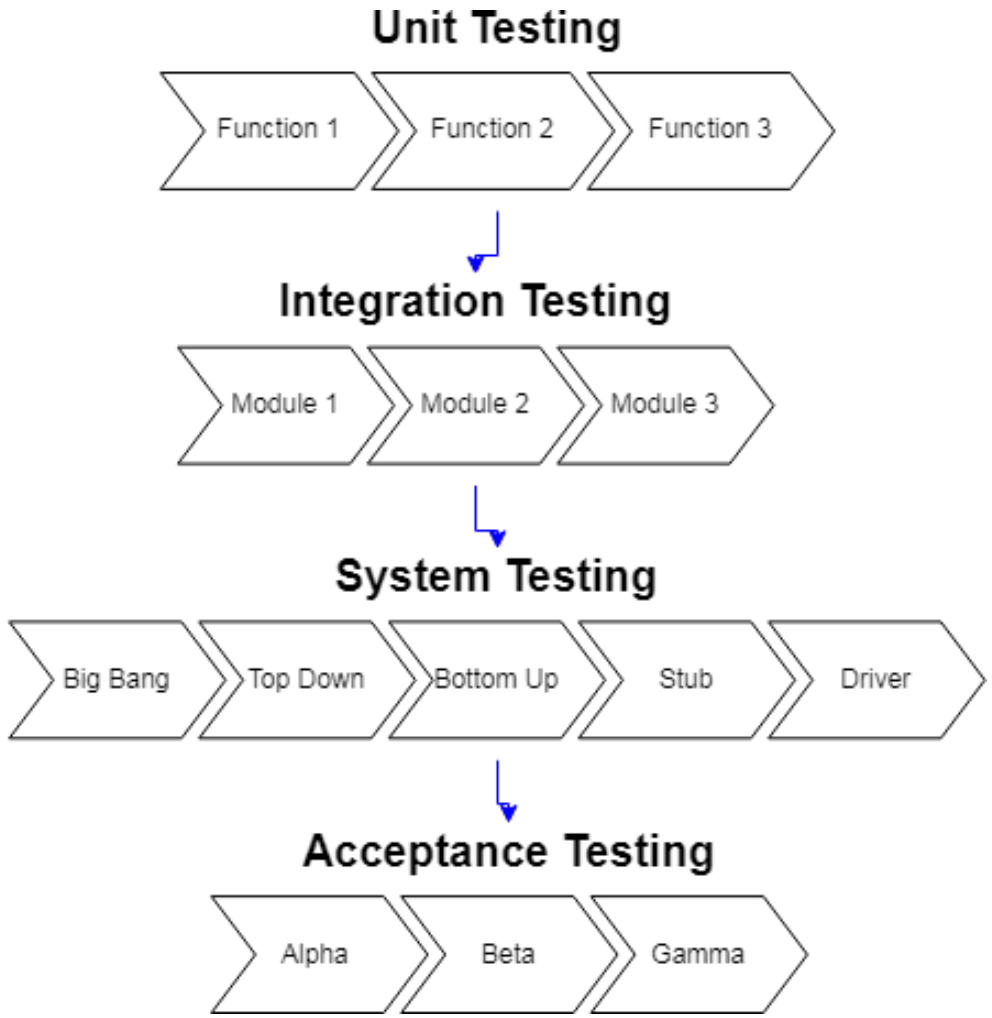
BİM539, Ders 8:
Birim Test, Bölüm 1

Birim Test nedir?

- Birim Test: Kodun en küçük birimini test etmek
  - Fonksiyonlar, metotlar veya sınıflar
  - Fonksiyon veya metodu doğrudan tetikler
- Saydam kutu testi için gereklidir
- Hedef: Kodun doğru çalıştığından emin olmak
  - Sistemin tamamının doğru çalışıp çalışmadığını test etmez
  - Oldukça yereldir

Yazılım Testinin Dört Aşaması



Daha Fazla
Saydam Kutu

Daha Fazla
Kara Kutu

Örnekler

- sort() metodunun testi sonucunda bu metodun girdileri sıralaması beklenir
- formatNumber() metodu testi sonucunda sayıları doğru bir şekilde formatlaması beklenir
- Integer değer bekleyen bir programa string değer gönderilmesi durumunda programın çökmemesi beklenir
- Null referans gönderildiğinde exception yakalanması gerekir
- send() ve receive() metotlarının bir sınıfta tanımlı olması beklenir

Birim testi kim yapar?

