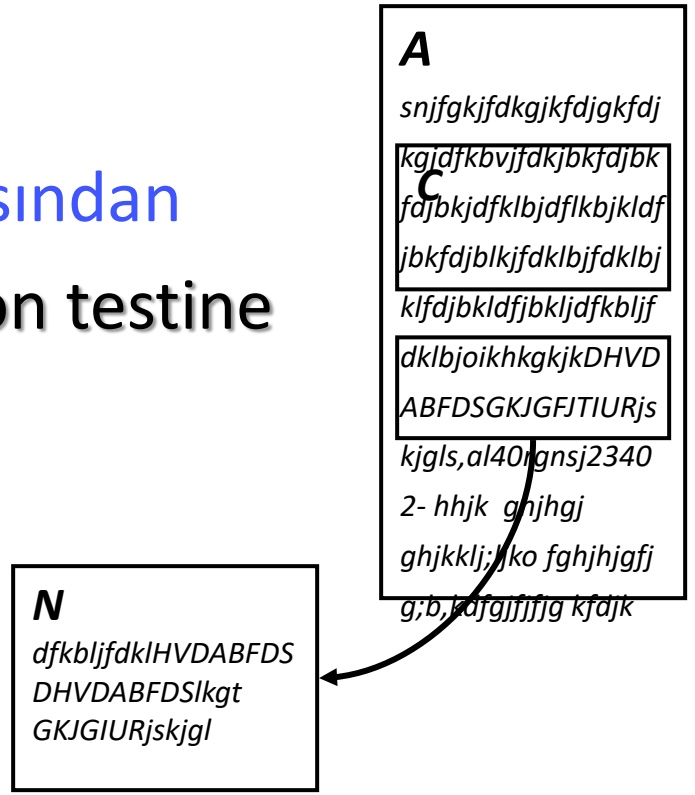
Regression Testing: What to Retest

- Kod parçası C, test edilmiş olan A uygulamasının içinde olsun.
- A uygulaması N kod parçası ile değiştirilmiş olsun.

❖ Eğer *C kodu, N kod parçasına* ***bağlı ise***

C üzerinde **Regresyon Testi** yapılır.

❖ Eğer *C* kodu, N ile değiştirilmiş kod parçasından tamamen **bağımsız ise**, C üzerinde **regresyon testine gerek yoktur**



Test Sonucu olarak «Test Errors» ve «Test Failures»

- Bir testin başarısız olması için 2 neden vardır:
  i) failed olabilir ii) error oluşabilir
- İkisi arasındaki fark önemsizdir.

Ama:

Failure: Kod parçasından beklenen değer elde edilmemiştir. Bir değişkenin sonuçta elde edilen değeri beklenen değerden farklı ise, test «fail» olur. Bu bölüme erişim olmuştur ve sistemsel sorun /çökme oluşmuştur.

Error: Beklenmeyen bir durum ile karşılaşılmıştır. İlişkideki sınıf henüz tanımlanmadığı için bir error oluşmuştur. Çünkü geliştirici kodu kendisinden beklenildiği şekilde üretmemiştir. Örneğin yazım (sentaks) hatası yapmıştır.

Her iki durumda da başarılı olarak tamamlanmaz (not pass). «test failure» ya da «test error» olarak raporlanır.

