BİM593, Ders 10: Test-Driven Development

Eski Karanlık Çağlarda...

Peki bu günlerde...

- Bu tür durumlardan sakınma konusunda testlerin ne kadar önemli olduğunu biliyoruz
- Kod kalitesi geliştiriciler de dahil olmak üzere herkesin sorumluluğundadır
- Geliştiriciler test yazarlar (genellikle birim test)

Test-Driven Development

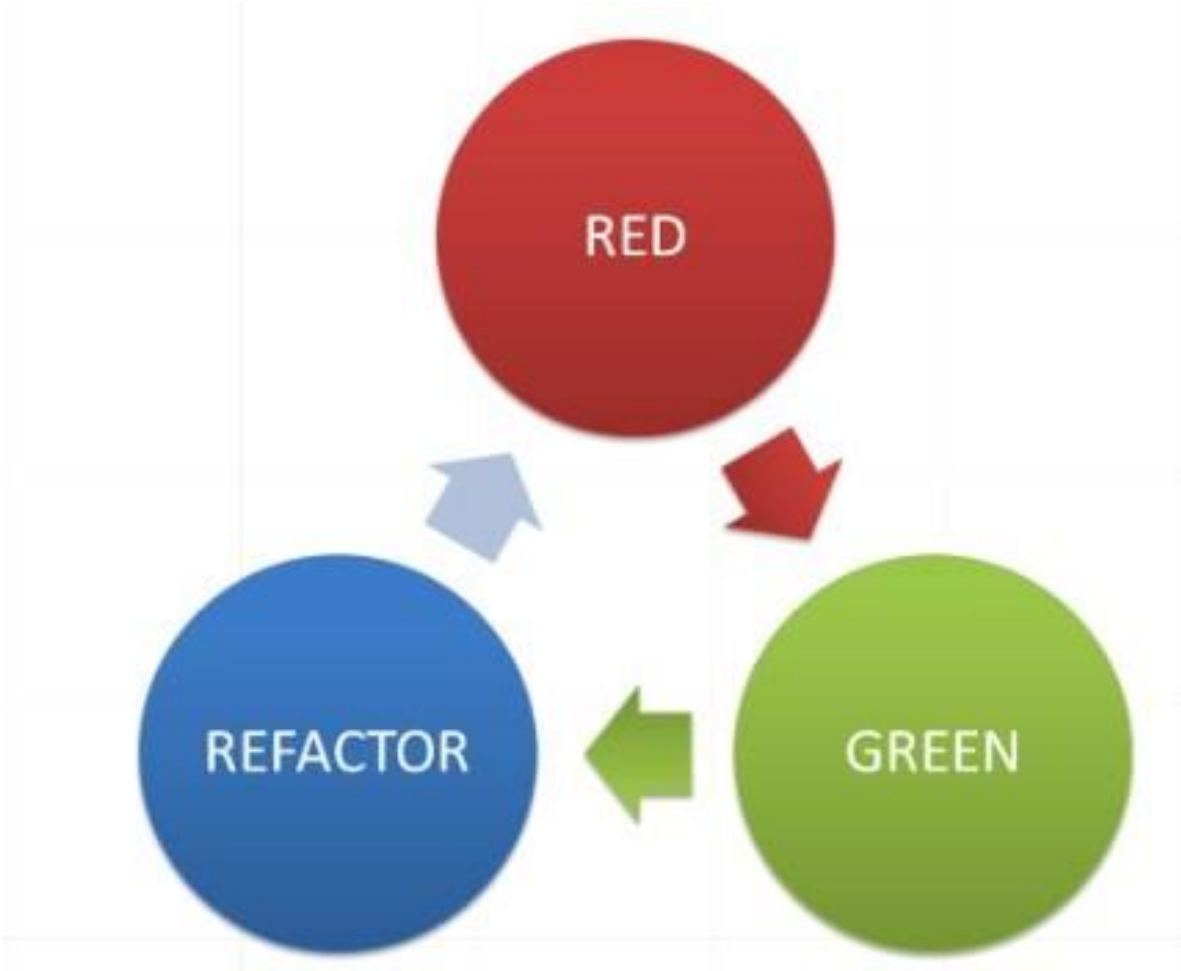
- Çok fazla test edilen yüksek kaliteli yazılım geliştirme stratejisidir
- Hala gelişmekte olan yazılım geliştirme dünyasının bir parçasıdır ☺

Yani TDD nedir?

