

# YAZILIM KALİTE GÜVENCESİ VE TESTİ

DERS NOTU 1  
2022 GÜZ

# Neler Göreceğiz?

- ❑ Yazılım Testi, yazılım mühendisliğinde niçin bir disiplin olmuştur?  
sorusuna cevap verilir.
  - ❖ «High dependency on **reliable** software»
- ❑ Yazılım testinde temel ilkeler
  - ❖ Kavramlar ve motivasyonlar
    - Quality requirements
    - Software is intangible
    - Faulty software is a serious problem
    - Testing helps to assess software quality

# Vaka Çalışması

## «The Risks of Using Faulty Software»

- ❑ Tüm yazılım sistemlerinin her bir sürümü, teslim edilmeden ve kullanıma sunulmadan önce uygun şekilde test edilmelidir.
  - ❖ Bu, herhangi bir hasar vermeden önce hataları tespit etmeyi ve gidermeyi ifade eder.
- ❑ Sistem bir siparişi hatalı yürütürse bu müşteri, bayi ve üretici için ciddi mali sorunlar getirir, üreticinin imajını zedeler.
- ❑ Ortaya çıkmamış hatalar (*undiscovered faults*), yazılımın çalışmasıyla ilgili riski artırır.

# Birim Testi ve Kod Tasarımı arasındaki ilişki

- ❑ Bir kod parçasının birim testi turnusol testi olarak düşünülebilir; ancak sadece bir yönde çalışır.
  - ❖ Bu, iyi bir olumsuz göstergedir (**good negative indicator**)
    - ✓ Genellikle yüksek doğrulukla (**high accuracy**) düşük kaliteli (**poor quality**) kod simgelenir.
- ❑ Kodun birim testinin yazılmasında zorlanılıyorsa, bu kodun iyileştirilmesi gereğinin güçlü bir işaret olacaktır.
  - ❖ Düşük kalite (**poor quality**) genellikle sıkı bağlılık ile (**tight coupling**) kendini gösterir
    - ✓ Bu, farklı ürünlere ait kod parçalarının birbirinden yeterince ayrılmadığı (**not decoupled**) ve bunları ayrı ayrı test etmenin zor olduğunu

# Birim Testi ve Kod Tasarımı arasındaki ilişki

Aynı anda:

- Kod parçasını birim test etme yeteneği kötü bir pozitif göstergedir (**bad positive indicator**).
  - ❖ Kod tabanını kolayca birim test edebiliyor olmak, kodun mutlaka iyi kalitede olduğu anlamına gelmez.
    - ✓ Proje, yüksek derecede ayrıtırılmış olsa bile bir felakette sonuçlanabilir.

# Software Entropy

Entropi, ait olduğu sistemdeki düzensizliğin ölçüsüdür.

Yazılımda entropi, bozulma eğilimi gösteren kod biçimi ile kendini gösterir.

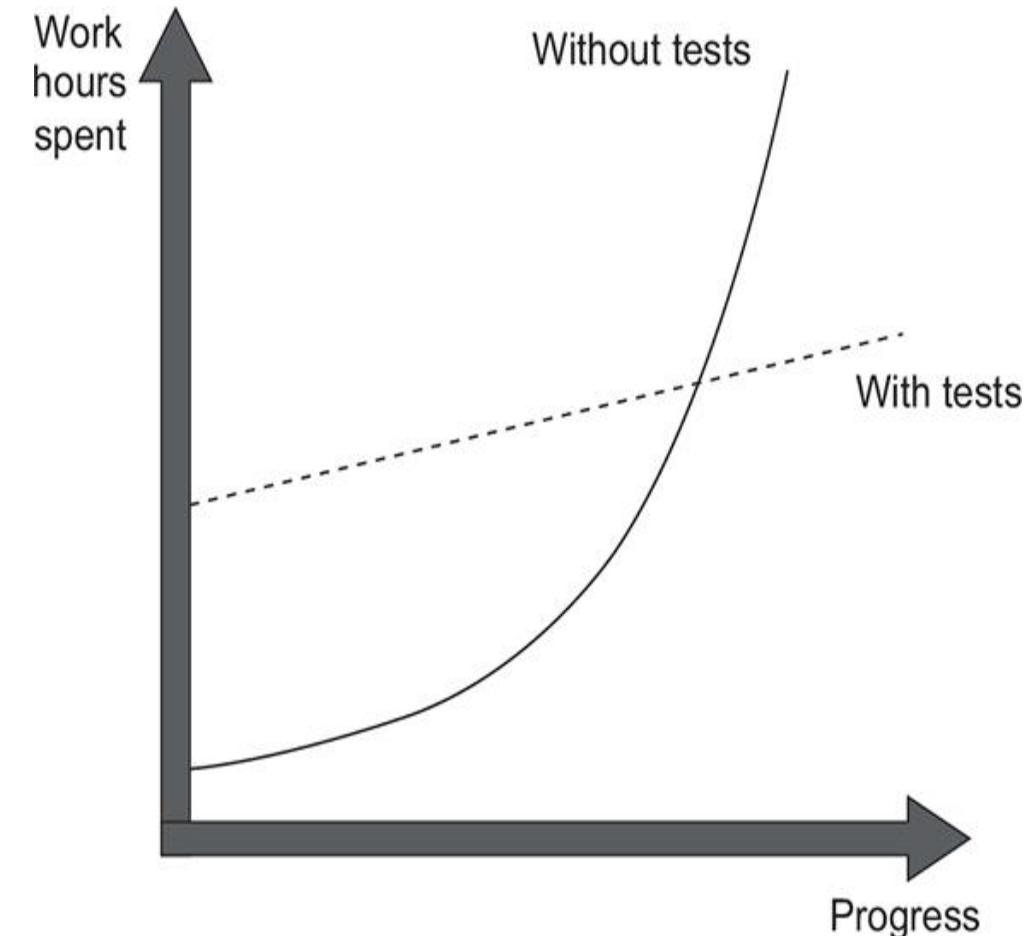
Bir kod temelinden her değiştirildiğinde içindeki düzensizlik miktarı veya **entropi** artar.

Gerekli kontroller yapılmadan sürekli **yeniden düzenlemelerle** sistem giderek daha karmaşık ve düzensiz hale gelir.

Bir hatayı düzeltmek daha fazla hataya neden olur ve yazılımın bir parçasını değiştirmek diğerlerini bozar.

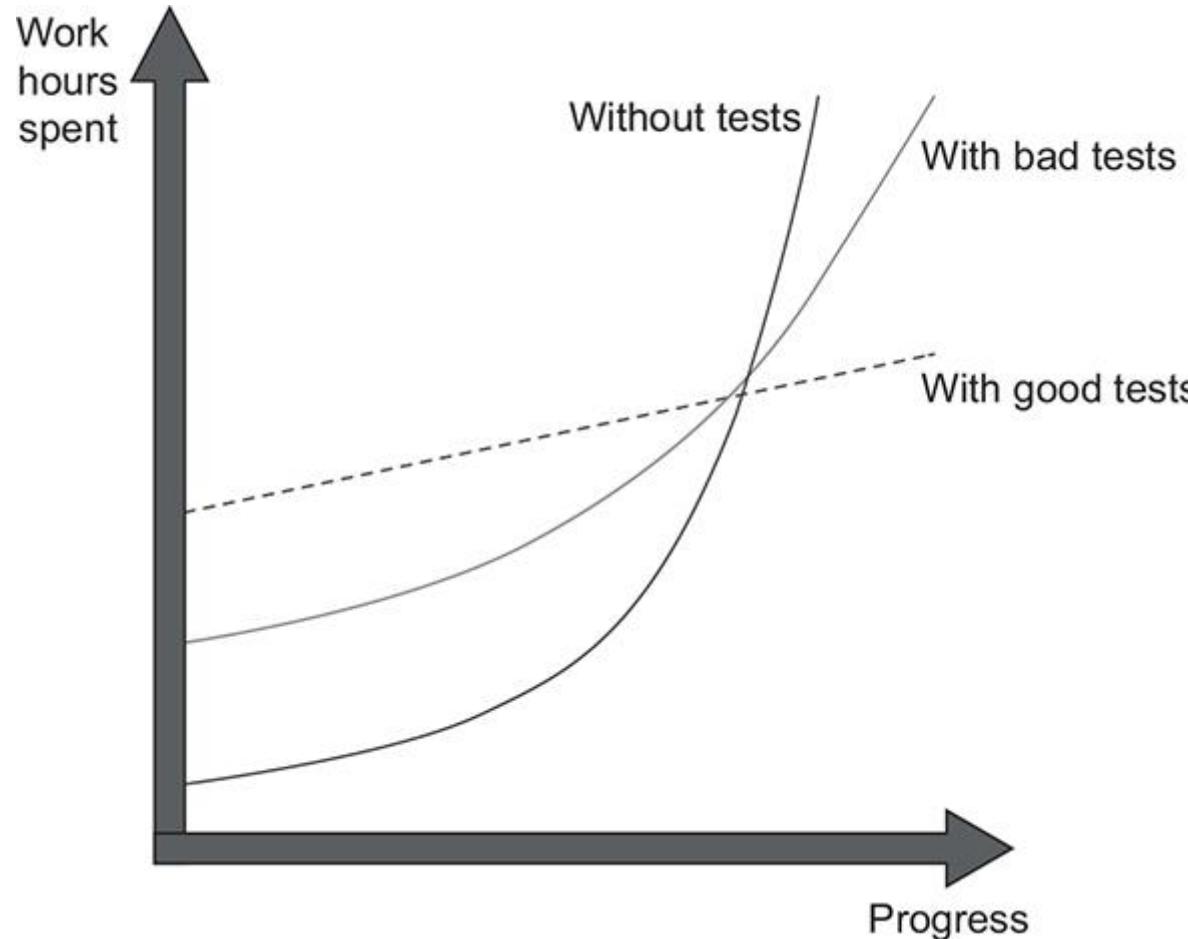
Kod tümüyle güvenilmez hale gelir.

Kodu tekrar istikrara kavuşturmak zorlaşır.



Sustainability (sürdürülebilirlik) ve scalability uzun vadede geliştirme hızını korur.

# Regresyon



Regresyon, genellikle bir kod değişikliğinden sonra (after a certain event) bir özelliğin çalışmasının amaçlandığı gibi durmasıdır.

Regresyon ve yazılım hatası (bug) terimleri eşanlamlıdır.

En temel sorun testlerin başlangıçta önemli çaba gerektirmesidir.

Ancak, sonraki aşamalarda projenin büyümESİNE yardımcı olur ve uzun vadede yararı ortaya

Temel kodları sürekli olarak doğrulayan testler yazılım ürününün ölçeklenmesine de (scability) katkı sağlar.

# Daha fazla birim testi daha iyi sonuç mu?

Hem testin değeri hem de bakım maliyeti önemlidir.

- ❖ Maliyet bileşeni, çeşitli faaliyetlere harcanan süreye göre belirlenir.

## SORUNLAR

- Temel kod yeniden düzenlenliğinde (refactoring) test te yeniden düzenlenenecektir (refactor)
- Testi her kod değişikliğinde çalışma
- Test tarafından oluşturulan yanlışalarla başa çıkabilme
- Temel kodun nasıl davranışını anlamaya çalışırken testi okumak için zaman harcamak

Yüksek bakım maliyetleri nedeniyle net değeri sıfıra yakın hatta negatif olan testler oluşturmak kolaydır.

- Sürdürülebilir (**sustainable**) proje büyümeyi sağlamak için, yalnızca yüksek kaliteli testlere odaklanılmalıdır
  - ❖ bunlar, test paketinde tutulmaya değer tek test türüdür.

# Daha fazla test ???

Code is a liability, not an asset (Kod bir varlık değil, bir yükümlülüğüdür)

Testler de koddur.

Bunlar, belirli bir sorunu çözmeyi amaçlayan kod tabanının bir parçasıdır.

Diğer kodlar gibi birim testleri de hatalara açıktır ve bakım gerektirir.

# Test için Öncelikler

- ❑ Test süreci *spot-check* bir yaklaşımıdır.
  - ❖ Dinamik test ağırlıklı olarak gereksinimleri ne kadar gerçekleştirdiğini ölçer.
  - ❖ İlgili test nesnesine farklı test durumları (cases) farklı veri setleri için uygulanır.
- ❑ Test süreci yapılan testlerden daha fazlasını içerir. Bunlar:
  - ❖ «test planning, test analysis, the design, implementation of test cases» ve
  - ❖ «writing reports on test progress» ve «risk analysis»
- ❑ Test aktiviteleri geliştirilen ürünün yaşam döngüsünden farklıdır ve test dokümantasyonu contract olarak adlandırılır.

# Test için Öncelikler

## Statik ve Dinamik testler

### ❑ Dinamik Testler

- ❖ Gereksinimlerin betimlendiği dokümanlar, kullanıcı hikayeleri ve kaynak kod geliştirme sırasında mümkün olan en erken zamanda yapılır.
  - ✓ Kodun dinamik davranışını analiz etmek için gerçekleştirilir.
- ❖ Analiz edilen giriş değerleri ve çıkış değerleri için yazılımın test edilmesini içerir.

### ❑ Statik Testler

- ❖ Belgelerdeki kusur-arıza (*faults*) ne kadar erken keşfedilir ve giderilirse, gelecekteki geliştirme süreci için o kadar iyidir, böylece hatalı (*flawed*) kaynak materyal bulunmaz.
  - ✓ Hata kaynaklarını bulmak daha kolaydır ve kolayca düzeltilebilir
  - ✓ Hataları önlemek için geliştirmenin erken aşamasında, kodu çalıştırmadan gerçekleştirilir.

# Test için Öncelikler

## «Absence-of-errors is a fallacy”

- ❑ Hataların olmaması da bir yanıldır

### *Validation*

- ❑ Test sadece sistemin müşteri gereksinimlerini, kullanıcı hikayelerini ya da diğer betimlemelerini kontrol etmek değildir. Bunlara ek olarak
  - ❖ Ürünün gerçek (real world) kullanıcılarının bekleyenlerini ne ölçüde karşıladığı da önemlidir.
  - ❖ Sistemin amacına uygun kullanılmasının mümkün olup olmadığı kontrol edilmesi ve planlanan amacına ulaşlığından emin olunması olarak adlandırılır.
    - ✓ Buna VALIDATION denir.

### *Verification*

- ❑ Statik test olarak adlandırılan tüm kontrollerdir.
  - ✓ COVAREGE TESTS

# Fault-Free Large System ??

- ❑ Böyle bir sistem yoktur.
  - ❖ Sistemin karmaşıklığı ve kod sayısına bağlı olarak değişir.
- ❑ «Many faults are caused by a failure to identify or test for exceptions during code development»
- ❑ Sistemin diğer parçaları mükemmel çalışsa bile, giriş verilerinin belli kısımlarında **faults** olabilir.
  - ❖ Sistem canlıya geçtiğinde büyük sistemlerde zaman zaman (giriş datasının belli kombinasyonları çalışırken) **failure** durumları ile karşılaşılabilir.