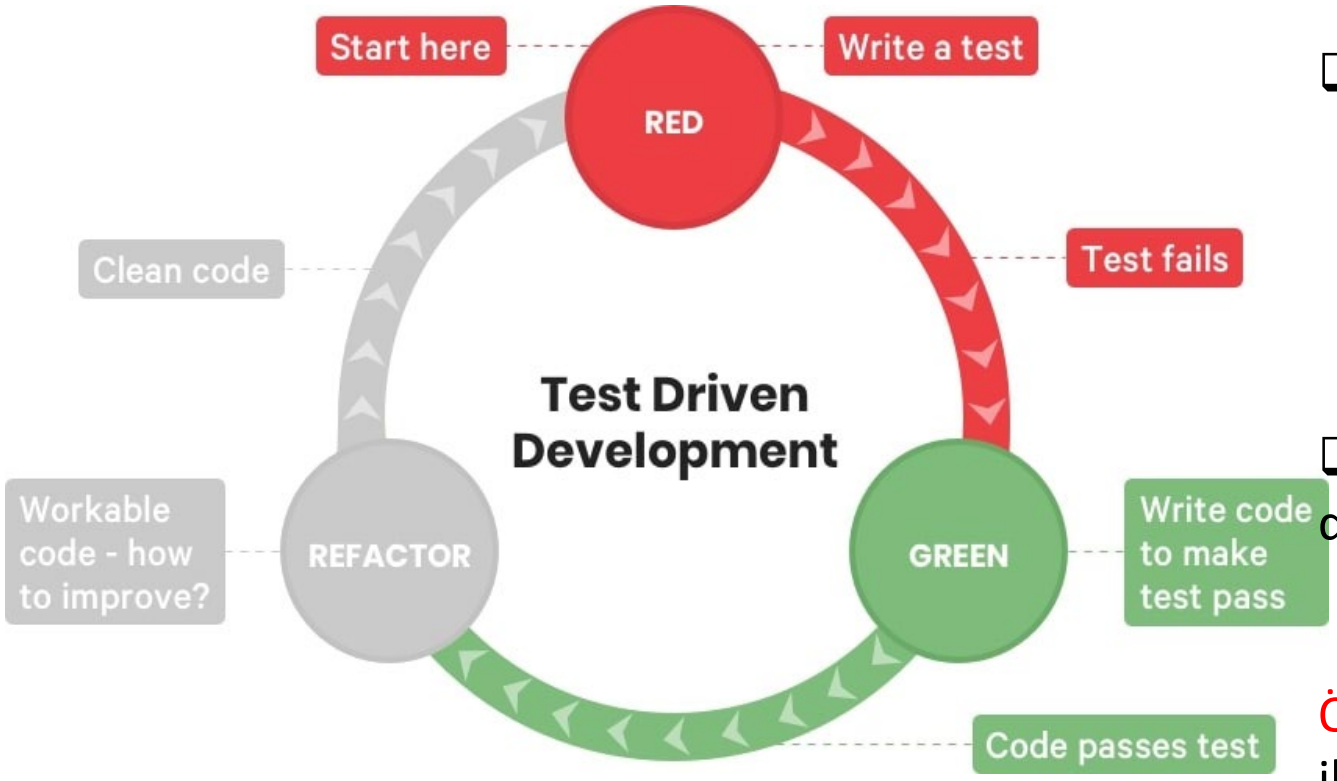
YAZILIM TESTİ VE PROJE YÖNETİMİ

Test Driven Development - TDD

2024 BAHAR

13 MAYIS

Test Driven Development /TDD



- ❑ İmplementasyon öncesinde önce bir test yazıldığı için başarısız olması beklenir.
  - ❖ Test başarılı ise yanlıştır.
  - ✓ Test, zaten var olan veya yanlış yazılmış olan herhangi bir şeyi açıklar.
- ❑ Test yazarken «green state» olmak «false positive» olma işaretidir.
  - ❖ İstenen işlevselliği uygulayan kod yazılmıştır; ama en iyi tasarım oluşturulmamıştır.
  - ❖ Sadece testi başarılı yapacak (pass) basit bir kod geliştirilmiştir.
- ❑ Bu tür testler kaldırılmalı veya kod yeniden düzenlenmelidir (refactored)
  - ❖ Bu işlemler, tasarımın geliştirilmesini, tasarımın okunabilir ve sürdürülebilir hale getirilmesini içerir.

Önemli : Yeniden düzenleme aşamasında (Clean Code ilkeleri) başlangıç tasarımı bozulmamalıdır.
Bir sonraki teste geçildiğinde ve sonraki işlevsellik parçası uygulanırken bu döngü tekrar edecektir.