Aşağıdaki özellikleri barındıran yazılım geliştirme stratejisidir:

1. Kod yazmadan önce test yazmadır
2. Sadece test edilen kodun yazılmasıdır
3. Sadece kodu test eden testler yazmadır
4. Çok kısa geri dönüş süresidir
5. Erken ve sık refactor'dür

Red-Green-Refactor Döngüsü



The Red-Green-Refactor Döngüsü

- Red – Yeni bir fonksiyon için test yaz
  - Test hata veriyor olmalıdır! (Bu yüzden Red)
- Green – Yalnızca testi geçecek kodu yaz
  - Şimdi test geçmeli. (Bu yüzden Green)
- Refactor – Kodu tekrardan gözden geçir ve daha iyi hale getir

TDD ile Fizzbuzz

- Gereksinimler:

Uygulama sayıları 1’den 100’e kadar ekrana yazdırmalıdır

Eğer sayı 3’e bölünebilir ise uygulama "Fizz" yazmalıdır.

Eğer sayı 5’e bölünebilir ise uygulama "Buzz" yazmalıdır.

Eğer sayı 3 ve 5’e bölünebilir ise uygulama "FizzBuzz" yazmalıdır.

- Örnek Çıktı: 1 2 Fizz 4 Buzz Fizz 7 8 Fizz Buzz ...

Red - Yeni test ekleyerek başla

```
@Test
public void testNumber() {
  assertEquals(_fb.value(1), "1");
 }
// Kod
public String value(int n) {
  return "";
}
```

Hata veren...


WELCOME
TO
FAIL
POPULATION: YOU

Green - Testi geçecek kod yaz

```
@Test
public void testNumber() {
  assertEquals(_fb.value(1), "1");
 }
// Code
public String value(int n) {
  return "1";
}
```

Refactor yapacak bir şey yok. Sıra sonraki testte!

Red - Bir başka test ekleyelim

```
@Test
public void testNumber2() {
   assertEquals(_fb.value(2), "2");
 }
// Code
public String value(int n) {
  return "1";
}
```


WELCOME
TO
FAIL
POPULATION: YOU

Green - Bir başka değişiklik yapalım

```
 public String value(int n) {
  if (n == 1) {
   return "1";
  } else {
   return "2";
  }
}
```

Daha iyi olabilir!

Refactor - Böyle daha iyi, testler hala geçerli!

```
public String value(int n) {
   return String.valueOf(n);
}
```

Red - Bir başka test ekle- hata vermeli

```
@Test
public void testNumber3() {
  assertEquals(_fb.value(3), "Fizz");
}
```

Green - Yeni koda ihtiyacımız var!

```
private boolean fizzy(int n) {
   return (n % 3 == 0);
}

public String value(int n) {
   if (fizzy(n)) {
    return "Fizz";
   } else {
    return String.valueOf(n);
   }
}
```


YESSS!!!
quickmeme.com

Refactor - Burada yapacak bir şey yok

```
private boolean fizzy(int n) {
   return (n % 3 == 0);
}

public String value(int n) {
   if (fizzy(n)) {
    return "Fizz";
   } else {
    return String.valueOf(n);
   }
}
```

Red - Bir başka test ekleyelim

```
@Test
public void testNumber5() {
  assertEquals(_fb.value(5), "Buzz");
}
```

… ve tabii hata verecektir

Green - buzzy(n) metodunu ekleyelim

```
private boolean buzzy(int n) {
    return (n % 5 == 0);
}
public String value(int n) {
    if (fizzy(n)) {
      return "Fizz";
    } else if (buzzy(n)) {
      return "Buzz";
    } else {
      return String.valueOf(n);
    }
}
```

Red - Son denklik sınıfı

```
@Test
public void testNumber15() {
  assertEquals(_fb.value(15), "FizzBuzz");
}
```

ve hata verecek...

