Birim Test – Part 2

Temiz Birim Test Nasıl Yazılır?

JUnit v. Hamcrest Assertions

Varsayılan olarak sunulan Junit assertion'larında bazı eksiklikler bulunmaktadır:

- **Assertion metotlarında beklenen ve mevcut değerlerin sıralamasının karıştırılması oldukça kolaydır** (Genelde ben de karıştırıyorum!). Sonuçları çok ağır değildir ancak burada yapılacak yanlış sıralama diğer programcıların kafasını karıştırabilir.
- **Beklenen-mevcut değer ilişkilerinin farklı olması için farklı bir stil gereklidir; yani farklı bir assertion yöntemi**
- Mevcut assertion yöntemleri biraz sınırlıdır
- Ayrıca hata mesajlarını özelleştirmek zordur

Bu nedenlerden dolayı bazı programcılar Hamcrest tarzı assertion'ı tercih etmektedir.

Farklı Bir Assertion Tarzına Geçtiğimizi Fark Ettiniz mi??

Hamcrest Assertions

```
@Test
public void isocelesTest() {
    Triangle.Type result = Triangle.classify(5, 10, 10);
    assertThat(result, equalTo(Triangle.Type.ISOSCELES));
}
```

Her iddia, generic `asserThat` metodunu kullanır

Genel assertion formatı, doğal dile daha yakındır ve test edilen birimin **gerçek sonucunun** parametresi ilk sıradadır.

Gerçek ve beklenen sonuçlar arasındaki ilişki bir eşleştirici tarafından (bu durumda `equalTo`) belirtilir.

Hamcrest Matchers

Hamcrest'te `equalTo` gibi çok sayıda "eşleştirici" bulunmaktadır.

Çeşitli türleri içeren assertionların yazılmasına yardımcı olabilirler, örneğin:

- Dizeler – örneğin, bir dizenin bir alt dize içerip içermediğini kontrol edebilir, büyük/küçük harf durumunu göz ardı edebilir vb.
- Koleksiyonlar – bir öğenin koleksiyonda olup olmadığı; bir koleksiyonun ne içerdiği, sıranın göz ardı edilmesi vb.
- Bakınız **http://hamcrest.org/JavaHamcrest/javadoc/2.2/org/hamcrest/Matchers.html**

Uygun eşleştirici mevcut değilse, kendinizinkini yazmak çok kolaydır.

Bakınız **https://www.baeldung.com/java-junit-hamcrest-guide**

Testlerinizi *Tam* ve *Kısa* Hale Getirin

Testin, okuyucunun sonuca nasıl ulaştığını anlaması için gereken tüm bilgileri içerdiğinden emin olun.
Başka alakasız ve dikkat dağıtıcı bilgi içermediğinden emin olun.

Bir test senaryosu, okuyucunun sonuca nasıl ulaştığını anlaması için ihtiyaç duyduğu tüm bilgileri içerdiğinde tamamlanmış demektir.

```
@Test
// An incomplete test!
public void shouldEndGameWithWinnerWhenFourInARowVertically() {
    Connect4 c4 = new Connect4(7, 6);
    makeMoves(c4);
    assertThat(c4.isGameOver(), equalTo(true));
    assertThat(c4.winner(), equalTo(Piece.RED));
}
```

Bu yardımcı yöntemde neler oluyor?
Neden **KIRMIZI**?
Nereden geliyor?

Testlerinizi **Tamamlayın**

Bir test senaryosu, okuyucunun sonuca nasıl ulaştığını anlaması için ihtiyaç duyduğu tüm bilgileri içerdiğinde tamamlanmış demektir.

```
@Test
// An incomplete test!
public void shouldEndGameWithWinnerWhenFourInARowVertically() {
    Connect4 c4 = new Connect4(7, 6);
    makeMoves(c4);
    assertThat(c4.isGameOver(), equalTo(true));
    assertThat(c4.winner(), equalTo(Piece.RED));
}
```

Bu yardımcı metotta neler oluyor?

Neden **Kırmzı**? Nereden geldi?

