BİM539, Ders 9:
Birim Test, Bölüm 2

Bu metodu nasıl test edersiniz?

```
public class Example {
    public static int doubleMe(int x) {
        return x * 2;
    }
}
```

```
// Şöyle bir şey olabilir…
@Test
public void zeroTest() {
   assertEquals(Example.doubleMe(0), 0);
}
@Test
public void positiveTest() {
   assertEquals(Example.doubleMe(10), 20);
}
@Test
public void negativeTest() {
   assertEquals(Example.doubleMe(-4), -8);
}
```

Peki bunu nasıl test edersiniz?

```
public class Example {
  public void quackAlot(Duck d, int num) {
    for (int j=0; j < num; j++) {
       d.quack();
    }
  }
}
```

1. Neresi test edilmeli? Test edilecek bir değer yok!
   - Davranışı test et: `quack()`, `num` kere çağırıldı mı?
2. `Example` sınıfını `Duck` sınıfı olmadan nasıl test ederiz?
   - `Duck` henüz oluşturulmamış olabilir
   - Oluşturulduysa bile `Duck` kodunu test etmek istemeyebiliriz
   - `Duck` için gerçeğini taklit eden bir "test kopyası" kullanın

İleri Birim Test Teknikleri

- Sınıf Bağımlılıklarını Kaldırma
  - Test Kopyaları (Test Doubles)
  - Stubs
- Davranış Doğrulama
  - Mocks

İleri Birim Test Teknikleri

- Sınıf Bağımlılıklarını Kaldırma
  - Test Kopyaları (Test Doubles)
  - Stubs
- Davranış Doğrulama
  - Mocks

Test Kopyaları

- Testte kullanabileceğiniz “Fake” objedir
- Nasıl isterseniz öyle davranış sergileyebilirler. Tam olarak gerçek gibi davranmak zorunda değiller.

Test Kopyası Örnekleri

1. Kopya veritabanı bağlantısı
   - Kopya aslında veritabanına bağlanmaz
   - Kopya önceden tanımlı veritabanı girdilerini test için döndürür
2. Kopya dosya nesnesi
   - Kopya aslında bir dosya açmaz
   - Kopya test etmek için dosya okuma hatasını simüle eder
3. Kopya RandomNumberGenerator
   - Aslında gerçek rastgele bir sayı üretmez
   - Daha önceden belirlenmiş değerleri döner ki test tekrarlanabilir olsun

Kopyalanmış Bağlı Sınıf (Test Edilmeyen Sınıf)

- Sınıfların kopyalanmış bağlı nesneleri, test edilecek olan sınıfın bağımlı olduğu sınıflardır
  – Bu sınıfların kopyalanma sebebi test edilecek sınıfın bağımlı olduğu sınıfları test etmeyi istememizdir.

- Test edilecek sınıfı kopyalamayın!

- Eğer bu sınıfı kopyalarsanız, neyi test edeceksiniz? ☺

Test Kopya Örneği

```
@Test
public void testDeleteFrontOneItem() {
    LinkedList<Integer> ll = new LinkedList<Integer>();
    ll.addToFront(Mockito.mock(Node.class));
    ll.deleteFront();
    assertEquals(ll.getFront(), null);
}
```

- `LinkedList` test etmek istiyoruz bu yüzden `Node`'u test etmek istemeyiz
- Kopya `Node` JUnit `mock` API ile gerçekleştirilir

İleri Birim Test Teknikleri

- Sınıf Bağımlılıklarını Kaldırma
  - Test Kopyaları (Test Doubles)
  - Stubs
- Davranış Doğrulama
  - Mocks

Stubs

- Kopyalar “sahte nesnelerdir”
- Stubs ise “sahte nesnelerin” “sahte metotlarıdır”

Stubs

- Stub metotlar:
  - “Metodu çağırmak yerine neye ihtiyaç duyarsanız onu verir“
- "Neye ihtiyaç duyarsanız" kısmı return değeridir
  - Gerçek metot çalıştırılmaz

Stub Örneği

```
public int quackAlot(Duck d, int num) {
   int numQuacks = 0;
   for (int j=0; j < num; j++) {
     numQuacks += d.quack();
   }
   return numQuacks;
}
```

- `quack()` metodunu `Duck` bağımlılığını kaldırarak stub metot yapmak istiyoruz

Bir test kopyası ve stub method oluştur

```
@Test
public void testQuackAlot() {
    Duck mockDuck = mock(Duck.class);
    when(mockDuck.quack()).thenReturn(1);
    int val = quackAlot(mockDuck, 100);
    assertEquals(val, 100);
}
```

Böylelikle test yerelleştirilmiş oldu

- Yalnızca quackAlot() metoduna odaklanmış olduk
  - Duck.quack() metodu veya Duck sınıfının çalışıp çalışmadığı ile ilgilenmedik
  - Duck.quack() ise Duck sınıfına ait ayrı bir birim testte test edilir

- Birim testler yalnızca en küçük birimleri test etmelidir
  - Aksi taktirde testler BRITTLE (breaks easily due to external changes, harici değişiklikler sonucunda kolayca bozulur) olur
  - Hata durumunda hatanın nokta atış kaynağını bulmak oldukça zorlaşır.