- Genellikle kodu yazan geliştirici tarafından yapılır
- Bir diğer geliştirici tarafından da yapılabilir
- (Oldukça nadiren), bir saydam kutu tester'ı tarafından yapılır.

Amaç Nedir?

1. Problemi erkenden tespit etme
2. Hızlı geri dönüş (turnaround) süresi
3. Geliştiricinin kodundaki problemleri anlaması
4. "Yaşayan dokümantasyon“
5. Toplamda birim testler bir test paketini oluşturur
   - Bütün test paketini çalıştırmak kodun değişmesi ile oluşacak yerel olmayan etkilerle ortaya çıkan kusurların hızlıca ortaya çıkmasını sağlar

Birim test nelerden oluşur?

- Birim testler aslında birimler düzeyinde test durumlarıdır
  - Aynı Bileşenler: Ön koşullar, çalıştırma adımları, son koşullar, …
- Birim testinin anatomisi (ör. JUnit kullanıldığında):
  - Ön koşullar: Ayarlamalar (değişken/veri yapıları başlatma, …)
  - Çalıştırma adımları: Bir veya daha fazla birim testi yapılacak metodun çağrısı
  - Son koşullar: assertion (son koşulların karşılanıp karşılanmadığının kontrolü)
  - (Opsiyonel) tear down kod (bir sonraki test için şartları sıfırlama)

LinkedList.equals()metodu için birim test

- Ön koşullar:
  - Tek bir node bulunan iki bağlı liste
  - Node'lar integer 1 değeri içerir

- Çalıştırma adımları: İki listeyi `equals()` metodu ile karşılaştır

- Son koşullar: Sonuç true olmalıdır

Birim test senaryosunun JUnit implementasyonu

```
// Tek dugumlu bir bagli listenin ayni dugum değerlerine
sahip olduğunun kontrolu
@Test
public void testEqualsOneNodeSameVals() {
   LinkedList<Integer> list1 = new LinkedList<Integer>();
   LinkedList<Integer> list2 = new LinkedList<Integer>();
   list1.addToFront(new Node<Integer>(new Integer(1)));
   list2.addToFront(new Node<Integer>(new Integer(1)));
   assertEquals(list1, list2);
}
```

- `assertEquals`: `equals()` metodunu parametrelerle çağırır ve eğer iddia edilen şey doğru ise true döner

Daha fazla test örnekleri için:

SoftwareTesting\junit_example

Assertions = Son Koşul Kontrolü

- Olan ve olması gereken değerlen incelendiğinde…
  - Birim testte BEKLENEN DEĞER ve SON KOŞUL'a karşılık gelmektedir
- Metot veya metotları çağırarak bir test çalıştırıldığında…
  - GÖZLENEN DAVRANIŞ elde edilir
  - Bu beklenen değer metodun return değeri veya metoda ait bir side-effect olabilir

- GÖZLENEN DAVRANIŞ == BEKLENEN DAVRANIŞ olmak zorundadır!

JUnit assertion'ları

- JUnit'teki bazı assertion'lar:
  - `assertEquals, assertArrayEquals, assertSame, assertNotSame, assertTrue, assertFalse, assertNull, assertNotNull, assertThat(*something*), fail(), ...`
- assertSame(Object expected, Object actual): referans karşılaştırma
  - İki referansı equals yerine == ile karşılaştırır
- assertThat(T actual, Matcher<T> matcher): Her şeyi kapsar
  - Ör. assertThat("BİM539", anyOf(is("bim539"), containsString("BİM")));
- fail() : her zaman hata veren assertion'lar için
  - Neden her zaman başarısızlıkla sonuçlanan bir test senaryosu bulunsun isteyesiniz ki?
  - Belki de kodun o kısmının çalıştırılmaması gerekiyor olabilir

fail() örnek

```
//  addToFront()'a null değer verince
IllegalArgumentException fırlatır

@Test
public void testAddNullToNoItemLL() {
   LinkedList<Integer> ll = new LinkedList<Integer>();
   try {
      ll.addToFront(null);
      fail("Burada hata fırlatmalıydı");
   } catch (IllegalArgumentException e) {
   }
}
```

- Kod çalıştırılınca asla fail'e düşmez

Daha fazla assertion için:

- JUnit Javadoc:
  - http://junit.sourceforge.net/javadoc/org/junit/Assert.html

Hangi değerler test edilmeli?