.

https://www.codica.com/blog/test-driven-development-benefits/

Bu yöntem ile ilgili farklı dillerdeki tüm kodları farklı Türkçe blog sitelerinde bulabilirsiniz

SUT System Under Test

- Arrange-Act-Assert (Test Driven Development)

  Düzenleme- İşlem (Eylem) –Beyan (İddia)

- Given-When-Then (Behaviour Driven Development)

Gherkin Dili

TDD - Test Driven Development

- Problem çözümüne, ürünün geliştirilmesine önce test yazılarak başlanır.
  - Çözüme hedeflenen yapı hakkında düşünerek başlanır.
  - «Tanımlanacak olan davranış (aktivite) ile ne yapılmak istenmektedir?» sorununun cevabı araştırılır.
  - Uzun yıllar bazı gruplar için mantıksız değerlendirilmiş olsa bile son metodolojilerde tercih edilmektedir.
  - Problemin asıl çözümü test yazımından sonra gerçekleşir.
- Test yapılmadan önce problemin kodu yazılırsa, teste geçildiğinde bu davranıştan ne beklendiği bilinmektedir.
  - Kod yazarak ürünün geliştirilmesi için kod yazılmasına beyanları, yani iddiaları (assertions) yazarak başlanır.
- TDD yaklaşımı ile problemin çözüm koddan önce testi yazıldığında geçerli olur.
  - O nedenle de önce test yazıldığında arrange (düzenleme) bölümünden başlamak iyi bir seçenektir.

İyi Kod Yazmada bir Alternatif:
3A / Arrange-Act-Assert

- Arrange (Düzenleme) bölümünde, test edilen sistem (System Under Test / SUT) ve sistemin bağlı olduğu, yani ilişkili olduğu bilgi istenen duruma getirilir.
- Act bölümünde, SUT üzerindeki yöntemler çağrılır, hazırlanmış olan bağımlılıklar iletilir; eğer varsa çıktı değeri elde edilir.
- Assert bölümünde sonuç doğrulanır.
  - Sonuç, dönen değer(retured value); yani SUT'in sonuçlanan (final) bir durumudur ya da farklı bir ifade ile,
  - Sonuç, SUT altında sistemin bağlı olduğu kod parçalarındaki yöntemdir.

Birim Testi için alternatif bir yol

- 3A Şablonu

Test « Arrange- Act- Assert» yaklaşımı ile gerçekleştirsin.

Test edilecek kod:

```
public class Calculator{
    public double Sum(double first, double second)
 {      return first + second;    }
olsun
```

Unit Test: SUT - 3A yaklaşımı «Sum» yönteminin kullanılması

```
public class CalculatorTests                    1   Class-container for a cohesive set of tests
{                                               2
public void Sum_of_two_numbers()                3   Name of the unit test
  {
      // Arrange
   double first = 10;                           4    Arrange section
 double second = 20;                            4
 var calculator = new Calculator();             4

  // Act
   double result = calculator.Sum(first, second);   5     Act section

  // Assert
     Assert.Equal (30, result);                        6     Assert section
  }
}
```

3A ile unit testi :
Mutlak Değer Fonksiyonu (Python)

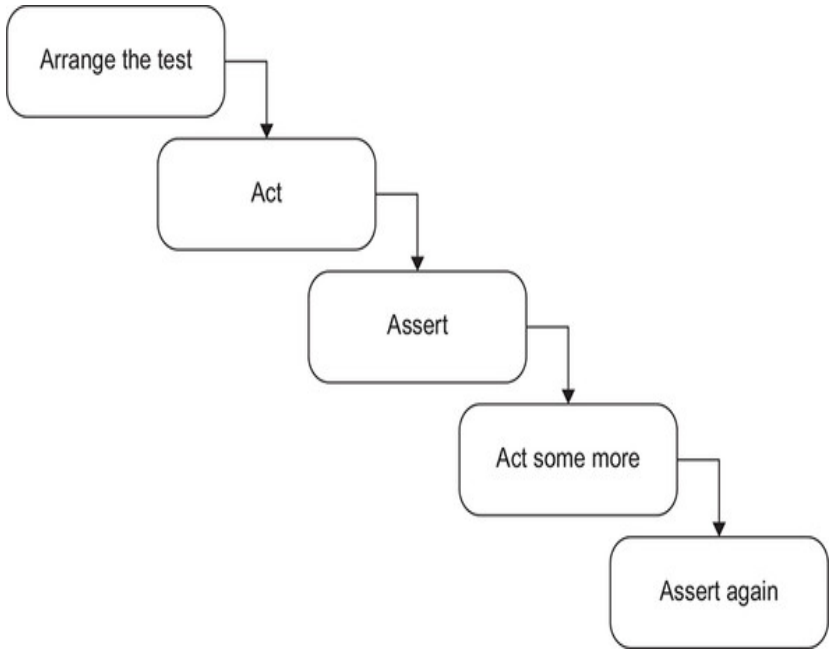
```
def test_abs_for_a_negative_number():

    # Arrange
    negative = -5

    # Act
    answer = abs(negative)

    # Assert
    assert answer == 5
```