Stub metot oluşturulmazsa ne olur?

- Kopya metoda ait bir stub oluşturulmazsa…
  - Yine de orijinal metot çalışmaz
  - Varsayılan değerler döner
    - Ör. Boolean dönüyorsa değer: false
    - Ör. Int dönüyorsa değer: 0
    - Ör. referans dönüyorsa: null
- `void` ise ne olur?
  - Burada stub'a ihtiyaç yoktur çünkü bir return değer yoktur

İleri Birim Test Teknikleri

- Sınıf Bağımlılıklarını Kaldırma
  - Test Kopyaları (Test Doubles)
  - Stubs
- Davranış Doğrulama
  - Mocks

Problemli Örnek

```
public class Example {
   public void quackAlot(Duck d, int num) {
      for (int j=0; j < num; j++) {
         d.quack();
      }
   }
}
```

1. Test nerede başlar, ne test edilir? Test edilecek bir değer yok!
   - Davranışı test et: `quack()`, `num` kere çalıştırılmalı
2. Example sınıfını Duck sınıfı olmadan nasıl test ederiz?

Davranış Doğrulama

- Buradaki doğrulamanın, doğrulama ve onaylama ile ilgisi yoktur.
- Davranış doğrulama vs. Durum Doğrulama
  - Durum Doğrulama: Programın durumunu test et
    - Metot çağrısı veya çağrıları sonucunda programın durumu doğru değişiyor mu?
    - Şimdiye kadarki yaptığımız şey de buydu
  - Davranış Onaylama: Kodun davranışını test et
    - Belli bir metot belirli bir sayıda çağırıldı mı?
    - Metot doğru parametrelerle mi çağırıldı
    - Bu doğrulama Mockito ile yapılabilir

Mock

- Mock: Davranış onaylamada kullanılmak üzere oluşturulan test kopyalarıdır

- Çok fazla framework bulunmaktadır (Mockito en ünlüsüdür ve bunu kullanacağız). Mock ve kopyalar arasında bir fark bulunmamaktadır
- Teknik olarak bir mock test kopyasının spesifik bir türüdür

Mock Örneği

```
@Test
public void testQuackAlot() {
   // Duck kopyası oluştur, quack()
   metodunu stub yap
   Duck mockDuck = mock(Duck.class);
   mockDuck.when(mockDuck.quack()).thenReturn(1);
   // mockDuck.quack()'u 5 kere çağıran quackAlot'u çağır
   quackAlot(mockDuck, 5);
   // quack 5 kere çağırıldı mı kontrol et
   Mockito.verify(mockDuck, times(5)).quack();
}
```

Metot Stub yapmak için çok karmaşık ise?

```
public class Duck {
  boolean alive = true;
  public void shoot() {
    boolean hit = …; /* karmaşık bir yörünge hesabı */
    if( hit ) alive = false;
  }
  public String toString() {
    return alive ? “alive” : “dead”;
  }
}
public DuckHunt {
  public void shootDuck(Duck d) {
    System.out.println(d.toString()); //  Beklenen return “alive”
    d.shoot();
    System.out.println(d.toString()); //  Beklenen return “dead”
  }
}
```

Bir Fake Oluşturun

```
public class FakeDuck extends Duck {
  // Karmaşık yörünge hesabı yapma
  public boolean shoot() { alive = false; }
}
@Test
public void testShootDuck() {
  // fake Duck sınıfı oluştur
  Duck fakeDuck = new FakeDuck();
  // shootDuck'ı çağır
  shootDuck(fakeDuck);
}
```

- Fake: Orijinal versiyonun daha basitleştirilmiş halidir

İyi bir Birim Test neye benzer?