# Freedom from Faults

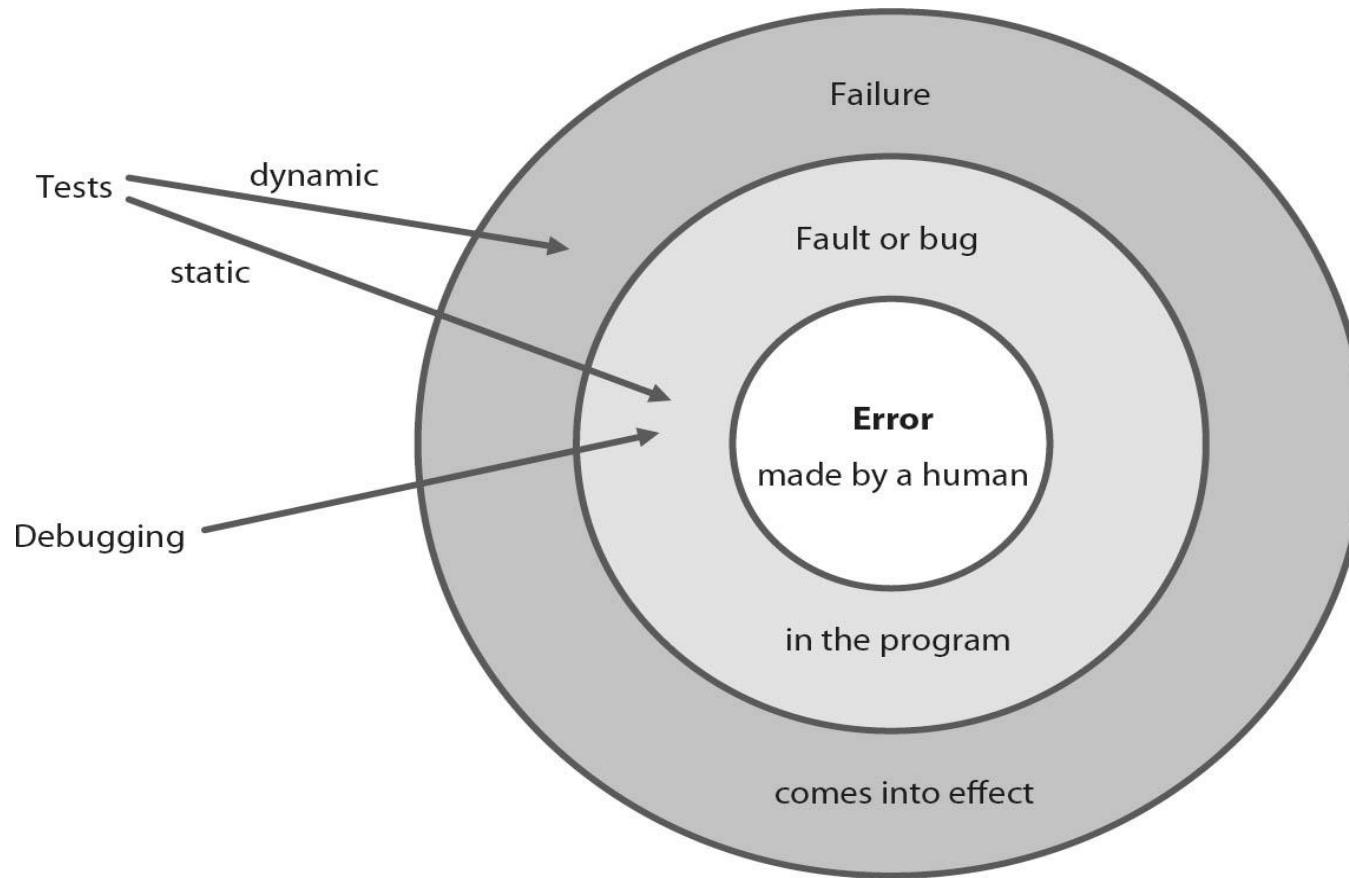
❑ «Kusurlardan(faults) test yoluyla arınılabilir mi?»

- ❖ Yapılan her bir test hatasız bile olsa, çok küçük programların dışında daha sonra yapılacak testler önceden mevcut olmayan bazı hataları ortaya çıkarabilir.
- ❖ Test yaparak hataların tamamen giderildiğini kanıtlamak mümkün değildir.

## «Fault» Durumunun Belirlenmesi

- ❑ «Fault» durumunun açıklasdırılması için o şeyin ne olması gerektiği tam olarak önceden tanımlanmalıdır.
- ❑ Bu da test edilecek alt sistemin gereksinimlerinin bilinmesini gerektirir.
- ❑ Böylece herhangi bir işlev için «fault» durumunun kararı hangi testlerin yapıldığına ve bunların test esasına göre verilebilir.

# Fault – Error – Failure



## «Defect» ve «Fault» Terminolojisi

- ❑ The test basis as a starting point for testing
- ❑ What counts as a defect?
- ❑ Faults cause failures
- ❑ Defect masking
- ❑ People make errors
- ❑ False positive and false negative results
- ❑ Learning from mistakes

## «Defect» nedir?

Defect → kusur, Fault → arıza olarak Türkçeleştirilebilir; ama defect ve fault ile benzer anlam içerirler.

- Sistemin davranışı ne zaman gerçek gereksinimlerine uymaz? sorusudur.
- Tanımlanmış bir gereksinimin yerine getirilmemesi, çalışma zamanında veya test sırasında beklenen davranışta tutarsızlık olmasıdır.
- Yazılım sistemleri yaşlanma veya aşınma nedeniyle failure (başarısız) olmaz.
  - ❖ Her kusur (defect), yazılımın kodlanması ile oluşur, ancak yalnızca sistem çalışırken görünür.

## «Fault», «Failure» ve «Bug»

- Sistemde «failure» nedeni pek çok uygulama için «fault» olarak özetlenir.
- Sistem «failure» test sırasında veya çalışma zamanında (run time) test eden veya kullanıcı tarafından ortaya çıkar.
  - ❖ Örneğin, sistem hatalı çıktı üretir veya çöker.

### ÖZET OLARAK:

- Sistem «failure» nedeni yazılımdaki bir «fault» dur.
- Bu durum «defect» olarak adlandırılır.
- «Bug» sözcüğü, kodda «error» vererek yanlış programlanmış veya unutulmuş bir komut gibi kodlama hatalarından kaynaklanan kusurları (defects) tanımlamak için de kullanılır.

# Defect Masking

- ❑ Bir yerde bir «fault» durumunun düzeltmesi, başka yerlerde beklenmedik yan etkilere neden olabilir ya da
- ❑ Bir yerdeki «fault» durumu ancak mevcut diğer «fault» durumlarının düzeltilmesi ile düzeltilebilir.
- ❑ Her «fault» sistem çözümü yani «failure» nedeni oluşturmaz.
  - ❖ Bazı «failure» durumları ile hiç karşılaşmayabilir, bir kez veya tüm kullanıcılar için sürekli olarak meydana gelebilir.
  - ❖ Bazı «failure» durumları , neden oldukları yerden çok uzakta meydana gelebilir.

# «Error» Nedenleri

- İnsan faktörü
- Zaman baskısı
- İşin karmaşıklığı
- Projenin katılımcıları arasındaki anlaşmazlık
- Sistem ile içsel (internal) ve dışsal (external) arayüzler arasındaki anlaşmazlık
- Proje katılımcılarının yeni teknoloji ile uyumsuzluğu
- Proje katılımcıları arasındaki anlaşmazlık

# Her «unexpected» sonuç «failure» oluşturur mu?

Hayır.

## False-pozitif

- ❖ Temeldeki bir «fault» veya bunun nedeni test nesnesinin bir parçası olmasa bile bu test bir «failure» gösterir.

## False-negatif

- ❖ Testin varlığını ortaya koymasına rağmen bir «failure» meydana gelmez. Bu tür bir sonuç "false negatif" olarak bilinir.

Benzer şekilde

«Correct pozitif» (test sonucu ortaya çıkan «failure» )

«Correct negatif» (test ile onaylanan «expected» davranış)

gerçekleşebilir.

# Test Terminolojisi -I

## ❑ Testing is not debugging

- ❖ «Debugging» yazılım kusurlarını (faults) hatalarını tespit ederken, «fault» oluşturan etkiyi ortaya çıkarmak için «test» kullanılır.
- ❖ Kusurları (faults) bulma ve düzeltme sürecine hata ayıklama «debugging» denir .
- ❖ «Debugging» süreci geliştiricinin sorumluluğundadır.

# Test Terminolojisi -II

## Test Confirmation (Correcting a Fault)

- Ürünün kalitesini yükseltir.

AMA :

- Düzeltmeler yeni «faults» bekentisini arttırır.
- Aşağıdaki iki önemli kabul çevik geliştirme ortamlarında değişebilir:
  - ❖ Test uzmanları testin konfirmasyonundan sorumludur.
  - ❖ Geliştiriciler ise ilgili bileşenin testi (component test) ve debugging işlemlerinden sorumludur.

OYSA gerçek dünyada «fault düzeltmeleri» yeni senaryolarla birlikte yeni «faults» ortaya çıkarır.

- Hedeflenen hatanın giderildiğinden emin olmak için önceki testlerin tekrarlaması ve mevcut «fault» düzeltme işleminin istenmeyen yan etkilerinin kontrol edildiği yeni testler yazılmalıdır.

# Test Terminolojisi –III

## Objectives of Test

- ❑ Statik ve Dinamik testler: Farklı amaçlar için
  - ❖ Test amaçları, hangi testin yapılması gereği içeriğe göre değişiklik gösterir

# YAZILIM KALİTE GÜVENCESİ VE TESTİ

DERS NOTU 2

2022 GÜZ

# Test Kalitesi için Kapsam Ölçümleri (Coverage Measurements)

- ❑ A coverage metric shows how much source code a test suite executes, from none to 100%.
    - ❖ Test paketi kaynak kodu ne oranda çalıştırılmaktadır?
  - ❑ Kapsama değeri ne kadar yüksekse testin o kadar iyi olduğuna inanılır.
    - ❖ %0 ile %100 arasında değişir.
  - ❑ Kapsam metrikleri değerli geri bildirimler sağlarken bir test paketinin kalitesini değerlendirmede etkin olarak kullanılamaz.
- Bu değerlendirme kodun birim test etme yeteneği ile aynı durumdur
- ❖ Kapsam ölçümleri iyi bir olumsuz göstergə olmasına rağmen kötü bir pozitif göstergedir.

## 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 ölçüdü değildir.

# Code Coverage

$$\text{Code coverage (test coverage)} = \frac{\text{Lines of code executed}}{\text{Total number of lines}}$$

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

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

1 Covered by the test

2 Not covered by the test

Kod kapsamının hesaplanması

Fonksiyonun toplam satır sayısı beşir.

Test tarafından çalıştırılan satır sayısı dörttür.

Test, true dönüşü dışında tüm kod satırlarından geçer.

Bu da  $4/5 = 0.8 = \%80$  kod kapsamı sağlar.

# Code Coverage

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

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

Kod değiştiğinde ne olur?

Test artık üç kod satırını da kullandığından, kod kapsamı %100 olur.

Kapsama değerleri kolaylıkla değiştirilebilir.

Kod ne kadar kompakt olursa, test kapsamı metriği o kadar iyi olur. Çünkü sadece satırlar hesaplanır.

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

# Branch Coverage Metric

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);
}
```

$$\text{Branch coverage} = \frac{\text{Branches traversed}}{\text{Total number of branches}}$$

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 branch sayısını değerlendirir.  
Bu dalları uygulamak için kaç satır kod gereği 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.

# Son sonucu kaydeden IsStringLong yöntemi

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

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

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

1  
2

3

Yandaki kod, **IsStringLong** yönteminin başka bir sürümüdür.

Son sonucu genel bir **WasLastStringLong** fonksiyonuna son sonuç kaydeder

1 First outcome

2 Second outcome

3 The test verifies only the second outcome.

\*ENCAPSULATION

declare fields/variables as private

provide public get and set methods, through properties, to access and update the value of a private field

# İrdeleme

## son sonucu kaydeden IsStringLong yöntemi

IsStringLong yönteminin iki sonucu vardır:

Explicit Sonuç: Dönüş değerinin, **return** anahtar sözcüğü ile belirlenen sonuç ve

Implicit Sonuç: İlgili özelliğin yeni değeri olan örtük bir değeri, yani önceki satırdaki public olan WasLastStringLong property (C# Properties (Get and Set)).

Örtük (implicit) sonuç doğrulanmamasına rağmen, kapsam metrikleri yine de aynı sonuçları verir:

Code Coverage %100

Branch Coverage %50

Bu irdelemeden şu sonuç çıkarılır:

Coverage metrics (kapsam ölçümleri), incelenen kodun tümüyle test edildiğini garanti etmez, sadece kodun çalıştırıldığını (yürüttüğünü) garanti eder. Çünkü örtük sonuç test edilmemiştir.

# Assertion-Free Testing

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

- ❑ code ve branch coverage metriklerinin ikisi de %100 olarak sağlanır.
- ❑ Faydası olamayan bir test yaklaşımıdır, herhangi bir şey doğrulamaz.

# %100 Coverage Gereksinimleri Değiştirir mi?

- ❑ İstenilenleri elde etmek için tüm test try/catch blokları ile yapışırsa testlerin geçmesi garantidir. Ama bu tür testler herhangi bir «assertion» uygulamaz.
  - ❖ Bu tür testler projeye hiçbir değer katmaz.
- ❑ Tüm çaba (effort) ve zaman (time) ve testlerin başarısını (ilerlemesini) sağlamak için gereken bakım (maintain) maliyetleri projeye zarar verir.
- ❑ Gereksinim %90 , daha sonra %80 ve zamanla tamamen çekilebilir.
  - ❖ Ama test edilen kod her sonucu tamamen doğrular.
  - ✓ İyi mi??

# Kapsam Metriği (Coverage Metric) external library kullanır mı?

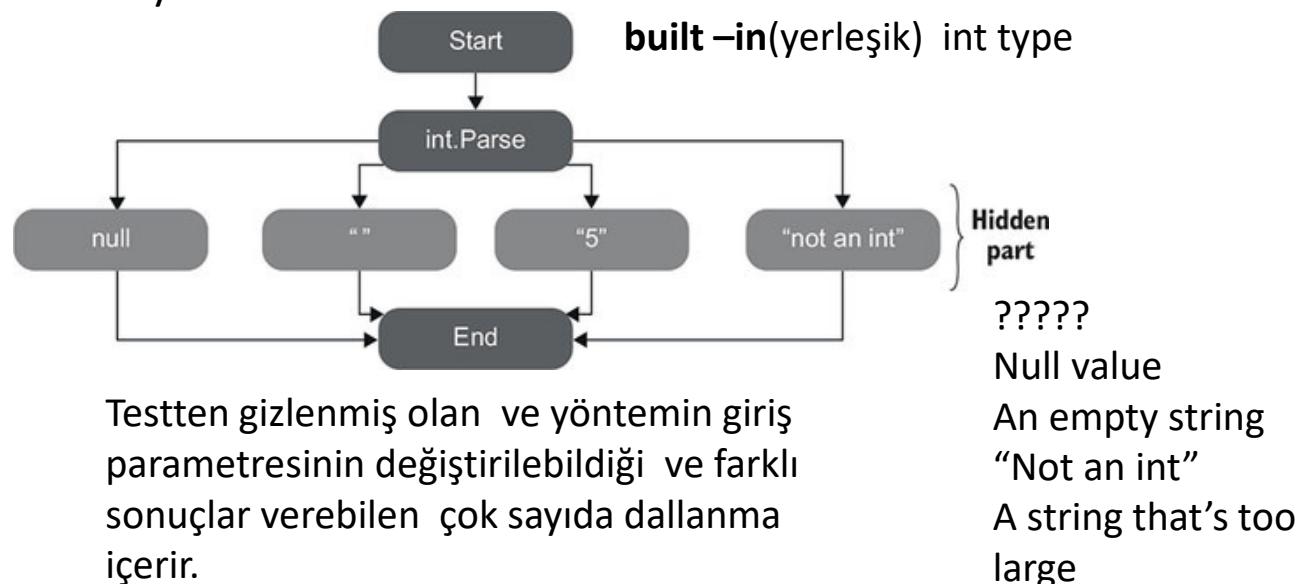
```
public static int Parse(string input)*  
{  
    return int.Parse(input);  
}
```