- İdealde...
  - Her bir denklik sınıfı
  - Sınır değerler
  - Edge durumlar
- Ve ayrıca hata mod'ları
  - Hata Modu: Hata vermesi beklenen girdiler
  - Olmadı gereken yerde başarısız olmak da gereksinimin bir parçası olabilir

Denklik Sınıfı / Sınır Değerler / Hata Modu

```
public int quack(int n) throws Exception {
  if (n > 0 && n < 10) {
    return 1;
  } else if (n >= 10) {
    throw new Exception("too many quacks");
  } else {   // n <= 0
    throw new Exception(“too little quacks");
  }
}
```

Denklik sınıfları: {..., -2, -1, 0}, {1, 2, ..., 9}, {10, 11, 12, ...}

Sınır değerleri: 0, 1, 9, 10

Hata modları: {..., -2, -1, 0} + {10, 11, 12, ...}

Public vs. Private Metotlar

- İki Yaklaşım:
  - Yalnızca public metotları test et
  - Bütün metotları test et - public ve private
- Yalnızca public metotları test et
  - Private metotlar daha fazla sıklıkla eklenir/çıkarılır/değiştirilir
    - Eğer private metotları test edersek, her kod değişiminde testleri de değiştirmemiz gerekir!
  - Private metotlar genelde public metotların bir parçası olarak test edilirler
  - Private metotların testi dil/framework sebebiyle zor olabilir

Public vs. Private Metotlar

- Bütün metotları test et – public ve private
  - Public/private ayrımı önemsizdir – Bütün metotların doğru olması istenmektedir.
  - Birim testin anlamı en küçük birimin test edilmesidir;
    Private metotların testi ise en küçük birimin ruhuna daha yakındır
- Hangi yaklaşım seçilmeli?
  - Her şey yazılım QA ile ilgilidir ve değişkenlik gösterir ☺

Yalnızca public metot testinin yeterli olduğu durum:

```
class Bird {
   public int chirpify(int n) {
     return nirpify(n) + noogiefy(n + 1);
   }
   // Cagirilan metotlar:
   private int nirpify(int n) { … }
   private int noogiefy(int n) { … }
   // Asla cagirilmaz! Test edilmeli!
   private void catify(double f) { … }
}
```

Yalnızca public metot testinin yetersiz olduğu durum:

```
// Bütün metotların karmaşık olduğunu
varsayalım
public boolean foo(boolean n) {
  if (bar(n) && baz(n) && beta(n)) {
    return true;
  } else if (baz(n) ^ (thud(n) || baa(n)) {
    return false;
  } else if (meow(n) || chew(n) || chirp(n)) {
    return true;
  } else {
    return false;
  }
}
```

- Her bir private metodun test edilip edilmediğini belirlemek bile oldukça zordur
- Eğer `foo` hata verirse hangi metotta kusur bulunmaktadır?

Private metotları nasıl test ederiz?

- Öncelikle programlama dilinin buna izin vermesi gerekir
- Java için düşünecek olursak reflection sayesinde buna izin vermektedir.

```
class Duck {
  private int quack(int n) { … }
}
// quack metodunu int parametresi ile al
Method m = Duck.class.getDeclaredMethod("quack", int.class);
// metodu erişilebilir yap
m.setAccessible(true);
// For instance methods, 1st argument is always instance
Object ret = m.invoke(new Duck(), 5);
```

- Ayrıntılar için chapter 24'ü okuyunuz
- Bu derste, çoğu zaman public metotların testinden söz ediyor olacağız

Textbook Chapter13'ü okuyunuz

- Ayrıca:
  SoftwareTesting\junit_example'i inceleyebilirsiniz.
- Kullanım Kılavuzu:
  - https://junit.org/junit5/docs/current/user-guide/
- Javadoc:
  - http://junit.sourceforge.net/javadoc/