsuz stringleri kullandım çünkü yorumları kaldırmak için KodTemizleme sınıfını kullanmam durumunda test birim testi olmaktan çıkıp entegrasyon testine dönüyordu. Örnek olarak bir parametreli testi göstermem gerekirse;

```
@ParameterizedTest
@MethodSource("tekSayisalTest")
void tekSayisalOperatorTestEt(String yorumsuzKod, int parametreSayisi) {
    OperatorBul operatorBul = new OperatorBul();
    assertEquals(parametreSayisi,operatorBul.tekSayisal(yorumsuzKod));
}
/*CSV uzun ve karmaşık stringlerde çok sorun çıkarttığı ve girdi olarakta değişkene atanmış stringi
 * kabul etmediği için argümanlı bir şekilde Parametreli testi gerçeklestirdim */
static Stream<Arguments> tekSayisalTest() {
    List<Arguments> tekSayisalTest = Arrays.asList(
            Arguments.of(TestStrings.yorumsuzTestKodDeneme,1),
            Arguments.of(TestStrings.yorumsuzTestKodOgrenci,3)
    );
    return tekSayisalTest.stream();
}
```

Bu bölümün ardından **tekrarlamalı** testlere geçtim, repeated testler benim için bu ödevde en rahat gerçekleme yaptığım bölüm oldu. Tekrarlamalı testten beklentimiz bir kodun sürekli tekrarlanması durumunda da şaşırmadan aynı sonucu vermesi olduğu için parametreli birim testini yaptığım bazı kodların tekrarlamalı testini yaptım. Tekrarlı testleri genelde beş tekrarlı olacak şekilde ayarladım;

```
@RepeatedTest(5)
@DisplayName("Tek sayisal Repeated Test. ")
void repatedTekSayisalTest() {
    OperatorBul operatorBul = new OperatorBul();
    assertEquals(3,operatorBul.tekSayisal(TestStrings.yorumsuzTestKodOgrenci));
}
```

Tekrarlamalı testlerin ardından **Faker** kütüphanesini kullanarak testler hazırlamaya başladım faker ile

bu yazılıma yönelik bir test verisi oluşturmakta zorlandım. Ayrıca faker kütüphanesiyle oluşturduğum girdiyi ayrı bir sınıfa koyup sonra test içinde çağırmam testin birim testliğini bozar mı emin olamadığım için her test için test edilecek kodu faker içinde yeniden oluşturdum. Burada faker ile oluşturduğum kod operatör ve operandlardan daha çok test edilecek kod içindeki verileri oluşturmak için kullanıldı. Örnek bir faker testi kodu göstermem gerekirse;

```
@Test
@DisplayName("Mantıksal Operator Faker Test")
void fakerMantiksalOperator() {
    OperatorBul operatorBul = new OperatorBul();
    Faker faker = new Faker();

    String kod = "public class " + faker.letterify("????").toUpperCase() + " {";
    kod += "private static final String USER = \"" + faker.name().fullName() + " " + faker.address().fullAddress() + "\";";
    kod += "public static void main(String[] args) {";
    kod += "int " + faker.letterify("?") + faker.letterify("?") + " = " + faker.random().nextInt(10) + ";";
    kod += "for (int i = 0; i < " + faker.random().nextInt(10) + "; i++) {";
    kod += "" + faker.letterify("?") + faker.letterify("?") + " += " + faker.random().nextInt(10) + ";";
    kod += "}";
    kod += " /* Bu kısım ''sayıların &&toplandığı alan*///Test||DataYorum";
    kod += "} String outPut=\"The user info is:\"+USER; ";
    kod += "}";
    assertEquals(0,operatorBul.mantiksalOperatorSayisi(kod));
    }
```

Faker Testlerinin ardından **Mock** kullanacağım testlere başladım. Burada Mock nesnesini genelde en başta birim test olarak oluşturduğum ve direk yorumsuz kod üzerinden işlem yapan testler yerine Mock nesneleri ile yorumsuz kodları döndüren bir mock nesnesi oluşturdum. Interfacelerde bazı eksiklerim olduğu için burayı gerçeklerken de biraz zorluk çektim. Örnek vermem gerekirse ;

```
@Test
@DisplayName("Tekli yorumları düzgün temizliyor mu?")//Burada kodu temizledikten sonra "//
void TekliYorumSatırı() {
    KodTemizleme kodTemizleme = new KodTemizleme();
    Mockito.when(kodOku.dosyaOku("Ogrenci.java")).thenReturn(TestStrings.testKodOgrenci);
    assertFalse(kodTemizleme.yorumSil(kodOku.dosyaOku("Ogrenci.java")).contains("//"));

}
```

Bunların ardından entegrasyon testlerine başladım genel olarak kodTemizleme ile OperatorBul içinde entegrasyon testlerini gerçekledim. Bazı bölümlerde ise parametreli testleri entegrasyon testleri ile birlikte yaptım.

## **4.**Zorlanılan kısımlar.

Ödevde en çok zorlandığım kısımlar yazılımı gerçekleme kısmıydı. Genel olarak yazılımı gerçekledikten sonra testlere başladım ve testlere başladığımda regexleri sadece x>y , x=y gibi ifadeleri eşleştirecek yani sadece değişkenlere arasındaki ifadeleri seçecek şekilde hazırladığımı fark ettim . Bu yüzden kodum x=2 , 2=x , x>2 ve 2>x gibi ifadelerin sayımını gerçekleştiremiyordu. Regex ifadelerimi yeniden düzenledim ve biraz önce bahsettiğim ifadeleri de dahil ettim. Ardından test yaparken bu seferde ifadelerimin fazladan operatör döndürdüklerini fark ettim. Yazdığım kod == , ++ , -- gibi ifadelerde operatörlerin birinci bölümlerini alıyordu yani y++ gibi bir ifade de regex ifadem y+ kısmını da dahil ettiği için elime fazladan operatör bilgisi geçiyordu bunu yine regexle ileri ve geri kontrol yaparak örneğin `((?<!\+)\+(?!\+))` gibi ifadeler ekleyerek fazladan operatör sayımını engelledim. Operand sayımında bulunan operatörler üzerinden gittiğim için Ogrenci dosyası kullanıldığında operatör sayısını yanlış veriyordu bu kısmı düzeltemedim.

KodOku sınıfında bir tane parametreli test yazdım bu test başarılıydı. KodTemizleme sınıfında 2 repeated, 1 entegrasyon, 2 faker , 1 parametreli ve 1 mockito nesnesi ile test yazdım bu 7 testte başarılı çalışıyordu. OperatorBul sınıfında ise 2 mockito, 5 repeated, 3 faker , 1 unit, 7 parametreli , 2 entegrasyon ve 1 tane parametreli entegrasyon testi yazdım. Bu testlerden 1'i başarısız oldu. Bun işlem ogrenci dosyasındaki operand sayısını bulma işlemiydi.