```
public void Test()  
{  
    int result = Parse("5");  
    Assert.Equal(5, result);  
}
```

\*.NET Framework int.Parse

Test sistemi ilgili yöntemleri çağrıduğunda kütüphane fonksiyonlarına ait kod yollarını hesaba katmaz.

Bu nedenle de kapsam metriklerinin, kaç tane olduğu ve testlerin kaç tanesinin uyguladığını görmemin bir yolu yoktur.



# API Tasarımı, Değiştirilmesi ve Testi

- ❑ API nasıl değiştirilebilir?
- ❑ Değişiklikler istemci kodunu bozar mı?
- ❑ API kullanıcıları geliştiriciyi nasıl etkiler?

Örneğin;

- ❑ API sınıflarından biri «internal» olarak kendi yöntemlerinden birini kullanıyorsa, bir kullanıcı bu sınıfı alt sınıflara ayırıp «override» ile çalıştırabilir; bu da pek çok şeyi değiştirebilir.
- ❑ Kullanıcı sınıfa farklı bir anlam verebilir, bu da yöntemin değiştirilmesini engeller.
- ❑ «Internal» tanımlanan uygulama seçeneklerinin değişikliği, kullanıcının inisiyatifindedir.
- ❑ Bu sorunu çeşitli şekillerde çözülebilir.
  - ❖ En kolay yol API kilitlemektir.
    - ✓ Java'da sınıflar ve yöntemler olabildiğince «final» yapılır.
    - ✓ C#'da sınıflar ve yöntemler «sealed» tanımlanır

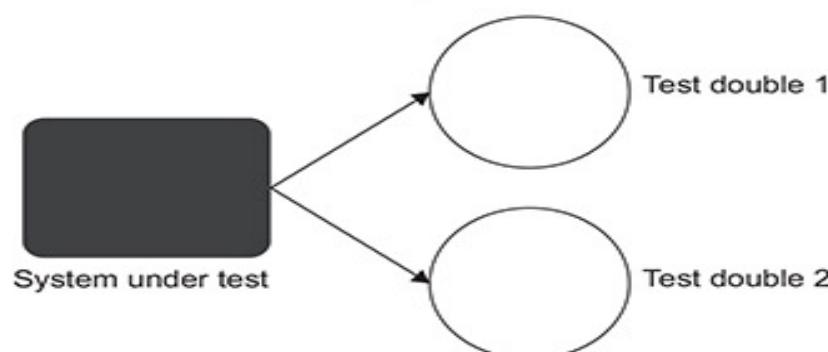
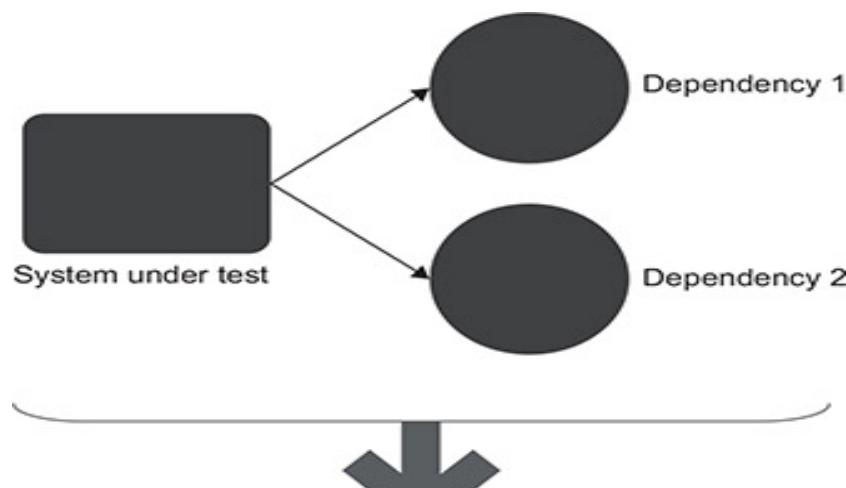
# API Tasarımı, Değiştirilmesi ve Testi

- ❑ Kullanılan dilden bağımsız olarak , API bir «singleton» şablon olarak yazılabilir ve/ veya «factory» şablon ile, yazılmak istenilenlerin kullanıcı tarafından değiştirilmesi önlenebilir.
- ❑ Uygulamada sorun çözülür mü?
  - ❖ Son yıllarda birim testi uygulamanın API yazmaya son derece önemli bir parça olduğu anlaşıldı ama önemseniyor mu?
- ❑ Üçüncü taraf API kullanan rastgele bir sınıf alarak birim testleri yazılsın.
  - ❖ API' nin kullanıldığı kod test koduna sıkı sıkı yapışır.
  - ❖ Kodun API sınıfları ile etkileşimlerini görebilmek veya kodun testi için dönüş değerlerini alabilmek için API sınıflarının kişiselleştirilmesi gereklidir.

## API Tasarımının Altın Kuralı

- ❑ Geliştirilen bir API için testler yazmak yeterli değildir.
- ❑ API nin kullanıldığı kod için de birim testleri yazılmalıdır.
  - ❖ Böylece kullanıcılar kodlarını bağımsız olarak test etmeye çalışıklarında ne ile karşılaşacakları öğrenilir.
- ❑ API geliştiricilerinin kullanan kodu test etmesini kolaylaştırmmanın tek bir yolu yoktur.
  - ❖ static, final ve sealed , iyi betimlemeler olarak yararlı olabilir.
- ❑ Test konusunda farkındalık önemlidir ve bunun için de deneyim gereklidir.
  - ❖ Deneyimlendiğinde diğer herhangi bir tasarım zorluğu gibi yaklaşılabilir.

# Birim Test nedir?



SUT, yani test edilen sistemin bağımlılıkları vardır. Sistem bu bağlantılarından (bağımlılıklardan) nasıl ayrılabilir?

Test edilen sistemin bağımlılıklarını test çiftleriyle (test double) değiştirmek, yalnızca test edilen sistemin doğrulanmasına odaklanmayı sağlar. Aksi takdirde büyük olan birbirine bağlı büyük nesne grafiğinin bölünmesi gereklidir.

Test double, piyasaya sürülmeye hazır gibi davranışan, ama karmaşıklığı azaltan ve testi kolaylaştıran basitleştirilmiş bir sürüm olan nesnedir (Gerard Meszaros, 2007)

## Birim Testi Yazılması Yaklaşımı : 3A

### 3A Şablonu

Testin (Pattern) « Arrange- Act- Assert» şeklinde gerçekleştirildiği bir yaklaşım vardır.

### Test edilecek kod:

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

## İyi Kod Yazmak İçin bir Şablon: 3A

- ❑ **Arrange** (Düzenleme) bölümünde, test edilen sistem (System Under Test, SUT) ve ilişkili olduğu bildirimler istenen duruma getirilir.
- ❑ **Act** bölümünde, SUT üzerindeki yöntemler çağrılar, hazırlanan bildirimler iletilir ve çıktı değerleri (eğer varsa) elde edilir.
- ❑ **Assert** bölümünde sonuç doğrulanır. Sonuç, dönen değerdir (retured value) ve SUT'in sonuçlanan (final ) bir durumudur veya SUT'nin bağlı olduğu kod parçalarına ait yöntemdir.

## unit test - 3A yaklaşımı: Sum yöntemi

```
public class CalculatorTests
{
    [Fact]
    public void Sum_of_two_numbers()
    {
        // Arrange
        double first = 10;
        double second = 20;
        var calculator = new Calculator();

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

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

- 1 Class-container for a cohesive set of tests
- 2 xUnit's attribute indicating a test
- 3 Name of the unit test
- 4 Arrange section
- 4
- 4
- 5 Act section
- 6 Assert section

## unit testi 3A yaklaşımı - Python : Mutlak Değer Fonksiyonu

```
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+programming&format=json'

    # Act
    response = requests.get(url)
    body = response.json()

    # Assert
    assert response.status_code == 200
    assert 'Python' in body['AbstractText']
```

## Feature testinde 3A İşleyışı

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

# Behavior-Driven Development & Test

- Arrange-Act-Assert şablonu bu yaklaşımada (Behavior- Driven Test) Given-When-Then olarak değişir
- Gherkin dili Given-When-Then adımlarını kullanır.
  - ❖ Böylece test senaryolarının davranışları betimlenir.

## Aslında

«Given-When-Then» ile «Arrange-Act-Assert» benzer test modelleme yapılarıdır.

The Biggest Tech Fails of 2021

<https://www.pcmag.com/news/the-biggest-tech-fails-of-2021>

230 of the biggest, costliest startup failures of all time

<https://www.cbinsights.com/research/biggest-startup-failures/>

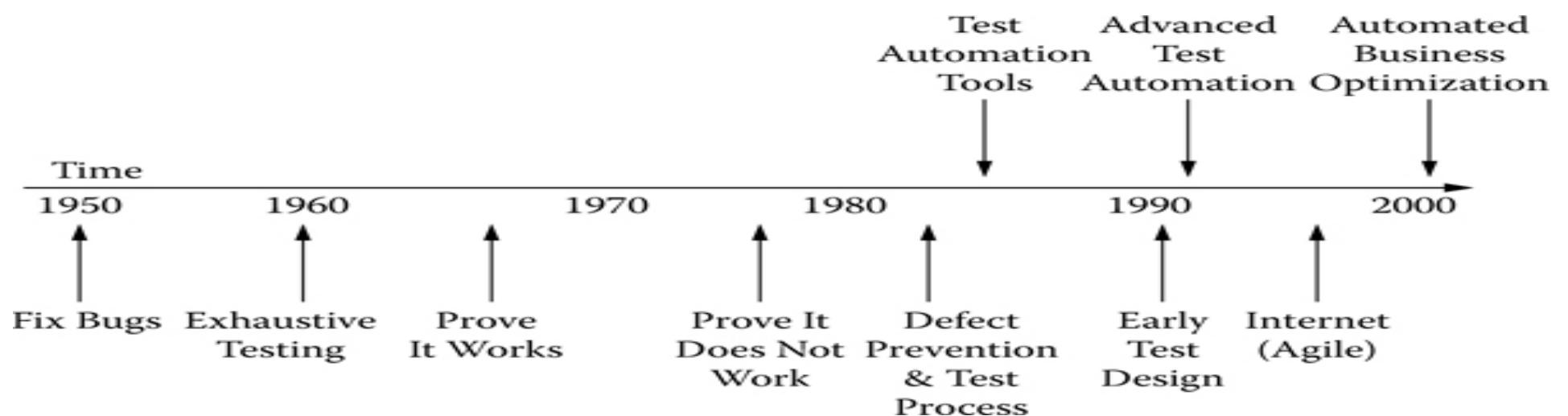
# YAZILIM KALİTE GÜVENCESİ VE TESTİ

DERS NOTU 3

2022 GÜZ

# Yazılımın Testi ve Kalite Güvencesinin Sağlanmasının Tarihsel Gelişimi

# Test Proseslerinin Gelişimi



## Doğruluğun Kanıtı (Proof of Correctness)

- ❑ 60'lı ve 70'li yılların test yöntemidir.
- ❑ Bu yaklaşım bugün hala kabul testi (acceptance test) olarak uygulanmaktadır.
- ❑ Pratikte zaman kaybı ve verimsiz (insufficient) bir yaklaşık olarak değerlendirilir.
- ❑ Basit testler için, yazılımın "çalıştığını" ve teorik olarak çalışacağını kanıtlamak kolaydır.
  - ❖ Yazılımların çoğu bu yaklaşım kullanılarak test edilmediğinden, gerçek uygulamanın implementasyonu (actual implementation) sırasında karşılaşılabilen çok sayıda «defect» ortaya çıkması olasıdır.

# Geleneksel (Traditional) ve Bugünün Testleri

- ❑ Geleneksel testler (1980'lerin başına kadar), bugünün sistem testi idi.
  - ❖ Çalışan kodun müşteriye teslimi sonunda sisteme neler yapıldığı, diğer bir ifade sistemin neler yaptığı idi
- ❑ Bugün ise test süreci, test uzmanın yazılım geliştirme yaşam döngüsünün tümünü testin her aşamasına eklemesi gereği bir süreçtir.
  - ❖ Kod testi kodun yazılması tamamlandıktan sonra gerçekleştirilebilir, ancak herhangi bir yanlışlık varsa, önceki geliştirme aşamalarının araştırılması gereklidir.
  - ❖ Hata bir tasarım belirsizliğinden veya bir programcıdan kaynaklanıyorsa, ürünün kullanıcıya teslimine kadar beklemek yerine sorunları oluşur olmaz bulmaya çalışmak daha kolaydır.
  - ❖ Araştırmalar, hataların yaklaşık yarısının gereksinimlerin belirlenmesi aşamasında (yazılım ürününün ne yapmasını istiyoruz?) veya tasarım aşamalarında olduğunu göstermektedir.
    - ✓ Bu aşamalardaki hatalar arttırıcı bir etki oluşturarak kodlama sırasında daha fazla hata oluşabildiği de geçektir.

## Exhaustive ( Kapsamlı) Test

□ 60 'lı yıllarla birlikte yazılımın kodlama boyunca olası yollarının kapsamlı (exhaustive) testine veya olası girdi veri varyasyonlarının tümünün sıralanışına önem verildi.

O yıllarda bile bir uygulamayı tamamen test etmek imkansızdı.

- ❖ program girdilerinin etki alanı çok büyük olabilir
- ❖ çok fazla olası giriş yolu vardır
- ❖ tasarım ve spesifikasyon sorunlarının test edilmesi zor olduğu saptandı.
- ❖ Bu nedenle de kapsamlı (exhaustive) testler o dönemde bile dikkate alınmadı ve teorik olarak olanaksız olduğu kanıtlandı .

## Slayt 6

---

**ZA1**

Zeynep Altan; 11.10.2020

## 80'li ve 90'lı yıllarda Yazılım Testi

- ❑ 80'li yıllar otomatik tool kullanılarak testin gerçekleştirildiği dönemlerdir.
- ❑ Hedef uygulamanın verimliliği (efficiency) ve kalitesinin (quality) artırılması hedeflenmeye başlamıştır.
- ❑ Başlangıçta basit olan bu test kitleri (tools), 90'lı yıllarla birlikte gelişti ve «early test design» kavramı doğdu.
  - ❖ Test kavram olarak "testleri ve test ortamlarını planlama, tasarlama, oluşturma, sürdürme ve yürütme" olarak yeniden tanımlandı.
  - ❖ Bu yazılım testinin bir kalite güvence bakış açısı olarak test edilebilirliğin yaşam döngüsüne sahip olduğunun tanımı idi.
  - ✓ Özetle iyi tanımlanmış bir test yönetilen bir süreç olarak ifade edildi.