3A ile Feature Testi
(package , webUI, mobil testler)

```
import requests
def test_duckduckgo_instant_answer_api_search():
  # Arrange
  url =
'https://api.duckduckgo.com/?q=python+programmi
ng&format=json'
  # Act
  response = requests.get(url)
  body = response.json()
  # Assert
  assert response.status_code == 200
  assert 'Python' in body ['AbstractText']
```

Arrange adımı: URL nin son ucunu Python için arar. Bu nedenle URL ve sorgu (query) parametreleri öne çıkar.
Act adımı: API çağrılır. “requests” ile URL kullanılır ve gelen cevap (response) çözümlenerek JSON ’dan (body) Python sözlüğüne gelir.
Assert adımı : HTTP statü kodu istenilen değer mi kontrol edilir. 200, “OK” ya da “success,” demektir ve bilgilendirme görünür.

3A yaklaşımı ile Test için Önemli: Arrange, Act, Assert



Tek bir eylem, testin **birim** testi alanında kalmasını sağlar
Test basittir, hızlı ve anlaşılması kolaydır.
Bir dizi **eylem ve iddia** (act and assert) içeren bir test **yeniden düzenlenmelidir.**

- **Çoklu arrange** (düzenleme) , act (işlem yapma) , **assert** (iddia) bölümleri testin aynı anda çok fazla şeyi doğruladığını belirtir.
  - Böyle bir testin birkaç teste bölünmesi gerekir.
- **Assert** ile ayrılmış birden fazla eylem bölümü varsa testin birden çok davranışı doğruladığı anlaşılır.
  - Bu test artık bir birim testi değil, bir **entegrasyon** testidir.

Test Kalitesinin Belirlenmesinde Kapsam Ölçümleri (Covarege Measurements)

- Test paketi kaynak kodu ne oranda çalıştırmaktadır?
- Kapsama değeri ne kadar yüksekse testin o kadar iyi olduğuna inanılır.
  - %0 ile %100 arasında değişir.
- Kapsam metrikleri önemli geri bildirimler sağlamalarına rağmen, bir test paketinin kalitesini değerlendirmede etkin değildirler.
- Kapsam metrikleri değerlendirilen kodun birim test etme yeteneğini ölçer.

Buradan:

- Kapsam ölçümleri **iyi bir olumsuz gösterge** (good negative indicator) olmasına rağmen **kötü bir pozitif göstergedir** (false positive indicator).

Good Negative Indicator / Bad Positive Indicator

- Bir metrik, temel kod için çok az kapsam olduğunu gösteriyorsa (%10), yeterince test yapılmamıştır.
- %100 kapsam kaliteli bir test paketi kullanıldığını garantisi etmez.
- Bir test paketi için yüksek kapsama alanı da kesinlikle paketin kalitesinin ölçütü değildir.

Code Coverage / Statement Coverage örneği: isStringLong yöntemi*

```
public static bool IsStringLong(string input)
{
    return input.Length > 5 ? true : false;
}

public void Test()
{
    bool result = IsStringLong("abc");
    Assert.Equal(false, result);
}
```

- Kod değişmiştir. Refactoring uygulanmıştır. Yani, klasik if komutu yerine yazılmıştır.
- Test, üç kod satırını kullanır; kod kapsamı (statement coverage) %100 'dür.
- **Sonuç:** Kapsama (covarege) değerlerinin kolaylıkla değiştirileceğidir.
- Kod ne kadar kompakt olursa, test kapsamı metriği o kadar iyi olur. Zira, sadece kod satırları hesaplanır.