Green - value() metodunu değiştir

```
public String value(int n) {
   if (fizzy(n) && buzzy(n)) {
   return "FizzBuzz";
   } else if  (fizzy(n)) {
    return "Fizz";
   } else if (buzzy(n)) {
    return "Buzz";
   } else {
    return String.valueOf(n);
   }
}
```

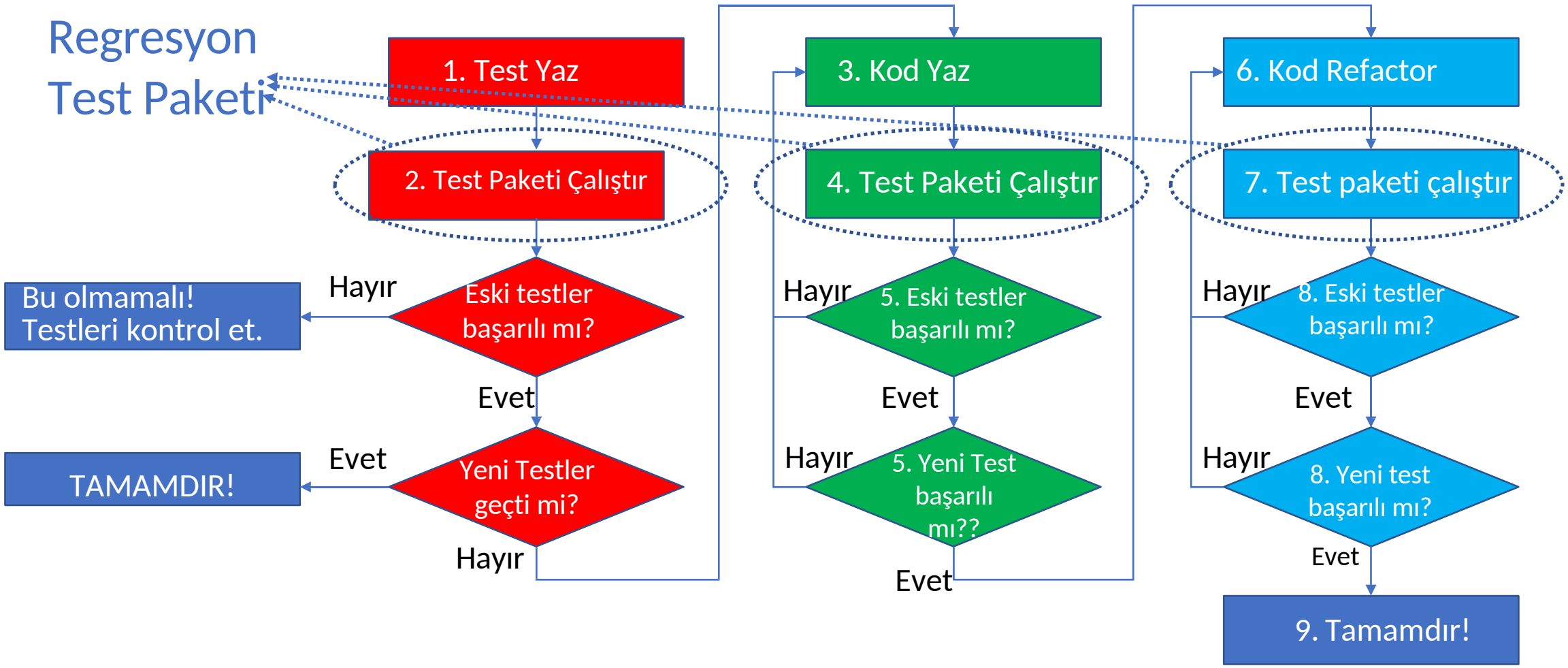
Refactor - Yapacak bir şey yok

```
public String value(int n) {
   if (fizzy(n) && buzzy(n)) {
   return "FizzBuzz";
   } else if  (fizzy(n)) {
    return "Fizz";
   } else if (buzzy(n)) {
    return "Buzz";
   } else {
    return String.valueOf(n);
   }
}
```

Sonuç?