## 90'lı yılların ortaları ve Agile (Çevik) Testler

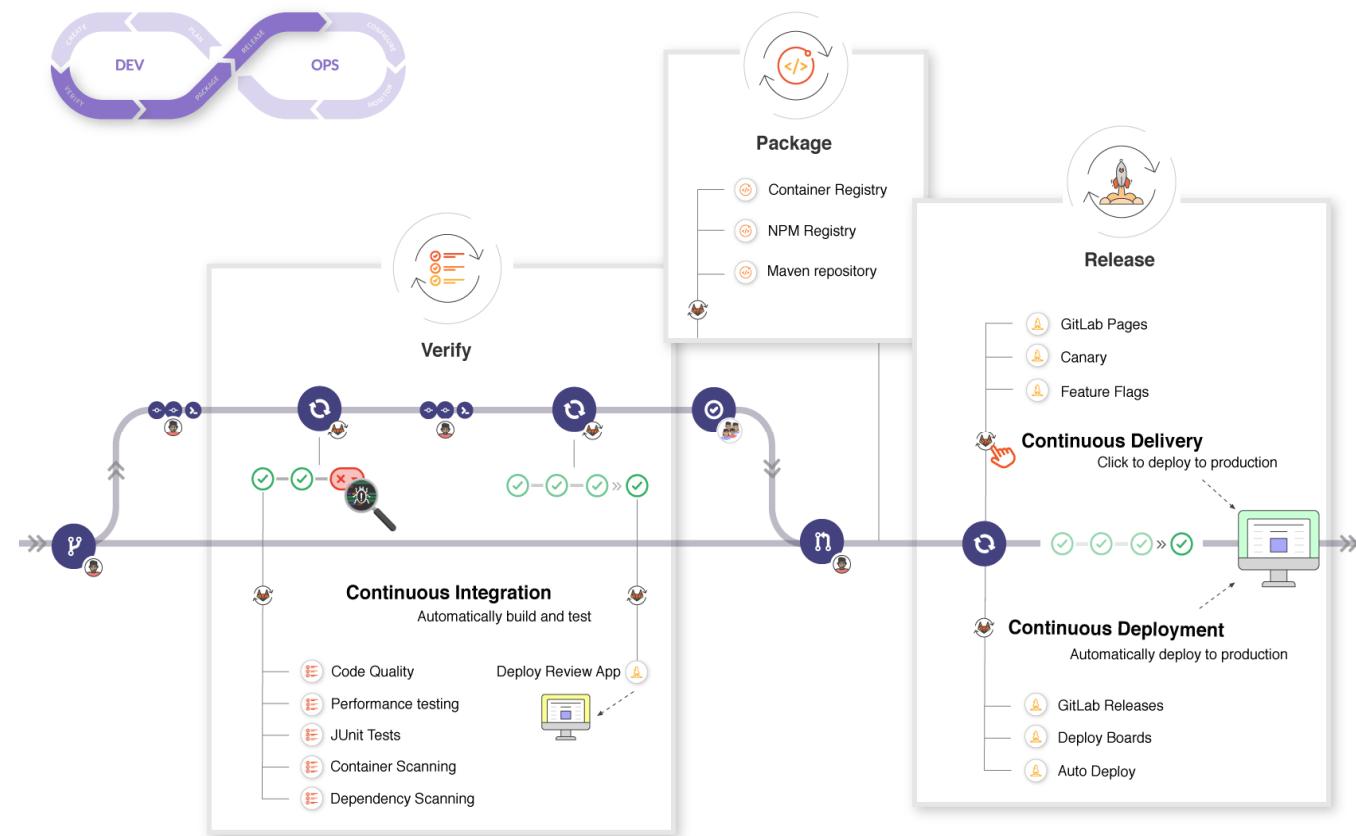
- ❑ 90'lı yılların ortalarına kadar test kavramı yazılım geliştirme yaşam döngüsünün bir süreci idi.
- ❑ Daha doğrusu 90'lı yılların ortalarına kadar standart bir test modeli olmadan yazılım geliştiriliyordu.
  - ❖ Bu a yazılım ürününün testini zorlaştıryordu.
- ❑ Bu dönemde test edilmesi gereken her şeyin önceden açık olarak tanımlanmasına gerek olmadığı görüldü.
  - ❖ Böylece gözde geçirilen her adımda beklenen sonucunu tanımlamaya gerek yoktu, belgelerin gözden geçirilmesi yeterli idi .
- ❑ Bu test yaklaşımları "çevik test" olarak adlandırıldı.
  - ❖ Bu test tekniklerden bazıları, keşif (*exploratory*) testleri, hızlı (*rapid*) testler ve risk odaklı (*risk based*) testlerdir.

## Yeni Metodolojilerde Test

- Test, kısa sürüm döngülerine uygun olmalıdır.
  - ❖ Her bileşen, her yeni artış için anında tekrarlanabilen yeniden kullanılabilir test senaryoları gerektirir.
  - ❖ Aksi durumda «increment» gerçekleştikçe sistem güvenilirliğinin düşüşe geçme riski vardır.
    - ✓ Her artış(increment), herhangi bir ek işlevi kapsayan yeni test senaryoları gerektirir
- Test otomasyonu, test çevik geliştirmeye uyarlanırken önemli bir araçtır.

# DevOps

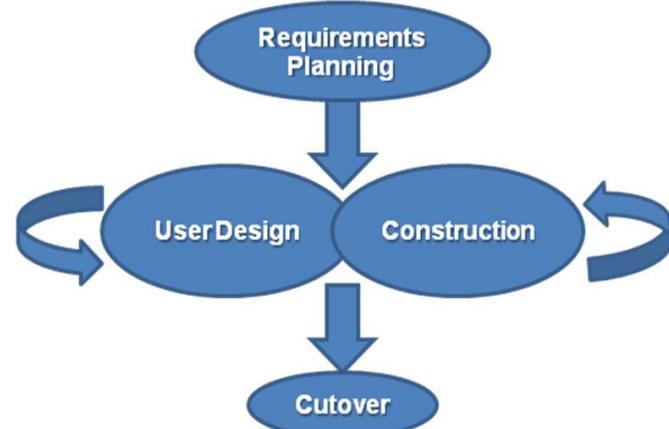
## Continuous Integration & Continuous Deployment



- ❑ DevOps, yazılım geliştirme (Dev) ve IT işlemlerini (Ops) birleştiren bir dizi uygulamadır.
- ❑ Sistem geliştirme yaşam döngüsünü kısaltır ve yüksek yazılım kalitesiyle ürünün sürekli teslimatı sağlanır.
- ❑ DevOps, Çevik (Agile) Ürün Geliştirme özellikler içerir.

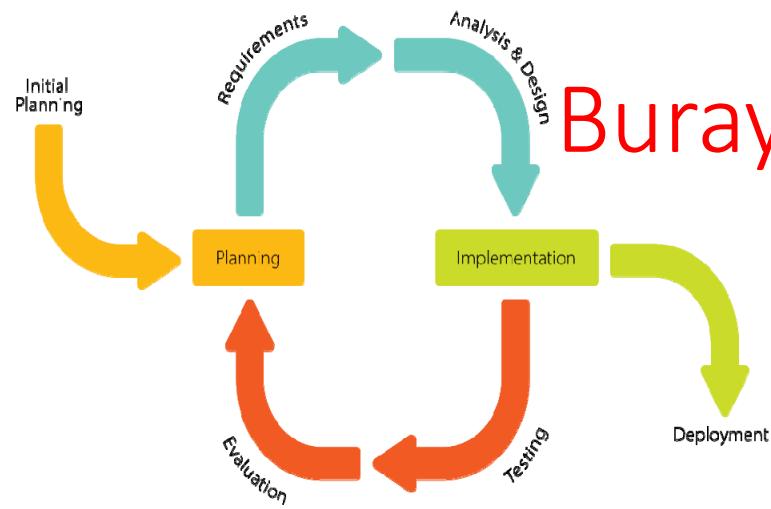
<https://docs.gitlab.com/ee/>

## Iterative/Incremental Development



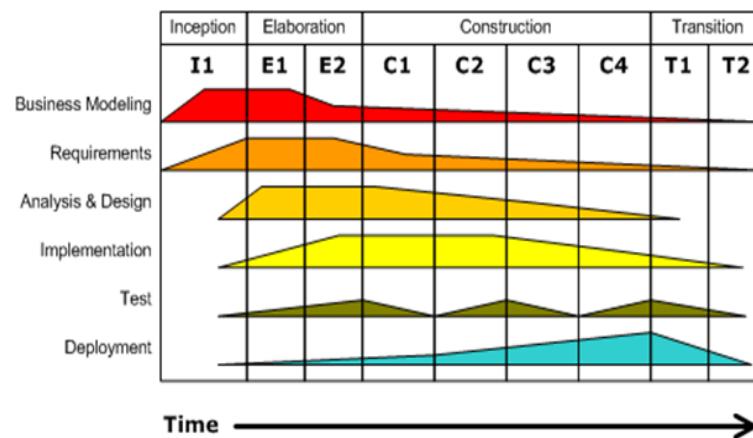
i) Rapid Application Development (RAD)

James Martin 1991



### Iterative Development

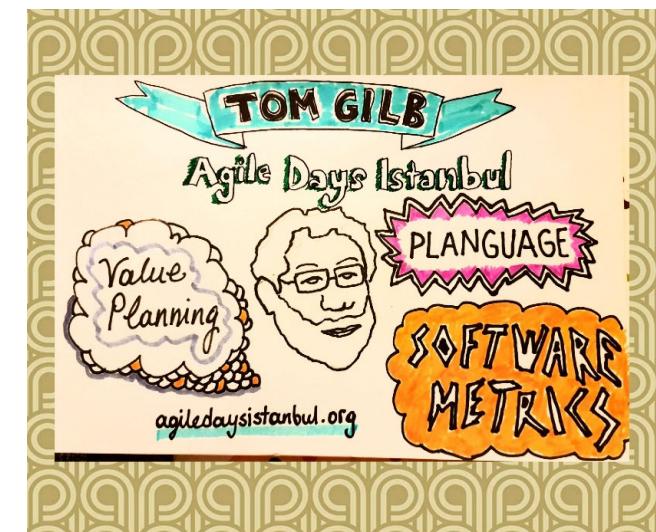
Business value is delivered incrementally in time-boxed cross-discipline iterations.



ii) Rational Unified Process (RUP)

Phillip Kruchten 2003

# Buraya Nereden Gelindi?



Evolutionary Development  
Tom Gilb 2005

# Birim Testi / Unit Testing

Arrange-Act-Assert (Test Driven Development)

Given-When-Then (Behaviour Driven Development) Gherkin Dilı

## unit testi 3A yaklaşımı - Python : Mutlak Değer Fonksiyonu

```
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+programming&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 çağrıılır. “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.

# Test Frameworks

Her programlama dilinin en az bir test framework tercihi vardır.

Örneğin:

JUnit, NUnit, Cucumber, and pytest testin otomatikleştirilmesini, gerçekleştirilmesini (execute) ve sonuçların raporlanmasıyı sağlar.

Framework bir test durumunu (test case) «good» ya da «bad» yapmaz.

# JustMock ile 3A Unit Test

Şablon kullanıldığı örnek bir depo ve depoya bağlı sipariş nesnesinin kodu aşağıdadır. Depo, farklı ürünlerin stoklarını tutar. Sipariş, ürün ve miktarını içerir.

Depo arayüzü ve sipariş sınıfı kodu:

```
public delegate void ProductRemoveEventHandler(string productName, int quantity);
public interface Iwarehouse
{
    event ProductRemoveEventHandler ProductRemoved;
    string Manager { get; set; }
    bool HasInventory(string productName, int quantity);
    void Remove(string productName, int quantity); }
public class Order
{
    public Order(string productName, int quantity)
    {
        this.ProductName = productName;
        this.Quantity = quantity; }}
```

```
public string ProductName { get; private set; }
public int Quantity { get; private set; }
public bool IsFilled { get; private set; }

public void Fill(Iwarehouse warehouse)
{
    if (warehouse.HasInventory(this.ProductName, this.Quantity))
    {
        warehouse.Remove(this.ProductName, this.Quantity);
    }
}

public virtual string Receipt(DateTime orderDate)
{
    return string.Format("Ordered {0} {1} on {2}", this.Quantity, this.ProductName,
orderDate.ToString("d"));
}
```

## 3A: Arrange

Önce siparişin verilmesi gereklidir.

```
var order = new Order("Camera", 2);
```

Depo oluşturulur (mock)

```
var warehouse = Mock.Create<IWarehouse>();
```

Gerçekten depoda 2 ürün var mıdır?

```
Mock.Arrange(() => warehouse.HasInventory("Camera", 2)).Returns(true);
```

**İstenilen değeri feri döndürürse bu aşama başarı ile tamamlanmıştır.**

## 3A ile Unit Test - Act

.

Depoya sipariş girilir.

```
order.Fill(warehouse);
```

İstenen eylem gerçekleştirildikten sonra (executed) ve başarıyla tamamlandığında ve siparişin gerçekten verildiğinden (fill) emin olunması gereklidir.

Depoda gerçekten 2 kamerası envanteri (stok) vardır.

# 3A ile Unit Test – Assert VisualStudio

## **Assert sınıfı**

Microsoft.VisualStudio.TestTools.UnitTesting

Microsoft.VisualStudio.QualityTools.UnitTestFramework

Namespace içerisindedir.

TestProject oluşturulduğunda Visual Studio otomatik olarak bir referans parametresi ekler.

IsFilled fonksiyonu ile siparişin doğru olup olmadığı kontrol edilir.

```
Assert.IsTrue(order.IsFilled);
```

## Verify Interaction

Deponun HasInventory yöntemi belirli parametrelerle çağrıldığında true döndürmesi sağlandı .

Ancak bu yöntemin gerçekten çağrılmış mıdır?

Bunun için **Arrange** yöntemi değiştirilebilir. ve depoyu işaretlenerek **HasInventory** çağrıılır.

```
Mock.Arrange(() => warehouse.HasInventory("Camera", 2)).Returns(true).MustBeCalled();
```

Bunun doğrulanması (verification) için Assert aşamasında warehouse nesnesi parametresi ile Mock.Assert çalıştırılmalıdır.