Testlerinizi **Tamamlayın**

Bir test senaryosu, okuyucunun sonuca nasıl ulaştığını anlaması için ihtiyaç duyduğu tüm bilgileri içerdiğinde tamamlanmış demektir.

```
@Test
public void shouldEndGameWithWinnerWhenFourInARowVertically() {
    Connect4 c4 = new Connect4(7, 6);
    c4.makeMove(0); // RED
    c4.makeMove(1); // YELLOW
    c4.makeMove(0); // RED
    c4.makeMove(1); // YELLOW
    c4.makeMove(0); // RED
    c4.makeMove(1); // YELLOW
    c4.makeMove(0); // RED
    assertThat(c4.isGameOver(), equalTo(true));
    assertThat(c4.winner(), equalTo(Piece.RED));
}
```

Tüm hamlelerin sıralanması yöntemi daha uzun hale getirir ve hareket sırasının yardımcı bir yöntem olarak yeniden kullanılmasını engeller, ANCAK ne olup bittiğini daha net hale getirir. Tahtaya iki sütun dikey parça ekleniyor ve 0. sütunda **KIRMIZI** kazanıyor

DRY

DRY – Don’t Repeat Yourself: mühendislerin bütün akışı takip etmek yerine kodun bir parçasını güncellemesi gerekir.

Dezavantaj: Kodu daha az anlaşılır hale getirebilir, **zincir şeklinde referans vermeye neden olabilir**. Bu, kodla çalışmayı kolaylaştırmak için ödenmesi gereken küçük bir bedeldir…

… ancak maliyet/fayda analizi testlerde farklı sonuç vermektedir:

- **Yazılım değiştiğinde testlerin bozulmasını isteriz**
- **Production ortamındaki kodun bir test kümesi bulunur böylelikle işler karmaşıklaşsa bile işlerin çalıştığını garanti eder. Test senaryoları ise kendi ayakları üzerinde durmalıdır.** Yani testleri test etmek durumunda kalırsanız bir şeyler ters gidiyor olabilir.

DAMP not DRY

DAMP: Descriptive And Meaningful Phrases (Açıklayıcı ve Anlamlı İfadeler)

DAMP, DRY'nin yerini almaz; yalnızca tamamlayıcıdır.

"Helper" yöntemler, ayrıntıları test edilen davranışla ilgili olmayan tekrarlanan adımları hesaba katarak testleri daha kısa hale getirerek daha net hale getirmeye yardımcı olabilir.

Ancak yeniden düzenleme, yalnızca tekrarı azaltmak için değil, testleri daha okunabilir ve açıklayıcı hale getirecek şekilde yapılmalıdır.

Açıklık ve kısalıktan ibaret olmayın.