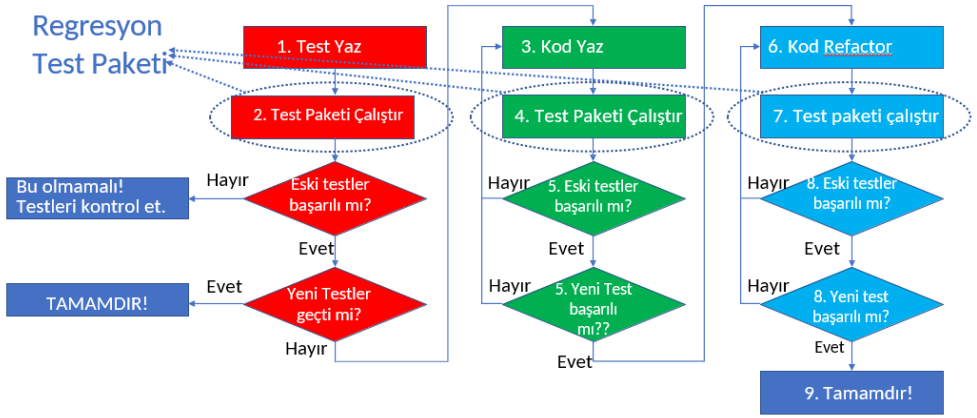
- Artık çalışan ve test edilmiş bir FizzBuzz uygulamasına sahibiz
- Tüm denklik sınıfları için otomatik testlere sahibiz
- Geliştirme sürecinin her noktasında ileriye giden bir yol izledik

Red-Green-Refactor Döngüsü Akış Şeması



Red-Green-Refactor Döngüsü Akış Şeması

- Her seferinde küçük bir test yaz
  - Geri dönüş döngüsünü kısa tut
- Bütün test paketini çalıştır
  - Regresyon hatalarına dikkat edin
  - Tüm bileşenlerin sürekli test edilmesini sağlar
- Yalnızca testi geçecek yeterli testi yazın
  - Test edilmemiş bölgeye girme isteğine karşı koyun ☺
  - Yazmış olduğun tüm kodun ve önceki kodların sürekli kapsandığından emin olun
- Her iterasyon sonunda refactor yapın.
  - Tekrar düzenleme şansı olacağından doğruya erişmeye odaklanmayı sağlar.
  - Refactor sonrası regresyon testi yapmayı unutmayın.



Refactor neleri içerir?

- Muhtemelen yazdığın ilk kod mükemmel olmayacaktır
  - Kötü algoritma seçimi
  - Kötü değişken isimlendirme
  - Kötü performans
  - Kötü dokümantasyon
  - Kolay anlaşılır olmaması
  - Genel kötü tasarım
- Refactor yapmadan önce çalışan bir sürümünüz olduğunu hatırlayın
  - Doğru çalışması kodun iyi görünmesinden daha önemlidir

Anahtar: Test etmediğiniz kodu yazmayın

- *YAGNI* - You Ain't Gonna Need It
  - Eğer test etmiyorsan ona şu anda ihtiyacın yoktur
  - Eğer ona şu anda ihtiyacın yoksa, gelecekte de ihtiyacınız olmayacak.
- *KISS* - Keep It Simple, Smarty-pants
  - Çok karmaşık, zekice ve yüksek mühendislik gerektiren kodlar yazma
  - Yalnızca anlaşılması ve daha sonra değiştirilmesi zor hale getirir
  - “Premature optimization is the root of all evil” –Donald Knuth
- Fake It ‘til You Make It
  - Testi geçmek için kesinlikle gerekli olmayan bir şeyi uygulamaya saplanıp kalmayın