```
Mock.Assert(warehouse);
```

## Verify Order of Calls

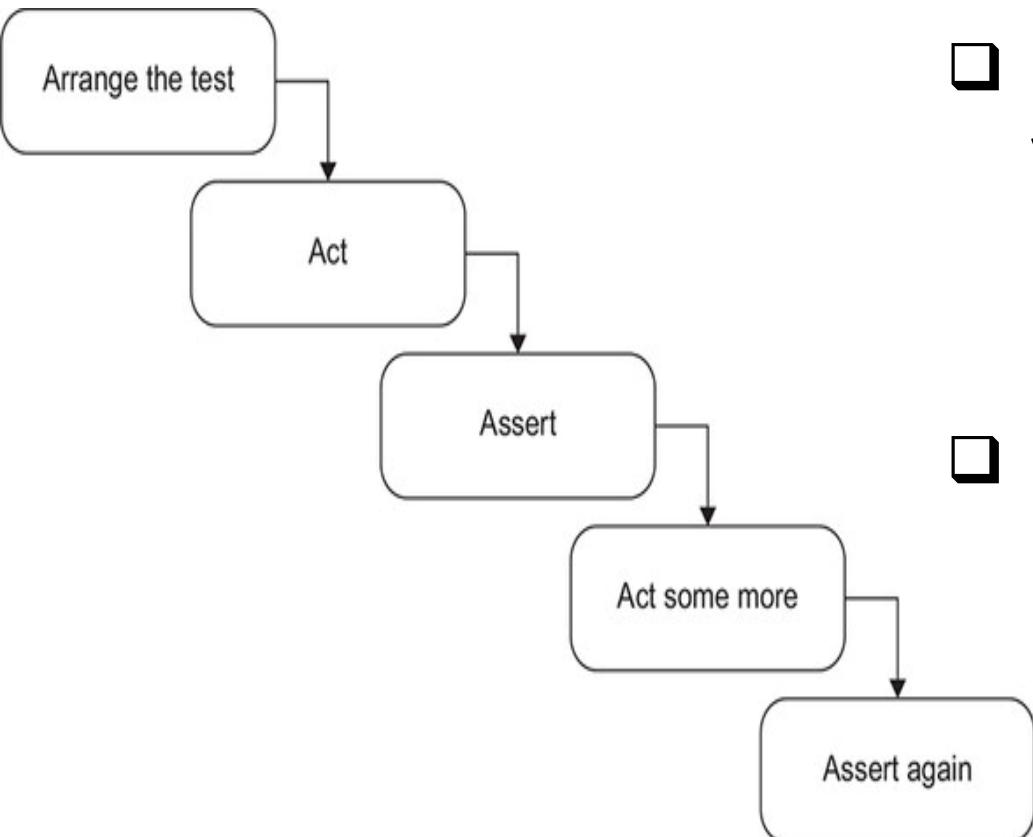
```
public interface IFoo
{ void Submit();
  void Echo();    }
[TestMethod]
public void
ShouldVerifyCallsOrder()
{ // Arrange
  var foo = Mock.Create<IFoo>();
  Mock.Arrange(() =>
    foo.Submit()).InOrder();
  Mock.Arrange(() =>
    foo.Echo()).InOrder();
// Act
foo.Submit();
foo.Echo();
// Assert
Mock.Assert(foo);  }
```

<https://docs.telerik.com/devtools/justmock/basic-usage/arrange-act-assert>

Düzenleme adımları linkten öğrenilebilir

# Test için 3A yaklaşımı

## Arrange, Act, Assert



- 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.

Tek bir eylem, testin birim testi alanında kalmasını sağlar

Test basittir, hızlı ve anlaşılmasının kolaydır.

Bir dizi eylem ve iddia (act and assert) içeren bir test , yeniden düzenlenmelidir. .

<https://docs.telerik.com/devtools/justmock/basic-usage/arrange-act-assert>

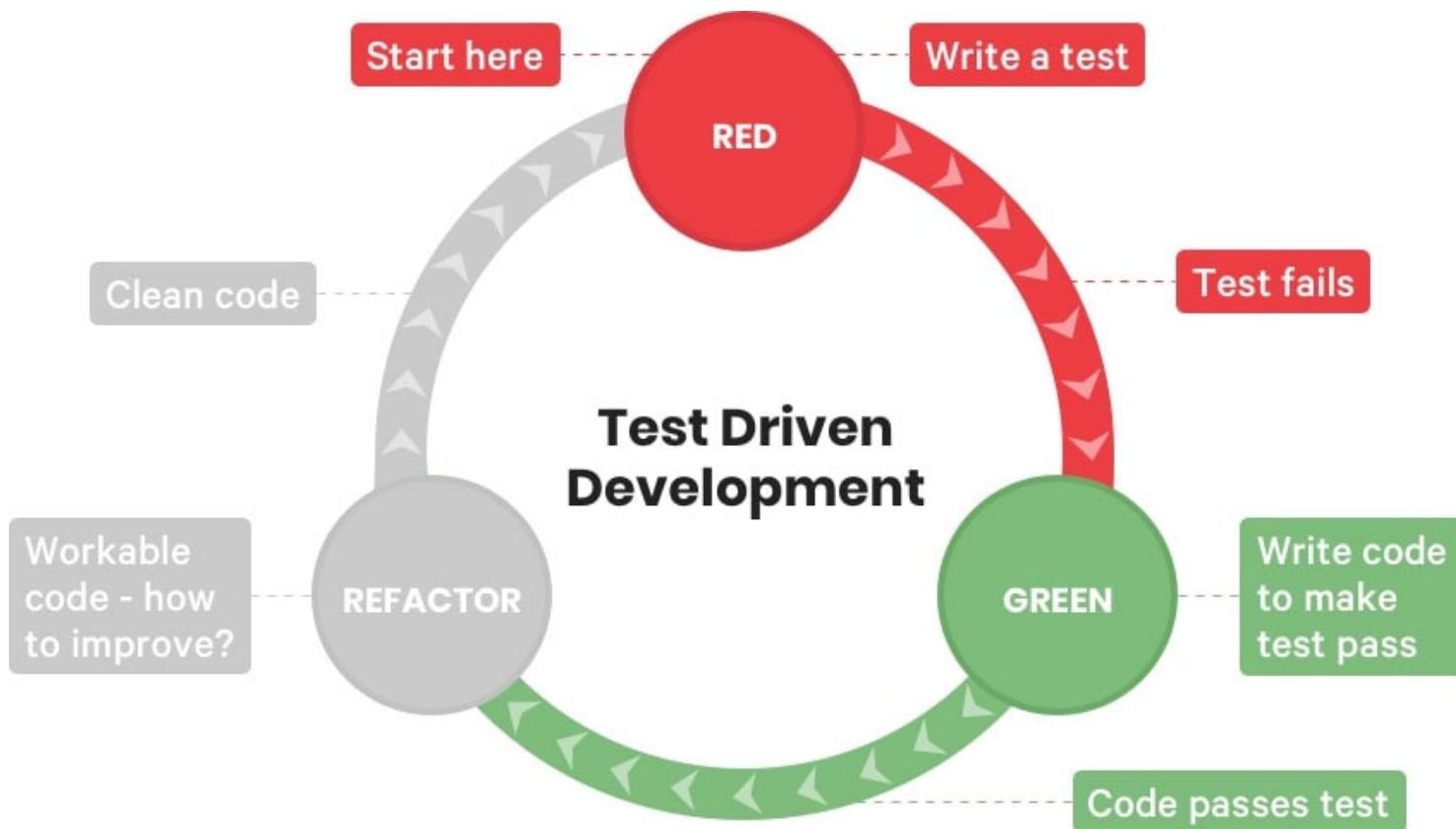
## TDD - Test Driven Development

- ❑ Test yazmaya genellikle «arrange»(düzenleme) bölümü ile başlanmasına rağmen, «assert» bölümü ile de başlamak mümkündür.
- ❑ Teste Dayalı Geliştirme (TDD) uygulandığında , yani bir özelliği geliştirmeden önce başarısız bir test oluşturulduğunda, yazılan kod parçasına ait özelliğin davranışını yeterince bilinmez.
  - ❖ Kod parçasının davranışından ne beklediğini belirmek ve bu bekleniyi karşılamak için sistemin nasıl geliştireceğini belirlemek gereklidir.

# TDD - Test Driven Development

- Problem çözümüne, ürünün geliştirilmesine önce test yazılarak başlanır.
  - ❖ Çözüme hedeflenen yapı hakkında düşünerek başlanır.
  - ❖ «Tanımlanacak olan davranış (aktivite) ile ne yapmak istenmektedir?» sorununun cevabı araştırılır.
  - ❖ Bazı geliştirici grubu için mantıksız değerlendirilse bile problemin asıl çözümü bundan sonra gerçekleşir.
- TDD, son yılların tercih edilen ürün geliştirme yaklaşımlarındanandır.
- Ürünün geliştirilmesi için kod yazılmasına beyanları, yani iddiaları (assertions) yazarak başlanır.
  - ❖ Böylece TDD yaklaşımı ile problemin çözüm koddan önce testi yazıldığında geçerli olur.
- OYSA:**
- Test yapılmadan önce problemin kodu yazılsa, teste geçildiğinde bu davranıştan ne bekleniği bilinmektedir.
  - ❖ O nedenle de önce kod yazıldığında teste arrange (düzenleme) bölümünden başlamak iyi bir seçenektır.

# TDD Development



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

# Test Driven Development - TDD

Write a test.

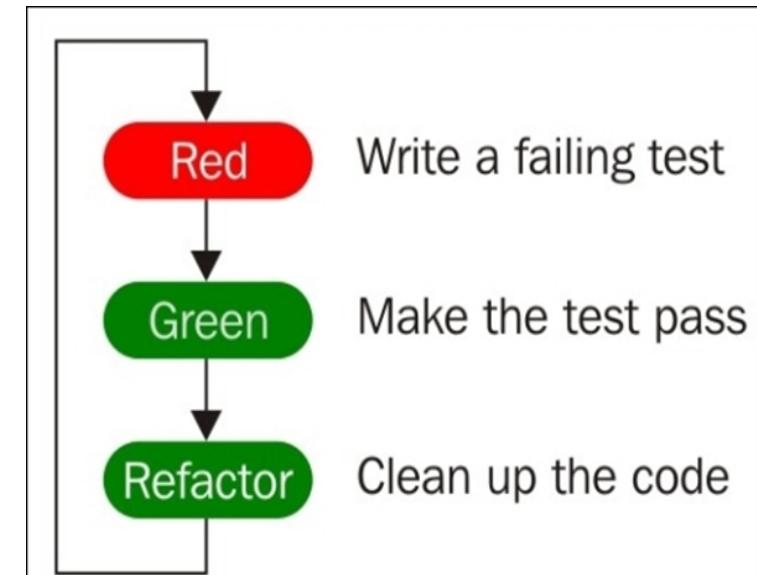
Run all tests.

Write the implementation code.

Run all tests.

Refactor.

Run all tests.



Implementasyon 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 işlevselligi uygulayan kod yazılmıştır. En iyi tasarım oluşturulmamıştır.
  - ✓ Sadece testi başarılı yapacak (pass) basit bir kod geliştirilir.

Bu tür testler kaldırılmalı veya kod yeniden düzenlenmelidir (refactored)

- ❖ Bu da, tasarımın geliştirilmesini, tasarımın daha okunabilir ve sürdürülebilir hale getirilmesini içerir.
- ❖ Önemli : Yeniden düzenleme aşamasında başlangıç tasarımını bozulmamalıdır.

Bir sonraki teste geçildiğinde ve sonraki işlevsellik parçası uygulanırken bu döngü tekrar edecektir.

# YAZILIM KALİTE GÜVENCESİ VE TESTİ

DERS NOTU 4  
2022 GÜZ

# Yazılım Kalite Güvencesi ve Testi Araştırma Ödevi 1

Teslim Tarihi: 11 Kasım çıktı olarak teslim edilecektir. Beykent Pusulada ödev yüklemesi (10 Kasım 23.59) değerlendirme sonuçlarının ilanı içindir.

Öncelikle belirtilmelidir ki bu süre yapılacak araştırmaların iki hafta devam edeceği kabulüne göre belirlenmiştir.

- 1) İyi bir Birim Testinin dört esası size göre nelerdir? Nedenlerini örnekler üzerinde de açıklayarak cevaplamınız beklenmektedir(25 puan).

Bu soruya vereceğiniz cevap yapacağınız araştırmanın derinliğine göre değişeceği için sizin için doğru olduğunu düşündüğünüz esasların paylaşılmaması özellikle önerilmektedir.

- 2) Martin Fowler ismi yazılım testi dünyasında öncü isimlerdendir. «Martin Fowler, Birim Testi (Unit Testing) ve Temiz Kod (Clean Code)» üçlemesini nasıl değerlendirirsiniz?

Önceki sorunun değerlendirme ölçütleri geçerlidir (25 puan).

- 3) Unit Test Frameworks ve çalışmaları ortamları araştırdıktan bunlarla ilgili olarak size göre nasıl bir sınıflandırma yaparsınız? Açıklayarak cevaplayın (15 puan).

- 4) Herhangi bir test framework indirerek işlevinin basit olmadığı bir kod parçasının birim testini 3A yaklaşımı ile yazın, gerekli açıklamaları yaparak ekran çıktılarını paylaşın. Ayrıca problem ifadesinin açık olarak verilmesi gerekmektedir (35 puan).

# Yazılım Kalite Güvencesi ve Testi Araştırma Ödevi 1

Araştırma süresince dikkat edilmesi ve eklenmesi gerekenler

- 1) 4. sorunun cevabı için GUI uygulamalarına ait birim testi yapılmamalıdır. Bu cevaplar değerlendirilmeyecektir.
- 2) Beykent Pusulada eklediğiniz ve ders günü çıktı olarak teslim ettiğiniz değerlendirme sonuçları vize sınavından önce Beykent Pusulada ilan edilecektir. Değerlendirme sonucunun %30 değeri vize sınavına ait değerlendirmedir.

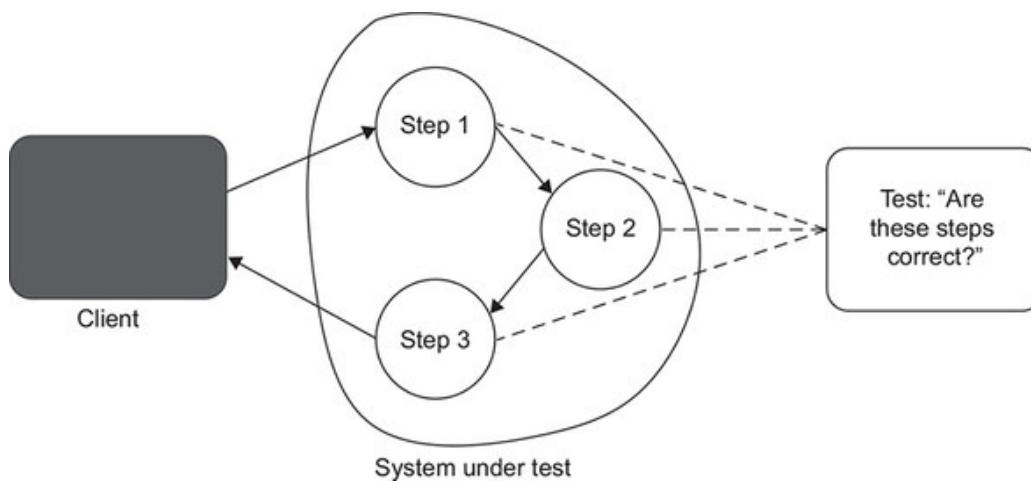
**Bu nedenle:**

- 3) Çalışmanın bireysel yapılması ve her bir sorunun sonunda yararlanılan kaynakların sizin için en önemli birkaç tanesinin verilmesi gerekmektedir.  
Her bir soruya cevap verebilmek için pek çok kaynak araştırması yapılabacağı kabul edilmektedir.
- 4) Soruların cevabı için sayfa sınırı yoktur. Fakat doğrudan ve karşılığı anlaşılmayan kopyala/yapıştır cevaplar kesinlikle değerlendirilmez.
- 5) Cevabınızı hangi soruya karşılık olduğunu anlayabilmek için numarası ve sorusu ile birlikte olmalıdır.
- 6) Bu ödev ayrıntılı bir araştırma çalışmasıdır. Yeterli araştırma yapmadığınız için emin olmadığınız cevapları eklememeniz önerilir.

# Yazılım Testinde «False Positive» ne demektir? (devam)

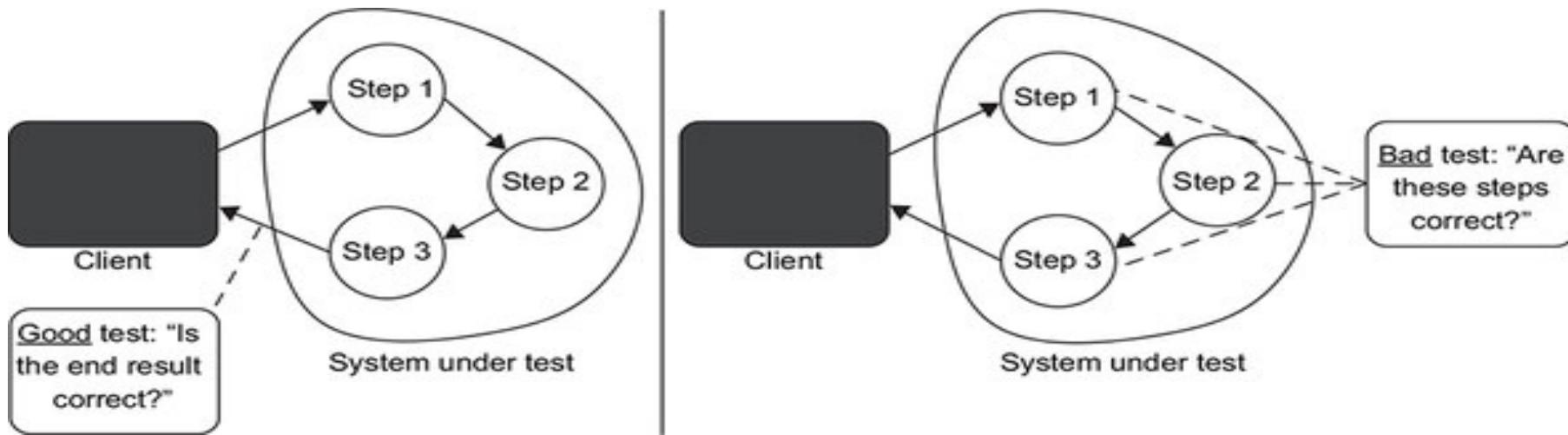
- Yazılım şirketinin yönetimi tarafından projenin önemli ölçüde değiştirilmesi ile geliştirme takımının buna göre değişiklikler yapması sonucu kullanılmayan kod (left over code) giderek artar ve bunlara olan bağımlılıklar nedeni ile bu kodlardan kurtulmak olası değildir.
- Proje için iyi bir test kapsamı (test coverage) uygulanmasına rağmen eski özellikler için refactoring (yeniden düzenleme) gerektiğinde yapılan testler başarısız olabilir.
  - ❖ Bunlar sadece uzun zaman önce devre dışı bırakılmış eski testler değil aynı zamanda yeni testlerde de görülebilir.
- Bu tür başarısızlıklar False Positive olarak sınıflandırılır.
- Karşılaşılan hatanın nedeni eski kod yığını ise «bu test devre dışı bırakılır; sonra bakılır» görüşü hakim olur.
  - ❖ Sonuç: Büyük bir hata devreye girene kadar her şey başarılı olarak çalışır.
    - ✓ Hata ile karşılaşılıp doğru tespit yapılmasına rağmen geliştiriciler tarafından dikkate alınmaz ve
    - ✓ Genellikle eski kod tamamen bırakılır.

# SUT'in Yeniden Düzenlemeye (Refactoring) Etkisi



- ❑ Böyle bir test, belirli bir implementasyonu, yani uygulamanın sonucunu elde etmek için SUT'nin atması gereken belirli adımların gerçekleşmesini bekler
  - ❖ Bu nedenle oldukça kırılgan, yani bağımlıdır.
- ❑ SUT'nin implementasyonu sırasında yapılması olası herhangi bir yeniden düzenleme bir test hatasına yol açabilir.

# SUT ve iki farklı yorumu



- ❑ Soldaki test, SUT uygulamasındaki ayrıntılarla değil de SUT'nin gözlemlenebilir davranışıyla eşlesir.
  - ❖ Böyle bir test, yeniden düzenlemeye de (refactoring) çok açık değildir.
  - ❖ Eğer varsa, çok az sayıda false positive tetiklenecektir.

## Regresyonlardan korunma ile yeniden düzenlemeye direnç arasındaki ilişki

Table of error types		Functionality is	
		Correct	Broken
Test result	Test passes	Correct inference (true negatives)	Type II error (false negative)
	Test fails	Type I error (false positive)	Correct inference (true positives)

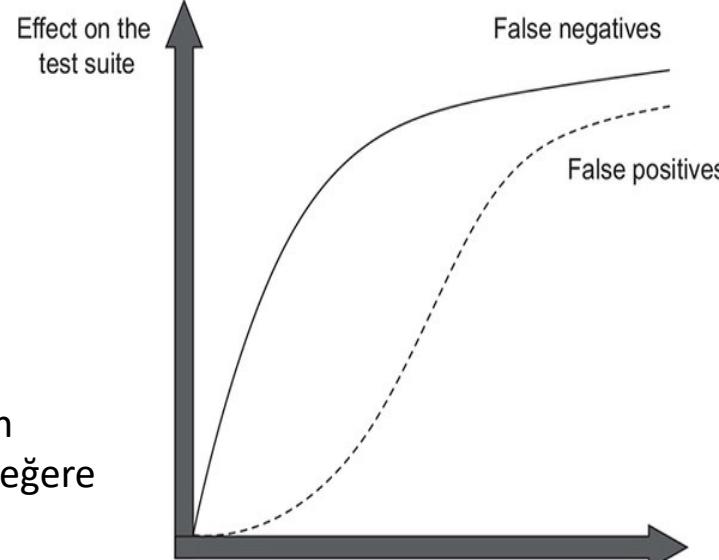
Protection  
against  
regressions

Resistance to  
refactoring

$$\text{Test accuracy} = \frac{\text{Signal (number of bugs found)}}{\text{Noise (number of false alarms raised)}}$$

Accuracy, sonuçların beklenen standart değere yakınlığıdır.

«false positive» ve «false negative» projeye etkileri

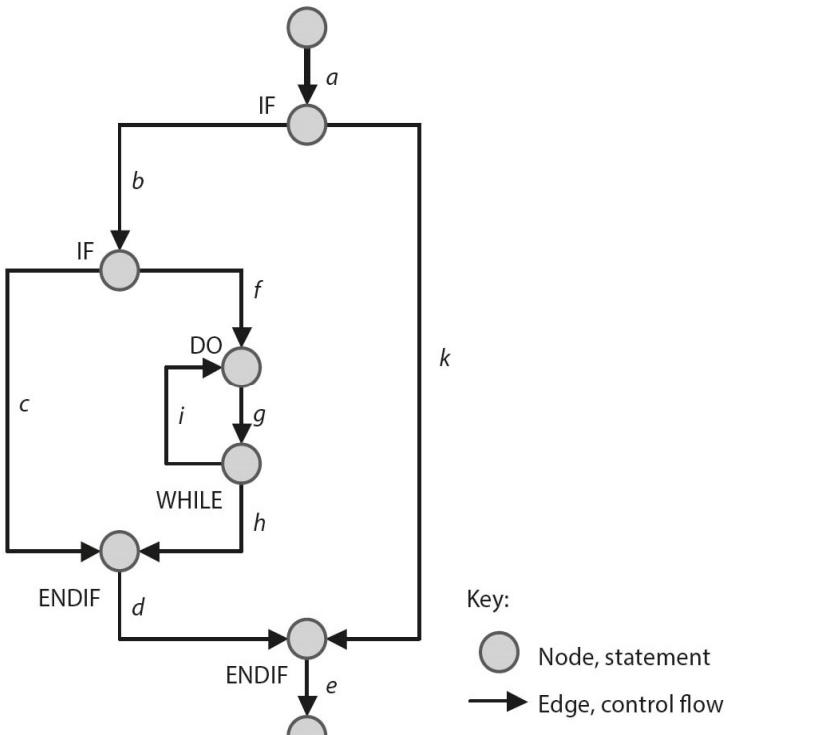


Testin doğruluğunu iyileştirmenin i) birinci yolu **payı**, yani **sinyali artırmaktır**. Bu, testi regresyon bulmada daha iyi hale getirmektir.

ii) İkinci yolu , **paydayı**, yani **gürültüyü azaltmaktadır**. Bu, yanlış alarm vermemek için testi daha iyi hale getirmektir.

Project duration

# Statement Testing & Coverage (devam)



Kontrol Akış Diyagramı  
Control Flow Diagram

- ❑ No information on statements that aren't executed
  - ❖ Kontrol akış diyagramı, işleme girmeyecek kısımlarla ilgili bilgi içermez
    - ✓ Çünkü testin amacı sadece testin implementasyonudur.
- ❑ Test senaryoları çalıştırıldıktan sonra, her bir ifadenin çalıştığı doğrulanmalıdır.
  - ❖ İstenilen «coverage» derecesi sağlandığında, test tamamlanmıştır.
  - ❖ Çalıştırılmamış (not executed) bir ifadenin çalışma koşullarını istenilen şekilde yorumlayabilmek mümkün olmaz.
    - ✓ Bu nedenle ifadelerin tümünün (%100) çalıştırılması önemlidir.

# Statement Testing and Coverage

## Test Case yada (Test Senaryosu)

*Node coverage in the control flow diagram* (kontrol akış diyagramındaki düğümler esastır)

Tüm düğümlere (yani ifadelere (statements)) **tek bir «test case»** ile erişilmelidir.

Test case

a, b, f, g, h, d, e

Sıralanışında tüm kenarları (edges) dolaşır.

*A single test case is enough*

- Tüm kenarlar (edges) bir kez geçildiğinde (traverse) tüm ifadeler (statements) düğümlerle (nodes) bir kez çalıştırılmıştır.
- %100 «coverage» kapsama sağlayan başka sıralanışlar da olabilir.
  - ❖ Ama , bu testin yol gösterici ilkelerinden biri yapılacakları en aza indirmektir.
    - ✓ Önceden tanımlanmış «coverage» (kapsam) derecesi olabildiğince az test senaryosu ile gerçekleşmelidir.

**Sonuç:**

Beklenen (expected) davranış betimlemelere göre tanımlanır.

Testin yürütümü ile, herhangi bir sapmanın (divergence ya da failures) belirlenmesi için «expected result» ile «actual result» karşılaştırılmalıdır.

# Statement Coverage Testing

$$\text{Statement coverage} = \frac{\text{Number of executed statements}}{\text{Total number of statements}} * 100$$