Testlerinizi Kısa Hale Getirin

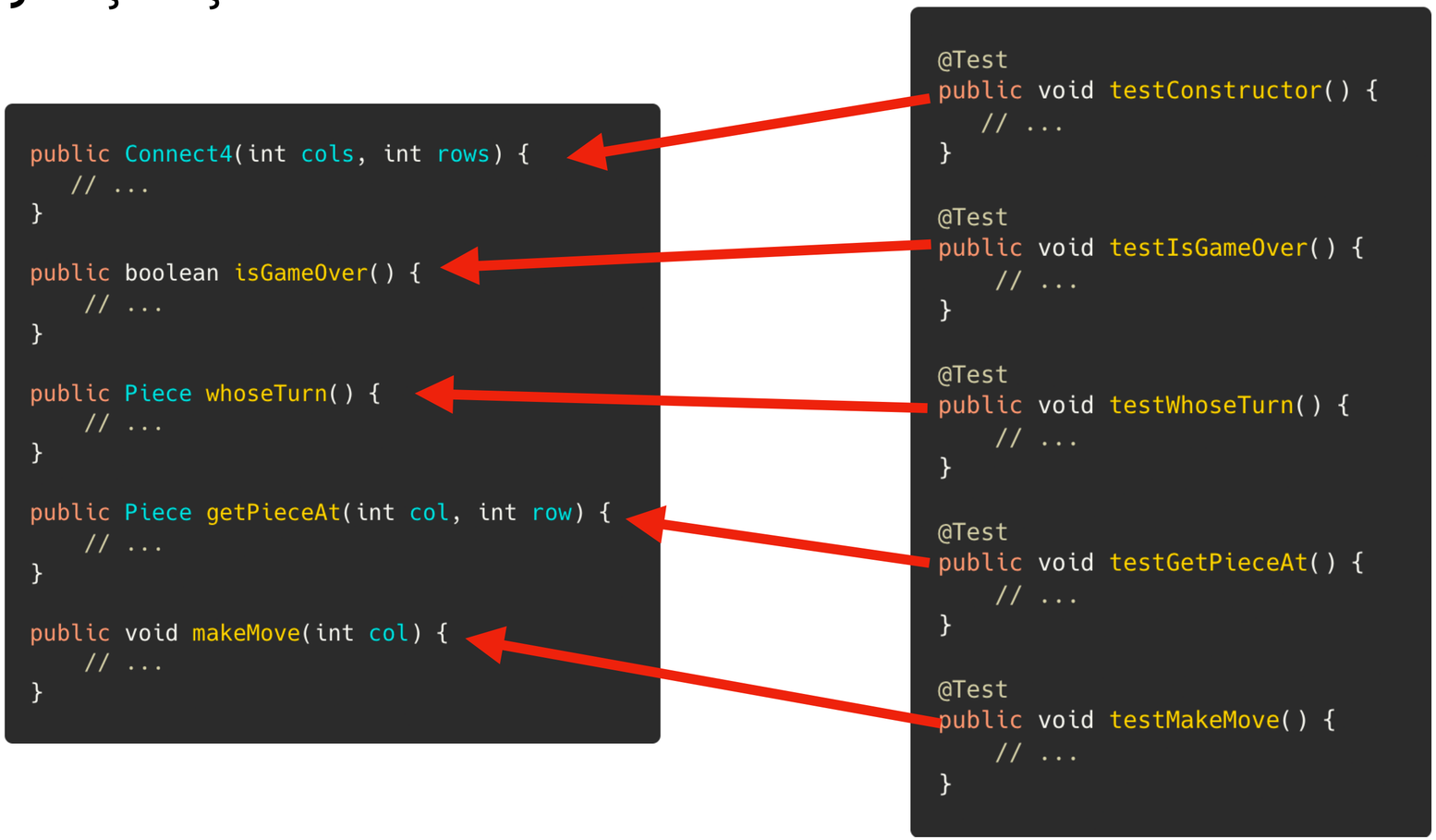
Bir test senaryosu, başka dikkat dağıtıcı veya ilgisiz bilgi içermediğinde kısa ve özdür.

```
@Test
public void shouldEndGameWithWinnerWhenFourInARowVertically() {
    Connect4 c4 = new Connect4(7, 6);
    c4.makeMove(0); // RED
    c4.makeMove(1); // YELLOW
    c4.makeMove(2); // RED
    c4.makeMove(2); // YELLOW
    c4.makeMove(3); // RED
    c4.makeMove(3); // YELLOW
    c4.makeMove(0); // RED
    c4.makeMove(1); // YELLOW
    c4.makeMove(0); // RED
    c4.makeMove(1); // YELLOW
    c4.makeMove(0); // RED
    assertThat(c4.isGameOver(), equalTo(true));
    assertThat(c4.winner(), equalTo(Piece.RED));
}
```

Testimiz burada test senaryosu için gerekli olmayan birçok hamle içeriyor ve tahtanın görselleştirilmesini ve test sonucunun ne olması gerektiğini daha da zorlaştırıyor.

Yöntemleri Test Etmeyin – Davranışları Test Edin

Birçok mühendisin ilk içgüdüsü, testlerinin yapısını kodun yapısıyla eşleştirmeye çalışmaktır.



Yöntemleri Test Etmeyin – Davranışları Test Edin

Ancak tek bir yöntemin birden fazla davranışı ve/veya daha fazla test gerektiren bazı zor köşe durumları olabilir.

Örneğin `makeMove'un` panonun durumuna bağlı olarak çeşitli davranışları olabilir.

Bu yönteme ilişkin **bir** testin hiçbir anlamı yoktur ve büyük ihtimalle çok açık ve net olmayacaktır.

En iyisi: Her davranış için **testler** yazın.

Davranışı Test Etme

Davranış, bir sistemin belirli bir durumdayken bir dizi girdiye nasıl yanıt vereceğine ilişkin yaptığı garantidir.