- Her koşmada aynı sonucu almalı
- Diğer testlerden bağımsız olmalı
- Tek seferde tek bir test senaryosu test etmeli
- Yerelleştirilmeli (En küçük birimi test etmeli)

İyi Birim Test:
Her koşmada aynı sonucu almalı

- Test her çalıştırılmada başarılı veya başarısız olmalı. Neden?
  - Aksi takdirde, hangi derleme veya versiyonun kusura sebep olduğu bilinemez
  - Kusur çok daha önce ortaya çıkmış olabilir ve bu güne kadar şans yardımıyla tespit edilememiş olabilir
- Yani…
  - Test senaryoları çalıştırılmadan önce bütün ön koşullar doğru ayarlanmalı
  - Testler çalıştırılırken rastgelelik olmamalı
    - Testin içinde rastgelelik olmamalı (Ör. Rastgele girdi değeri verilmemeli)
    - Programın içinde rastgelelik bulunmamalı (Ör. Zar oyunu)
- Program içindeki rastgelelik nasıl kaldırılır?!
  - Test edilebilir kod yazma dersinde ayrıntılarına bakacağız ☺

İyi Birim Test:
Diğer testlerden bağımsız olmalı

- Bir test diğer testlerden etkilenmemelidir. Neden?

- Bir test paketindeki testlerin bir kısmını çalıştırmayı tercih edebiliriz
  - Eğer bir test seçilmemiş bir başka teste bağlı ise doğru çalışmayacaktır…

- Testleri farklı sıra ile çalıştırmayı tercih edebiliriz
  - Hatta bunları paralel çalıştırmayı tercih edebiliriz
  - Pek çok birim test framework'ü hızlı tamamlanması için paralel çalıştırmayı destekler

- Hata durumunda bile testlerin tamamlanmasına izin verilir
  - Bir test hataya düşerse diğer testleri etkilememelidir

İyi Birim Test:
Her seferinde yalnızca bir şeyi test eder

- Tek bir testte farklı test senaryoları bulunmamalı. Neden?
  - Test senaryosu hata alırsa kalan testler test edilemez
  - Test hatasında hangi testin hata verdiğini tespit etmek zordur

- Yani test içerisinde yalnızca bir metot çağırın
  - Bu metot test ettiğiniz metottur.

- Eğer test içerisinde bir “if..else” varsa bu kötü koddur (code smell)!
  - Bir şeyden dönen değere göre hangi metodun çağırılacağı hiç de uygun bir test yaklaşımı değildir.

İyi Birim Test:
Yerelleştirilmeli (Testler yalnızca en küçük birimi test etmeli)

- Yalnızca birimi test et başka bir şeyi değil. Neden?
  - Testte hata meydana gelirse, birim test ile kusuru anlayabilirsin
  - Aksi olursa kusur dahil ettiğin bir başka kodda olabilir
    ☛Kusurlar birim testin amacıdır

- Bir birim (metot) bir başka metoda bağımlı ise ne olacak?
  - Test kopyaları ve stub'ları kullan

JUnit kullanılabilir tek birim test framework'ü değildir!

- Java için bile!
- Ancak xUnit framework'leri daha yaygındır ve kullanımı daha kolaydır
  - C++: CPPunit
  - JavaScript: JSUnit
  - PHP: PHPUnit
  - Python: PyUnit

Tavsiyeler

- Testleri mümkün olan en kısa sürede yaz.
  - İdealde, testler kodlamaya başlamadan yazılır (sonraki derste TDD kısmında inceleyeceğiz).
- Başkalarının test etmesini kolaylaştıracak şekilde geliştirme yapın.
  - Ör. Eğer harici objeleri metot içerisinde oluşturursanız fake oluşturması da zorlaşır

```
void addCat() {
  Cat cat = new Cat(1, “cat”); // Bunun fake’i nasıl oluşturulur?
  list.add(cat);
}
```

  - Sonraki ders olan “Test Edilebilir Kod Yazma” dersinde inceleyeceğiz
- Eski sistemlerde ilerledikçe test ekleyin, bataklığa düşmeyin ☺

Birim Test != Sistem Testi

- Teslimat 1 için yaptığın manuel testler sistem testidir
  - Bütün sistemin çalışıp çalışmadığı kontrol edilir
- Teslimat 2 için otomatize olarak yaptığın testler birim testleridir
  - Her bir birimin fonksiyonelliğini bireysel olarak test edersiniz
- Uygun test süreci her ikisini de barındırır
  - Birim testler bir kod parçasının lokal hatalarını ortaya çıkarır
  - Sistem testi kodun bütün parçalarının bir arada doğru çalışıp çalışmadığını test eder

Textbook Chapter 14'ü okuyun

- Ayrıca, Mockito kullanımı için JUnit örneğini inceleyin: SoftwareTesting/junit_example

•Mockito Kullanım Kılavuzu:
https://javadoc.io/static/org.mockito/mockito-core/3.2.4/org/mockito/Mockito.html