```
print (int a, int b) {  
int sum = a+b;  
if (sum>0)  
print ("This is a positive result")  
else  
print ("This is negative result")  
}
```

**Test Case 1:**

a = 5, b = 4

Total number of statements = 7

Number of executed statements = 5

**Statement coverage** =  $5/7*100 = 500/7 = 71\%$

**Test Case 2:**

a = -2, b = -7

Total number of statements = 7

Number of executed statements = 6

**Statement coverage** =  $6/7*100= 600/7 = 85\%$

# C0 Coverage

- Statement coverage C0 (C-zero) coverage olarak ta adlandırılır.
  - ❖ «weak exit criterion» belirler.
- «100% statement coverage» sağlamak zor olabilir.
- Örneğin program «exceptions» içeriyordur ve bunları test sırasında oluşturmak kolay değildir, fazla çaba (effort) gerektirir.
- 100% coverage gerekiğinde ve bazı ifadelere erişmek ve test için çalıştmak zor olduğunda, erişilemeyen (unreachable) ya da ölü kod (“dead” code) olarak adlandırılır.

# Karar Testi

## Decision Testing

- ❑ Karar testi diğerine göre daha fazla «test cases» gereklidir.
- ❑ «White box» testlerde «statement coverage test» yaklaşımından biraz daha kapsamlı tanımlama ölçütleri «decision test coverage» ile tanımlanır.
- ❑ Kaynak kod ile ilgili olarak alınacak kararlar test sürecinin odağıdır.
  - ❖ Karar alındığında kararın etkisi belirlenir. Etki, bu karar bir sonraki adımda hangi ifadelerin çalıştırılacağını belirlenmiştir.
- ❑ Bu durum test süresince devam eder.
- ❑ Kod ait olarak alınan karar IF-CASE deyimleri, döngüler ve bunlara benzer ifadeler içindir.
  - ❖ Karar ise doğal olarak «true» veya «false» sonucunun döndürülmesidir.
    - ✓ Bu değerler daha sonra hangi çıktı yolunu (branch) izlemek gereği ve/veya bir sonra çalıştırılacak deyim için kullanılır

## Branch Testing

- ❑ Branch testinde atomik alt koşullar tanımlanır.
  - ❖ Bu da hem «true», hem de «false» değerler için koşulların birlikte test edilmesidir.
- ❑ Bir koşul AND, OR ya da NOT gibi mantıksal (logical ) koşullar içermez.
  - ❖ Ayrıntıya girer ve “>” , “<” ya da “==” gibi ilişkisel (relational) operatörleri içerir.
- ❑ Test nesnelerindeki (test cases) tek bir karar çoklu koşullardan oluşabilir.

# Branch Coverage Testing

«More test cases are required»

Örnekte 100% statement coverage : a, b, f, g, h, d, e olarak verilmiştir.

«c, i, k » kenarları «statement coverage» test case için çalıştırılmadı.

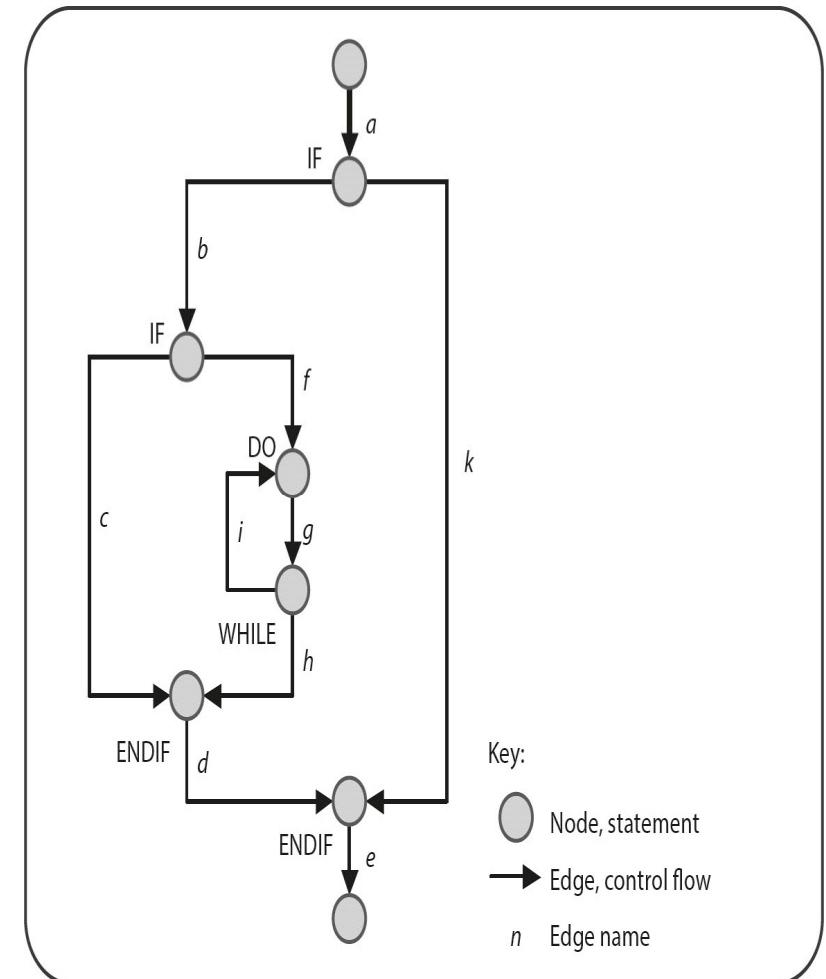
Burada:

c ve k kenarları (edges) ile false IF-statement, i kenar ile de döngünün başa dönmesi kontrol edilir. 100% branch coverage için üç «test case» gereklidir.

**Test case 1:** a, b, c, d, e

**Test case 2:** a, b, f, g, i, g, h, d, e

**Test case 3:** a, k, e



Control Flow Diagram

# «Decision» ve «Branch» Testinin Karşılaştırılması

Bu iki test arasındaki fark, iki teknikten elde edilen kapsama (coverage) dereceleridir.

## Karar Testi (Decision Test)

- Örnek olarak ELSE kısmı boş olan bir IF ifadesi verilmiş olsun:
  - ❖ "true" koşulu değerlendirilirse karar (decision) testi için %50 kapsama (coverage) elde edilir.
  - ❖ «false» için bir ifade içeriyorsa ilave bir test durumu (test case) gerekektir ve kapsama (coverage) değeri %100'e çıkar.

## Branch Test

Branch Testi, şekildeki kontrol akış şemasını «control flow graph» izlediği için farklı sonuç verir.

- THEN iki «branch» ve bir düğüm (node) oluşturmuş, ELSE ise düğüm (node) içermez.
  - ❖ Bu IF ifadesinde toplam üç «branch» vardır.
- "true" için test senaryosu çalıştırılırsa, Branch Test üç daldan ikisini kapsar ve %66.6 kapsama (coverage) sağlanır.
  - ❖ Oysa "decision" testinden "true" için %50 sonuç alınır.
- Problem "false" için bir ifade içerirse (bu örnekte yok) ve bu da test senaryosu olarak tanımlanırsa, %100 «branch coverage» elde edilecektir.

# C1 Coverage

- ❑ Decision coverage, C1 coverage olarak da adlandırılır.
  - ❖ Bir karar sonucunun yoluna (branch), yani tamamlanmış olup olmadığına bakılır.

## Her iki test sonuçlarının irdelenmesi

Verilmiş örnekte a'dan e'ye tüm sıralanışlar (branches) ve bunlarla ilgili karar çıktıları (decision outcomes), üç farklı «test case» için 3 defa baştan sona dolaşım ile gerçekleşir(traversed).

- ❑ Test case «k edge» içermediği kabul edilir. Bu durumda «k edge» hesaplamalara dahil değildir.
- ❑ Branch Coverage :  $(9/10) \times 100\% = 90\%$ 
  - ❖ Graf, kapsama (coverage) için toplamda 10 «branch output» içerir
    - ✓  $(2 \cdot 2^2 + 2 = 10)$
- ❑ Decision Coverage:  $(5/6) \times 100\% = 83.33\%$ 
  - ❖ 3 decision (karar) olmasına rağmen, 6 çıktı (output) vardır
    - ✓  $(3 \cdot 2 = 6)$
- ❑ Her iki test için aynı test senaryosu kullanılmıştır ve ara adımda elde edilen sonuçlar farklıdır
  - ❖ Oysa sonuç değeri (result output) olarak 100% kapsama ( coverage) hedeflenir.
- ❑ Elde edilen sonuçlar «statement coverage» ile karşılaştırıldığında, «statement coverage» ile tek bir test case ile 100% sonucuna ulaşılmıştır.

## Sonuç:

# «Decision» ve «Branch» Testlerinin Bağımlılığı

- «Decision Coverage» ve «Branch Coverage», «Structural Coverage Analysis» yapısında test teknikleridir ve birbirleriyle yakından ilişkilidir.
  - ❖ «Decision Coverage» DO-178B/DO-178C (Software Considerations in Airborne Systems and Equipment Certification) standartını,
  - ❖ «Branch Coverage» ISO 26262 (Road Vehicles Functional Safety Package) standartını kullanır.
- «Branch coverage», koşullu bir kaynak kodu ifadesinden her çıktıının (output) çalıştırılmasını gerektirir.
  - ❖ Bu nedenle, bir if ifadesi için «branch coverage», then kısmının ve else kısmının da çalıştırılmasını gerektirir
    - ✓ Eğer else kısmı yoksa, if ifadesi yine de kararı «true» ve «false» durumlarını uygulamalıdır.
- «Decision Coverage» koşullu ifadeleri, «Branch Coverage» ile aynı şekilde kapsar ve lojik (mantıksal) değişkenlerin atamalarını da içerir.

# YAZILIM KALİTE GÜVENCESİ VE TESTİ

DERS NOTU 5

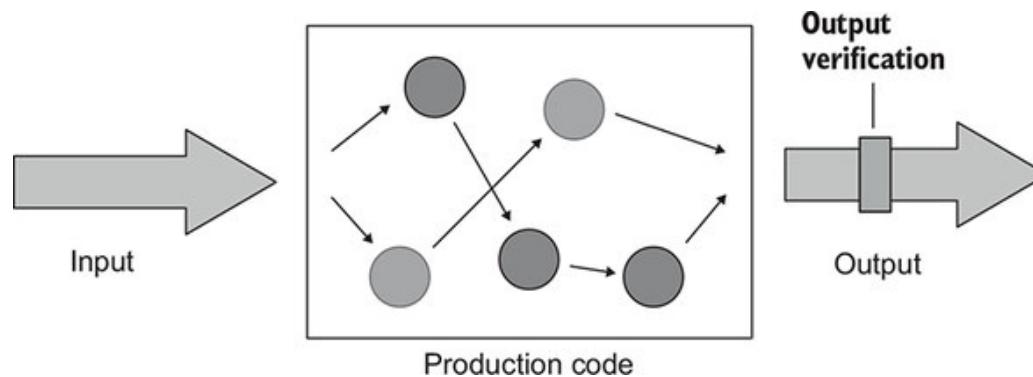
2022 GÜZ

## 3 Farklı Birim Testi Türü

- Output-based testing
- State-based testing
- Communication-based testing

Tek bir test içerisinde bu stillerden biri, herhangi ikisi ya da tümü yapılabilir.

# Output-Based Style



Testin hiç «side effect» özelliği olmadığı kabul edilir ve sonuç sadece SUT sonucu ve çağrıran kısma dönen değerdir.