AMA

- Daha fazla kodu daha az alana sıkıştırmak, test paketinin değerini veya temel kodların sürdürülebilirliğini değiştirmeyecektir.

*** isStringLong** yöntemine uygulanan birim testi
Manning,Unit Testing Principles, Practices, and Patterns isimli kitaptan alınmıştır.

Branch Coverage Metric isStringLong yöntemi)

Test paketi tarafından uygulanan kod dallanmalarının sayısının tüm kodun toplam dallanma sayısına oranıdır.

```
public static bool IsStringLong(string input)
{
    return input.Length > 5 ? true : false;
}

public void Test()
{
    bool result = IsStringLong("abc");
    Assert.Equal(false, result);
}
```

IsStringLong yönteminde iki dallanma vardır: Değişkeninin uzunluğunun beş karakterden büyük olduğu durum ve olmadığı durumdur.

Test, bu dallardan yalnızca birini kapsar, dolayısıyla branch coverage kapsamı metriği 1/2 = 0,5 = %50'dir.

Test edilen kodun önemi yoktur.
Branch coverage metriği yalnızca dallanma (branch) sayısını değerlendirir.
Bu dalları uygulamak için kaç satır kod gerektiği dikkate alınmaz .

«Branch Coverage» Sorunları

- Testin tüm olası sonuçları doğruladığı garanti edilemez.
- Kod yollarının denenmesinden ziyade, gerçekten test edilmesi için birim testler de uygun assertions (iddialar) içermelidir.
  - Test edilen sistemin ürettiği sonuç, tam olarak üretmesi beklenen sonuç mudur?
  - ✓Bu sonucun birkaç bileşeni de olabilir
  - ✓Kapsam metriklerinin anlamlı olması için bileşenlerin tümü doğrulanmalıdır.

Alternatif test: Sonucun kaydedildiği yöntem bildirimi

```
public static bool WasLastStringLong { get; private set; }*

public static bool IsStringLong  (string input)
{
   bool result = input.Length > 5 ? true : false;
   WasLastStringLong = result;                          1
   return result;                                       2
}

public void Test()
{
   bool result = IsStringLong("abc");
   Assert.Equal(false, result);                         3
}
```

IsStringLong yöntemi farklı şekilde test edilir.
Sonuç, genel bir **WasLastStringLong** fonksiyonuna kaydedilir.

*https://livebook.manning.com/book/unit-testing/chapter-2/v-1/1

1 First outcome /explicit (açıık(donuç
2 Second outcome /implicit (kapalı) sonuç
3 The test verifies only the second outcome.

isStringLong yönteminin testlerinin İrdelenmesi

IsStringLong yönteminin iki sonucu vardır:

- *Explicit Sonuç:* Dönüş değerinin kodlandığı sonuç
- *Implicit Sonuç:* İlgili özelliğin yeni değeri olan örtük bir değer

Örtük sonuç doğrulamamasına rağmen, kapsam metrikleri yine de sonuçları gösterir.

Code Coverager %100

Branch coverage %50.

- Coverage metrics (kapsam ölçümleri), hedeflenen kodun test edildiğini garanti etmez, sadece çalıştığını (yürütüldüğünü) garanti eder.