Yapana kadar taklit et

- mocks/stubs uygula
- Ancak ufak seviyede fonksiyonellik uygulayabilirsin

Test:

```
assertEquals(sqrt(4), 2);
```

Kod:

```
public void sqrt(int n) {
    return 2;
}
```

Birim Testleri Hızlı ve Bağımsız Yap

- Her küçük değişiklikte bütün test paketinin çalıştırıldığının farkında mısınız?
  - Yani test süresi geliştirme sürecinde oldukça fazla yer kaplamaktadır
- Uzun test gecikmelerini nasıl engelleyebiliriz?
- Hızlı: Her bir bireysel birim testin çalışmasını hızlandır
  - Gecikmeye yatkın test bileşenlerini kopya ve stub kullanarak test edin (Ör. Veritabanları, dosyalar, network I/O)
- Bağımsız: Testlerin bir başka testin sonucuna bağlı olmamasını sağlayın
  - Bu geliştiricinin yalnızca değiştirdiği kodun testini dilediği gibi çalıştırmasına olanak tanır
  - Testlerin birbirinden ayrılıp paralel çalışmasını sağlar

TDD'ın Faydaları

- Testlerin konu ile alakalı olmasını sağlar
  - Testler tam olarak uyguladığımız işlevsellik için yazılmıştır
  - Gelecekte uygulama ihtimaliniz olan hayali işlevsellikleri test etmezsiniz
- Kodun konu ile alakalı olmasını sağlar
  - Kod birim testlerle ifade edilen kullanıcı ihtiyacının ötesine geçemez
- Küçük adımlar atmanızı sağlar
  - Deneyimli programcılar küçük adımların hataları yerelleştirmeye yardımcı olduğunu bilir
- Otomatik olarak sunulan büyük test paketi (en azından öyle hissedilir ☺)
  - Her zaman %100 kod kapsamı sayesinde regresyon hatalarının önlenmesine yardımcı olur
- Sürekli test edilerek koda olan güveni artırır

TDD'ın dezavantajları

- Test projenin ana giderlerinden biri olur
  - Bakım açısından: Özellikle çok fazla mock varsa
  - Geliştirme döngüsü açısından: Tam bir RGR döngüsü test sürecini yavaşlatır (Testi hızlandırmak için çok fazla uğraşırsanız bakım maliyeti de artar)
- Karmaşık mimari tasarımlar ile büyük projeler yapmak zorlaşır
  - TDD sizi dar bir açıda tutar ve uzun vadeli iyi tasarımlar yapmanızı engeller
  - Bazı şeyler küçük adımlarla gerçekleştirilemez
- Birim testlere odaklanmak testin diğer yönlerinin kısa sürede kaybolması anlamına gelebilir
  - Birim testler mock kullanım yoluyla entegrasyon testini aktif olarak önler
  - Birim test bir modülün veya sistemin hiçbir zaman bütünüyle test edilmediği anlamına gelir
  - Birim test kod merkezlidir ve son kullanıcı girdilerini içermez

TDD = Bir tür önce test geliştirme tekniğidir

- Temel fikir, kodlamadan önce ilk başta beklenen davranışı düşünmektir.
- Programın ne yapması gerektiğini çözmek gerekir (gereksinimler!)
- Gereksinimler doğrultusunda testler yazılır
- Test doğrultusunda kod yazılır

* Not: ATDD (Acceptance Test Driven Development) ve BDD (Behavior Driven Development) gibi farklı önce test geliştirme teknikleri bulunmaktadır.

Textbook Chapter 15'i okuyunuz

- "TDD is dead. Long live testing." - David Heinemeier Hansson: https://dhh.dk/2014/tdd-is-dead-long-live-testing.html