```

public class PriceEngine
{
    public decimal CalculateDiscount (params Product[]
products)
    {
        decimal discount = products.Length * 0.01m;
        return Math.Min(discount, 0.2m);
    }
}

```

[Fact]

```

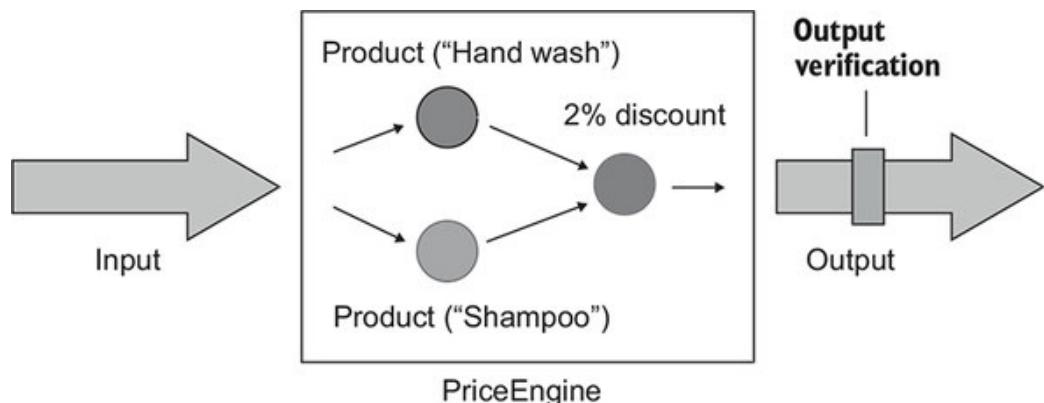
public void Discount_of_two_products()
{
    var product1 = new Product("Hand wash");
    var product2 = new Product("Shampoo");
    var sut = new PriceEngine();

    decimal discount = sut.CalculateDiscount (product1,
product2);

    Assert.Equal(0.02m, discount);
}

```

- PriceEngine, ürün sayısını %1 ile çarpar ve sonucu %20 ile sınırlar.
- Ürünleri herhangi bir veritabanında tutmaz. CalculateDiscount() yönteminin tek sonucu, çıktı değeri döndürdüğü indirimdir.



- Çıktı tabanlı birim testi stili, işlevsel (fonksiyonel) test olarak da bilinir. Bu test stili, side effect (yan etkisi olmayan) kod tercihini vurgulayan bir programlama yöntemi olan fonksiyonel programlamada yaygın kullanılmaktadır.

# State-Based Style

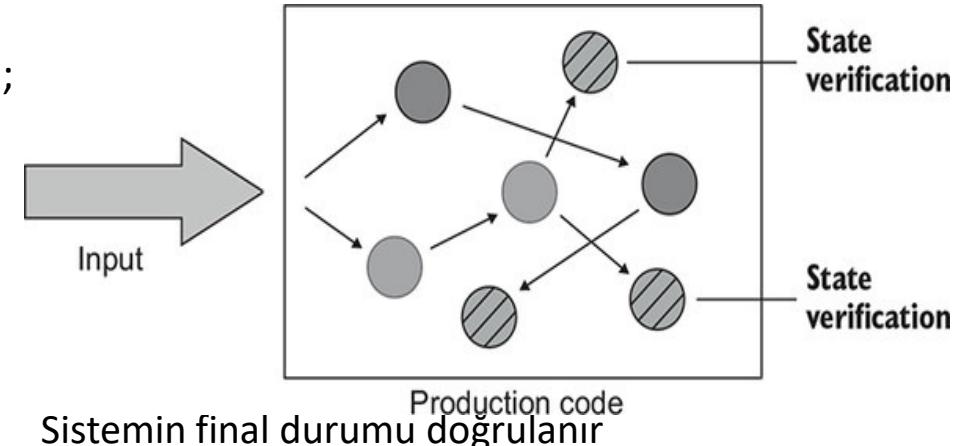
```
public class Order
{ private readonly List<Product> _products = new List<Product>();
  public IReadOnlyList<Product> Products => _products.ToList();
  public void AddProduct(Product product)
  {   _products.Add(product)   } }
```

[Fact]

```
public void Adding_a_product_to_an_order()
{   var product = new Product("Hand wash");
  var sut = new Order();

  sut.AddProduct(product);

  Assert.Equal(1, sut.Products.Count);
  Assert.Equal(product, sut.Products[0]);
}
```



- Durum tabanlı stilde, bir işlem tamamlandıktan sonra sistemin durumunu doğrulanır.
- Durum terimi, SUT'nin kendisinin, ortak çalışanlarından birinin veya veri tabanı veya dosya sistemi gibi süreç dışı bir bağımlılığın durumunu ifade edebilir.
  - ❖ Order sınıfı yeni ürün ekler.
  - ❖ Test ekleme yapıldıktan sonra Products koleksiyonunu doğrular.
  - ❖ AddProduct sonucu siparişin durumunda yapılan değişikliktir.
    - ✓ Bu sonuç önceki stilden farklıdır.

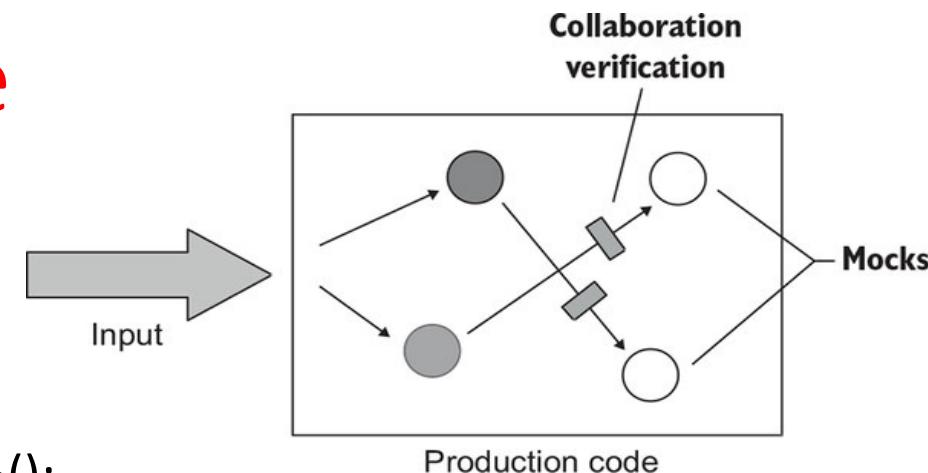
# Communication-Based Style

[Fact]

```
public void Sending_a_greetings_email()
{
    var emailGatewayMock = new Mock<IEmailGateway>();
    var sut = new Controller(emailGatewayMock.Object);

    sut.GreetUser("user@email.com");

    emailGatewayMock.Verify
        (x => x.SendGreetingsEmail("user@email.com"),
         Times.Once);
}
```



Bu stil, test edilen sistem (SUT) ile iletişimde olduğu parçalar arasındaki etkileşimi doğrulamak için taklit (mock) kodlar kullanır. Yani, SUT'un etkileşimlerini taklitleriyle değiştirir ve SUT'nin etkileşimde olduğu parçaları doğru şekilde çağrıdığını doğrular.

## Test Stillerinin Karşılaştırılması

- ❑ Kod parçalarında yapılan regresyon testleri ve bunların geri bildirim hızına göre belirlenen ölçütler kullanılarak test stilleri karşılaştırılabilir.
- ❑ Regresyonlara karşı koruma metrikleri tanımlanmış olan üç test stiline bağlı değildir.
  - ❖ Regresyonu belirlemenin ve regresyondan korunmanın metrikleri aşağıdaki üç özelliğe bağlıdır. Bunlar:
    - ❖ Test süresince çalıştırılan kod miktarı,
    - ❖ Kodun karmaşaklılığı,
    - ❖ Testi yapılan uygulama alanının önemi (importance of domain).

## Yeniden Düzenlemeye (Refactoring) Direnç Durumu ve Test Stilleri

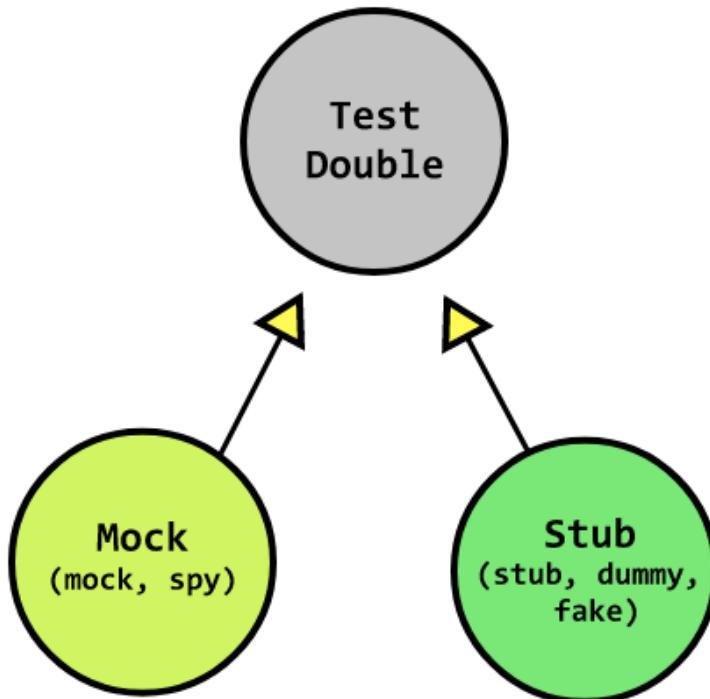
- Yeniden düzenlemeye dirençlilik durumu false positives (false alarms) sayısı ile ölçülür.
  - ❖ Bunlar, yeniden düzenleme sırasında oluşturulan «false positive» testlerdir.
  - ❖ «False positives» aynı zamanda kodun implementasyondaki ayrıntılarına bağımlılıklarının (coupling) sonucudur ve gözlemlenmesi mümkün değildir.
- Çıktıya dayalı test, «false positives» ile karşılaşmamak için en iyi koruma sağlar. Çünkü;
  - ❖ Testler yalnızca test edilen yöntemle eşleşir.
  - ❖ Bu tür testlerin uygulama ayrıntılarına bağlanabilmesinin (couple) tek yolu, test edilen yöntemin kendisinin bir uygulama ayrıntısı (yani *production code* içerisinde) olmasıdır.

# Yeniden Düzenlemeye (Refactoring) Direnç Durumu ve Test Stilleri

- ❑ Duruma dayalı testler genellikle «false positive» ile karşılaşmaya daha yatkındır.
  - ❖ Test edilen yönteme ek olarak, bu tür testler sınıfın durumu ile de çalışır.
  - ❖ Yapılan test ile implementasyon kodu (production code) arasındaki bağlantı (coupling) ne kadar büyük olursa, bu testin sizıntı yapan bir uygulama detayına bağlanma olasılığı o kadar artacaktır.
  - ❖ Durum tabanlı testler genellikle implementasyondan (production code) daha büyük bir API yüzeyine bağlanırlar.
- ✓ Bu nedenle de bunların uygulama ayrıntılarıyla bağlantılı olma (coupling) olasılığı da oldukça yüksek olacaktır.

## Yeniden Düzenlemeye (Refactoring) Direnç Durumu ve Test Stilleri

- ❑ İletişim tabanlı (communication –based) testler, yanlış alarmlara (false positives) karşı en savunmasız test stilidir.
  - ❖ Test çiftleriyle etkileşimleri kontrol eden testlerin büyük çoğunluğu kırılganlaşır (fragile).
  - ❖ Taslaklarla (Stubs) etkileşim her zaman gerçekleşir.
    - ✓ Bu tür etkileşimlerin kontrolü mümkün değildir.
    - ✓ Taklitler (mocks), uygulamanın (production code) sınırı dışındaki etkileşimleri doğruladıklarında ve sadece bu etkileşimlerin yan etkileri (side effects) kullanıcı tarafından görülebildiğinde iyi bir seçimdir.
- ❑ İletişim tabanlı testlerin kullanılması, yeniden düzenlemeye karşı direnci korumak için ekstra çaba gerektirir.



- Test çifti terimi ile «stunt double» (dublöör) sözcüğünden esinlenilmiştir.
- Test sırasında simgelenen bir nesne
- (production object) herhangi bir nesneyi işaret eder.
- Mocks: Command-like Operations (outgoing interactions or state changes (test stili olarak)) bağımlılıklarına (coupling) karşı «mocks» tercih edilir.
- Test edilen sisteme (SUT) «mocks» iletilir ve daha sonra bir testin onaylanması (assert) aşamasında, bağımlılıkların durumunu değiştirmek için doğru çağrıların yapılmış olduğu kontrol edilir.
- «Mocks» sadece «library» fonk. kullanır; bir ya da iki satırda tamamlanır.

Spy : Casus elle müdahale (hand-roll) dışında, taklitlerle (mocks) tamamen aynı şeydir.

Stubs ve Mock kullanıldığında karşılaşılabilecek potansiyel problemler nelerdir?

# «Mocks» ve «Stub» Farkı Nedir?

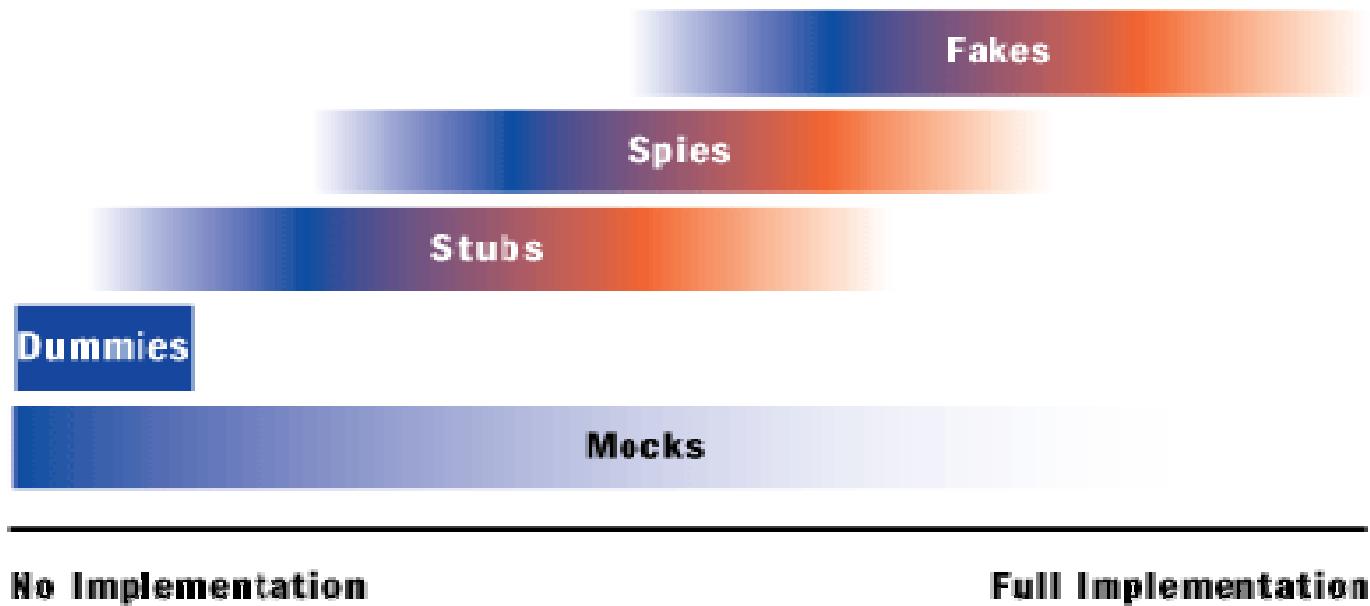
◻ Mocks nesnelerdir.

- ❖ Bu nesneler kayıt çağrılarını alırlar.
- ❖ Test savı (assertion) sırasında beklenen tüm eylemlerin gerçekleşip gerçekleşmediği doğrulanır (verify).

◻ Stub bir nesnedir.

- ❖ Ön tanımlı (predefined) datayı tutar ve test sırasında çağrırlara cevap vermek üzere bunu kullanır.
- ❖ Gerçek veri ile cevap veren nesnelerin içeriilmesi istenmediğinde, ya da istenmeyen «side effects» olduğunda kullanılır.
  - ✓ Bir örnek, bir yöntem çağrısına yanıt vermek için veri tabanından bazı verileri alması gereken bir nesne olabilir.
  - ✓ Gerçek nesne yerine bir stub eklenir ve hangi verilerin döndürülmesi gerektiği tanımlanır.

# Komutlar için «Mocks», Sorgular için «Stubs»



<https://blog.ploeh.dk/2013/10/23/mocks-for-commands-stubs-for-queries/>

Örnekler için Danimarkalı yazılımcı Mark Seeman blog (2013)

## Maintainability of Output-Based Tests

- ❑ En sürdürülebilir test stilidir.
- ❑ Ortaya çıkan testler her zaman kısa ve özlüdür ve bu nedenle «maintainability» kolaydır.
- ❑ «Output-based style» sadece girdi ve çıktıya indirgenir;
  - ❖ Test için gerekenler sadece birkaç satır kodla hazırlanır; bu da bir yönteme girdi sağlamak ve çıktısını doğrulamaktır.
- ❑ Çıktı tabanlı testte temel alınan kodun genel/ iç (internal) durumu değişmemesi gereği için, bu testler süreç dışı bağımlılıklarla ilgilenmez.

# Maintainability State Based Tests

[Fact]