Assertion-Free Testing

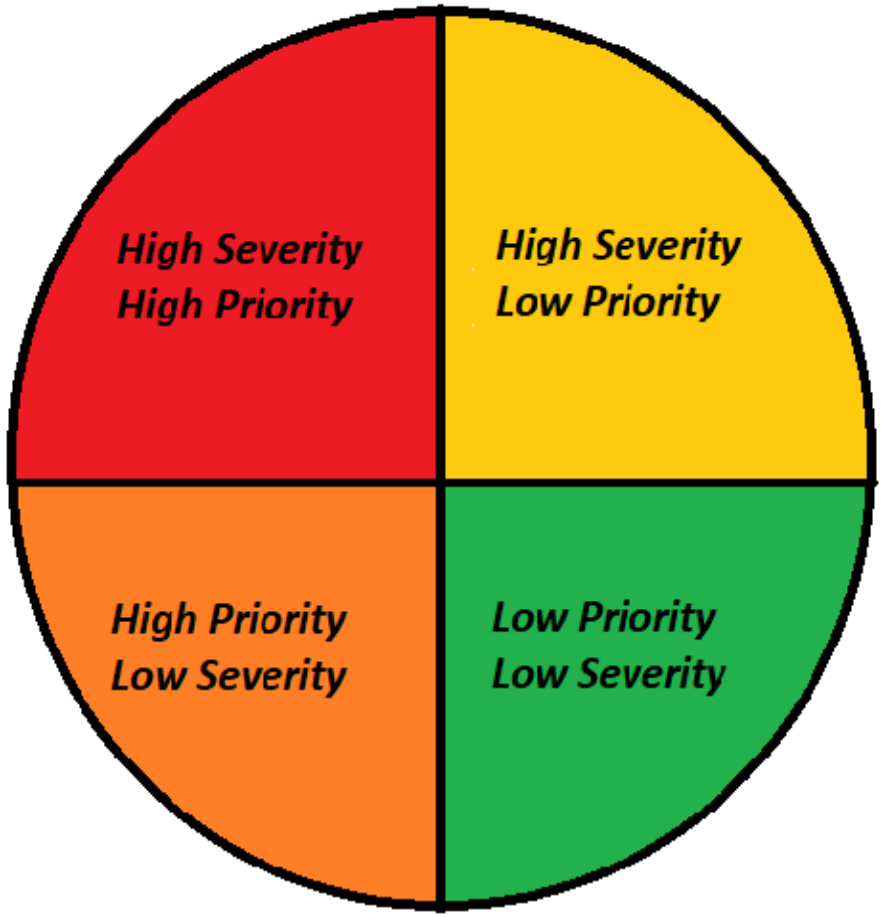
Assert* sınıfının kullanılmadığı test

```
public void Test()
{
    bool result1 = IsStringLong("abc");         1
    bool result2 = IsStringLong("abcdef");      2
}
```

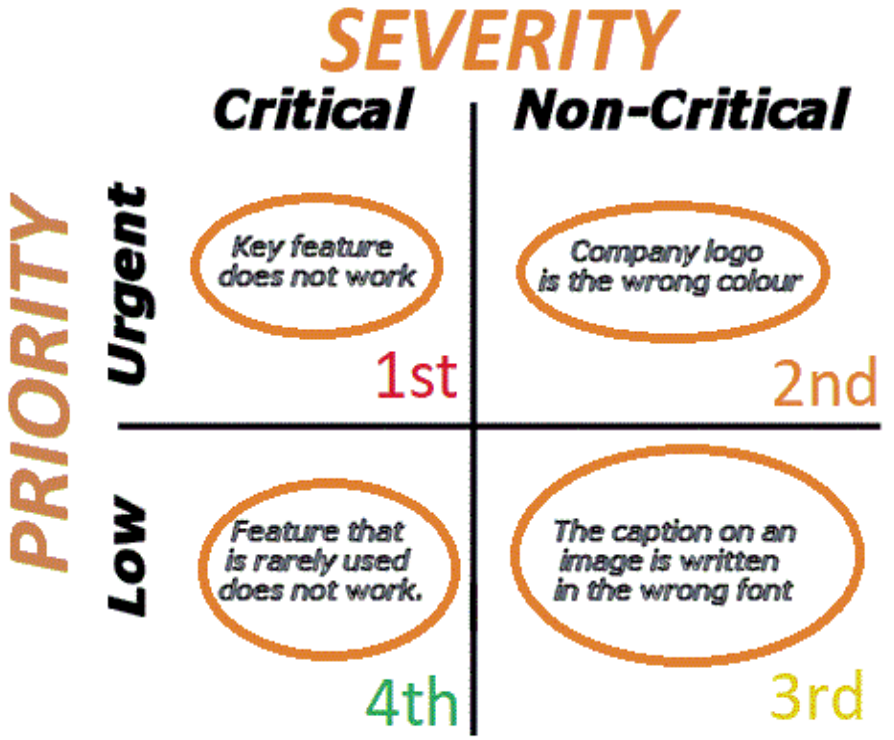
code ve branch coverage metriklerinin ikisi de %100 olarak sağlanır. Bu değerlerin testin kalitesinde bir rolü yoktur. Herhangi bir şey doğrulanmamıştır.