Davranış şöyle tanımlanabilir: **“given X when Y, then Z”**

Örneğin:

Given bir Connect4 tahtası RED ilk başlar

When RED bir parça oynadığında

Then YELLOW sonraki sıradadır.

Göz önüne alındığında bir Connect4 tahtası RED ilk başlar

Şunu yaparsam: RED bir parça oynadığında

Şu olur: YELLOW sonraki sıradadır.

Davranış Odaklı Testler Yazma

Davranış Odaklı testler daha çok doğal dil gibi okumaya eğilimlidir, bu nedenle onları buna göre yapılandırın.

```
@Test
public void shouldChangePieceAfterTurn() {
    // Given a Connect 4 Board, with RED starting first
    Connect4 c4 = new Connect4(7, 6);

    // When RED makes a move
    c4.makeMove(0);

    // Then it's YELLOW's turn next
    assertThat(c4.whoseTurn(), equalTo(Piece.YELLOW));
}
```

Test Edilen Davranışın Ardından Yapılan İsim Testleri

İyi bir isim, test edilen eylemleri («when») ve beklenen sonucu («then») ve bazen de durumu («given») tanımlar.
İsmin «should» ile başlaması iyi bir yöntemdir; ör.

`shouldInitializeCorrectly` `shouldChangePieceAfterTurn`

`shouldEndGameWithWinnerWhenFourInARowHorizontally` **vs.**

Testlere Mantık Koymayın

Testlere veya mantıksal işlemlere koşul cümleleri veya döngüler koymayın.

Testler, test edilmeyi gerektiren kod parçaları olarak değil, gerçeğin basit ifadeleri olarak okunmalıdır*!*

Burada test, tahtanın her konumunu kontrol etmeye ve onun bir parçaya(Piece) ayarlanmadığından (yani null olduğundan) emin olmaya çalışıyor.

```
@Test
public void shouldInitializeCorrectly() {
    // Given a new Connect 4 Board
    Connect4 c4 = new Connect4(7, 6);

    // Then it's RED's turn
    assertThat(c4.whoseTurn(), equalTo(Piece.RED));

    // Then the board has no piece in every position
    for (int i=0; i < 7; i++) {
        for (int j=0; j < 6; j++) {
          assertThat(c4.getPieceAt(j, j), nullValue());
        }
    }

    // Then the game is not over
    assertThat(c4.isGameOver(), equalTo(false));

    // Then there is no winner
    assertThat(c4.winner, equalTo(null));
}
```

Ama hata, geliştirici bunun yerine `(j, j)`'yi kontrol ederek bir hata yaptı `(i, j)`. Test başarısız olmayacağından hatanın fark edilmesi zordur.

Testlere Mantık Koymayın

Testlere veya mantıksal işlemlere koşul cümleleri veya döngüler koymayın.

Testler, test edilmeyi gerektiren kod parçaları olarak değil, gerçeğin basit ifadeleri olarak okunmalıdır*!*

Daha küçük bir 2x2 kartı başlatmak ve her konumu açıkça kontrol etmek daha iyidir

```
@Test
public void shouldInitializeCorrectly() {
    // Given a new Connect 4 Board
    Connect4 c4 = new Connect4(2, 2);

    // Then it's RED's turn
    assertThat(c4.whoseTurn(), equalTo(Piece.RED));

    // Then the board has no piece in every position
    assertThat(c4.getPieceAt(0, 0), nullValue());
    assertThat(c4.getPieceAt(0, 1), nullValue());
    assertThat(c4.getPieceAt(1, 0), nullValue());
    assertThat(c4.getPieceAt(1, 1), nullValue());

    // Then the game is not over
    assertThat(c4.isGameOver(), equalTo(false));

    // Then there is no winner
    assertThat(c4.winner, equalTo(null));
}
```

Birim Testlerinizi Netleştirme

1. Testlerinizi **kısa** ve **eksiksiz** yapın (DAMP ve çok fazla DRY olmadan!)
2. Testleri yöntemler etrafında yapılandırmayın; bunun yerine **davranışlar etrafında yapılandırın**
3. Davranışı test etmek için **Given-When-Then** modelini kullanın
4. **Test Edilen Davranışın Ardından Yapılan İsim Testleri**
5. **Testlere mantık koymayın**