*https://learn.microsoft.com/en-us/dotnet/api/microsoft.visualstudio.testtools.unittesting.assert?view=visualstudiosdk-2022

https://docs.python.org/3/library/unittest.html



pie chart



Example

https://medium.com/hepsiburadatech/bug-severity-and-priority-matrix-ae14fb344559

https://medium.com/hepsiburadatech/bug-severity-and-priority-matrix-ae14fb344559

Priority /Öncelik

Önemsiz/Trivial : Kusur /defect, işlevselliği veya verileri etkilemez. Geçici bir çözüme bilegerek yoktur. Üretkenlik / verimlilik etkilenmez. Bu rahatsız edicidir.

Düşük/ Low : Kusur, onarılması gereken bir durumdur; ancak onarım daha ciddi bir kusur giderilene kadar ertelenebilir.

Orta /Medium: Bu tür bir hata /bug , sistemin tamamen arızalanmasına, iş akışlarının veya programların tamamen durmasına neden olmaz. Ama bu tür hatalar kullanıcılar arasında sıkıntı yaratabilir: Bu da verimliliği / üretkenliği düşürebilir.

Yüksek/High: Bu hatalar /bug , dikkat çekici hatalarla (show- stopper bugs) aynı belirtilere sahiptir. Ancak iş akışını / uygulamayı tamamen durdurmayacaktır. Yüksek öncelikli hatalar geçici bir çözüm ile önlenebilecektir.

Kritik /Critic : Kusur /defect, uygulamanın önemli bir işlevselliğini etkiler ve uygulama, hata düzeltilmeden (fixing the bug) teslim edilemez.

Severity / Önem Derecesi

Önem Derecesi diğer ölçümlerden daha geniş kapsamlıdır. Anlamı, sistemin bu hatadan ne kadar etkilendiğinin belirlenmesidir

Kritik: Sistemin bileşenlerini sonlandıran ve verilerde bozulmaya neden olan kusurdur.. Başarısız olan işlev kullanılamazsa ve sonuca ulaşmak alternatif bir yöntem yoksa, ciddiyet kritik olarak belirtilir.

Büyük /Majör : Sistem bileşenlerini sonlandıran ve verilerde bozulmaya neden olan kusurdur. Başarısız işlev kullanılamaz, ancak sonuçsa ulaşmak için alternatif bir yöntem vardır.

Orta: Fesihle sonuçlanmayan ancak sistemin yanlış, eksik veya tutarsız sonuçlar üretmesine neden olan kusurdur.

Minör: Sistemin kullanılabilirliğine zarar vermeyen ve kusurların etrafında çalışılarak istenilen sonuçların kolayca elde edilebildiği kusurdur.

Kozmetik: Sistemin iyileştirilmesi gerektiği, uygulamanın görünümü ile ilgili değişikliklerin olduğu kusurdur.

Details

Type:	Technology Hardening	Status:	IN ANALYSIS (View Workflow)
Priority:	Major	Resolution:	Unresolved
Affects Version/s:	None	Fix Version/s:	None
Component/s:	None		
Labels:	Bug Short_Term		
Detail User Story:	As a I want to So that		
Epic Link:	Campaign & Offer Management		
Assumptions:	Optional - assumptions made during estimation		
Acceptance Criteria:	Given When Then		
Severity:	4 - Important functionality is unavailable with no workaround		

https://medium.com/hepsiburadatech/bug-severity-and-priority-matrix-ae14fb344559

Details

Type:	Bug	Status:	OPEN (View Workflow)
Priority:	Trivial	Resolution:	Unresolved
Affects Version/s:	None	Fix Version/s:	None
Component/s:	None		
Labels:	MPX_FE		
Epic Link:	Coupon Management		
Detail Bug:	Error Scenario Response Request Endpoint Specific Data (Which merchant, order no in order to reproduce the case by developers) Attachments Screenshot		
Severity:	4 - Important functionality is unavailable with no workaround		

https://medium.com/hepsiburadatech/bug-severity-and-priority-matrix-ae14fb344559