```
public void Adding_a_comment_to_an_article()
{
    var sut = new Article();
    var text = "Comment text";
    var author = "John Doe";
    var now = new DateTime(2019, 4, 1);

    sut.AddComment(text, author, now);

    Assert.Equal(1, sut.Comments.Count);      1
    Assert.Equal(text, sut.Comments[0].Text);  1
    Assert.Equal(author, sut.Comments[0].Author);  1
    Assert.Equal(now, sut.Comments[0].DateCreated);  1
}
```

- Durum tabanlı testler çıktı tabanlı testlerden daha az sürdürülebilirdir.
- Test bir metne bir yorum ekler ve ardından bunun metnin yorumlar listesinde olup olmadığını kontrol eder.
- Test basitleştirilmiş ve sadece tek bir yorum içermesine rağmen, «assert» kısmı dört satırdır.
- Sonuç olarak bu testler genellikle buradakinden daha fazla veri doğrulaması gereklidir ve bu da testin boyut olarak önemli ölçüde büyümeyi gerektirir.

## Maintainability of Communication-Based Tests

- ❑ İletişim tabanlı testler, test çiftlerinin (test doubles) ve etkileşim iddialarının (interaction assertions) ayarlanması gerektiğini gerektirir.
  - ❖ Bu da testin çok fazla yer kaplaması demektir.
- ❑ «Mock chains» (mocks veya stubs diğer mock döndürür vs. ) içeriği için testler daha da büyür ve daha az sürdürülebilir, yani devamlılığı sağlanabilir hale gelir

# Coverage Criteria

Graph Coverage /Structural Coverage  
Criteria

## Coverage Criteria (Kapsam Kriterleri)

- ❑ A coverage criterion is simply a recipe for generating test requirements in a systematic way (Kapsam kriteri, test gereksinimlerini oluşturmak için sistematik bir bir yol göstericidir.)

In other words:

- ❑ A coverage criterion is a rule or collection of rules that impose test requirements on a test set

(Kapsam kriteri, bir test seti için test gereksinimlerinin oluşturulduğu bir kural veya kurallar bir koleksiyonudur.)

# Graph Coverage Criteria

- ❑ Structural Coverage Criteria : Defined on a graph just in terms of nodes and edges

(Grafın sadece düğümleri ve kenarları olarak betimlenir)

- ❑ Data Flow Coverage Criteria : Requires a graph to be annotated with references to variables

(Değişkenler referans alınarak graf açıklamaları yapılır)

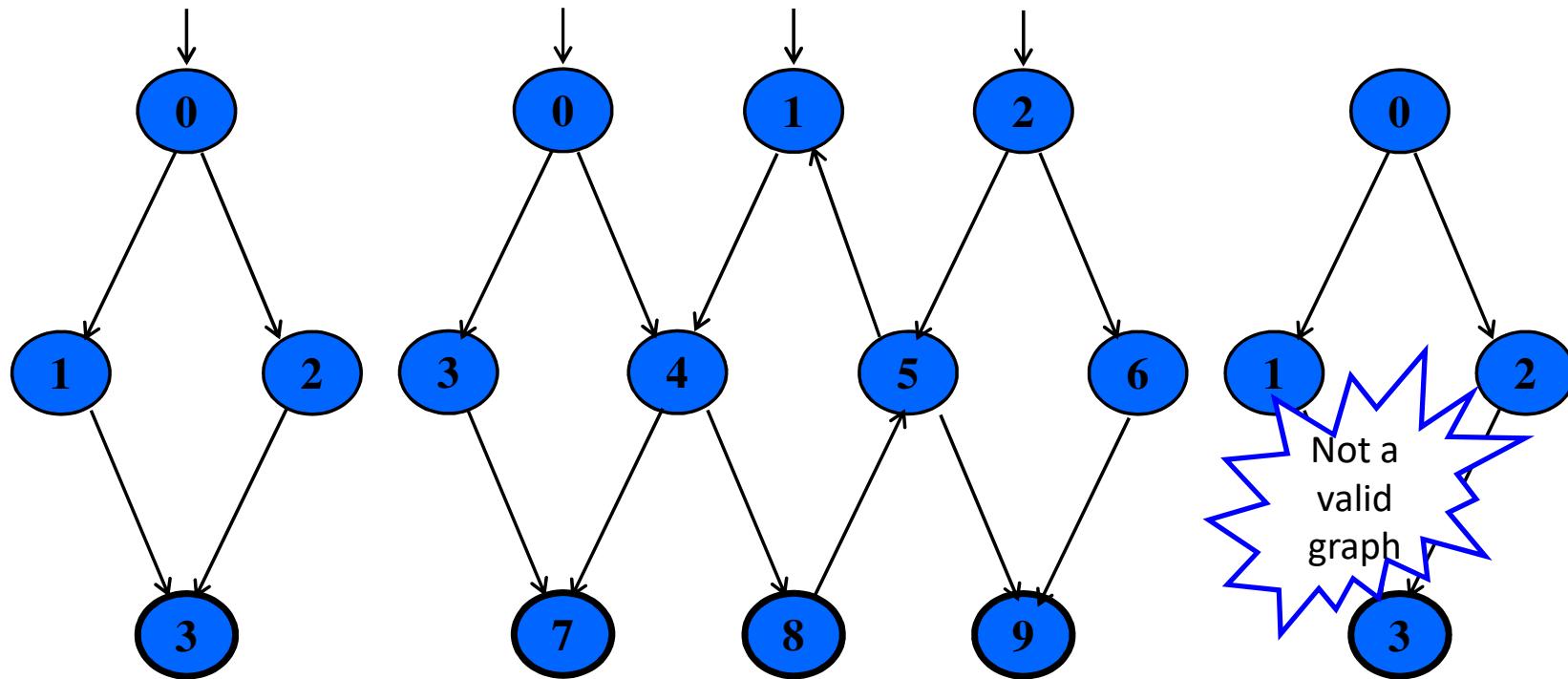
## Definition of a Graph

- A set  $N$  of nodes,  $N$  is not empty
- A set  $N_0$  of initial nodes,  $N_0$  is not empty
- A set  $N_f$  of final nodes,  $N_f$  is not empty
- A set  $E$  of edges, each edge from one node to another

## «Coverage» Kriterleri: Niçin Önemli?

- ❑ Bazı uygulamalarda kapsam (coverage) kriterlerinin sağlanması kolay değildir.
  - ❖ İşlemlerin tamamlanmasının uzun süre gerektirmesi nedeni ile maliyeti yüksektir.
  - ❖ Bu durumda belli bir «coverage» düzeyinde sonuç elde edilmesi hedeflenir.
- ❑ Bazı gereksinimler sağlanmadığı taktirde testin gerçekleştirilmesi mümkün değildir.

# Graf Teorisi ve Yazılım Testi



$$N_0 = \{ 0 \}$$

$$N_f = \{ 3 \}$$

$$N_0 = \{ 0, 1, 2 \}$$

$$N_f = \{ 7, 8, 9 \}$$

$$N_0 = \{ \}$$

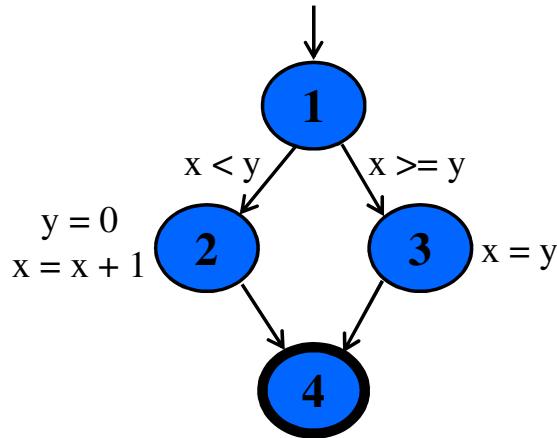
$$N_f = \{ 3 \}$$

## Test Yolları ( Test Paths)

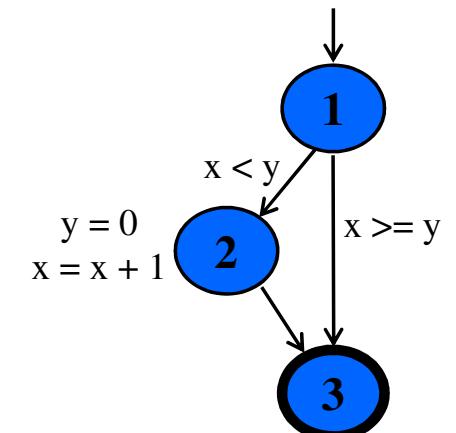
- ❑ Başlangıç düğümünden başlar ve final düğümünde biter
- ❑ Test yolları test durumlarının işleyişini yönlendirir.
  - Bazı yollar pek çok test ile çalışır.
  - Bazı yolların hiçbir test ile çalışması (execute) mümkün değildir.
- ❑ Tüm test yolları tek bir düğüm ile başlar ve farklı bir düğümde sonlanır.

# Control Flow Graph : if Statement

```
(x < y)
{
    y = 0;
    x = x + 1;
}
else
{
    x = y;
}
```



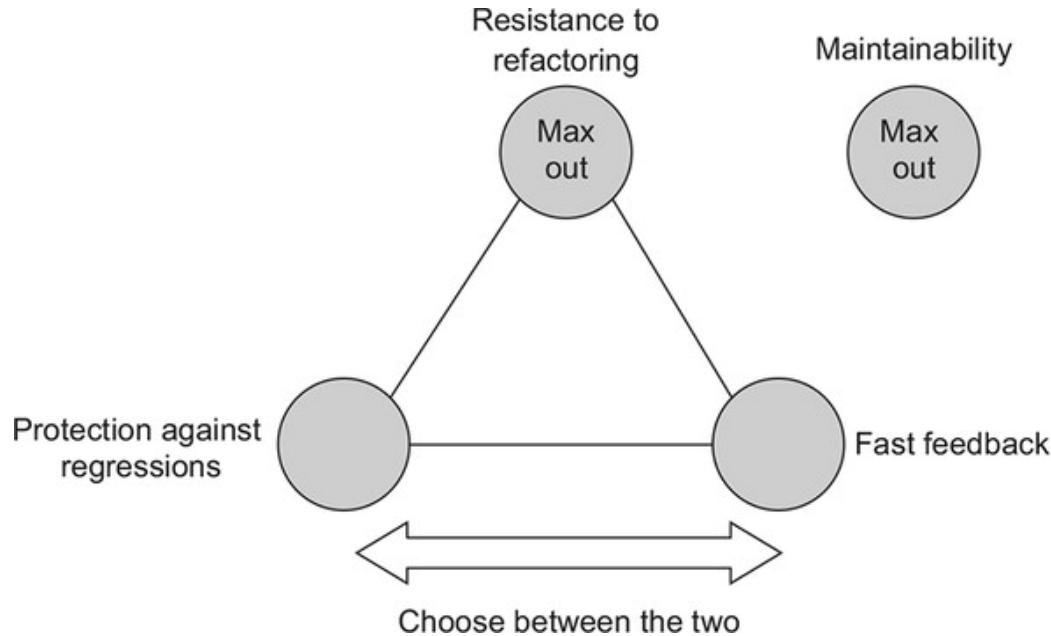
```
if (x < y)
{
    y = 0;
    x = x + 1;
}
```



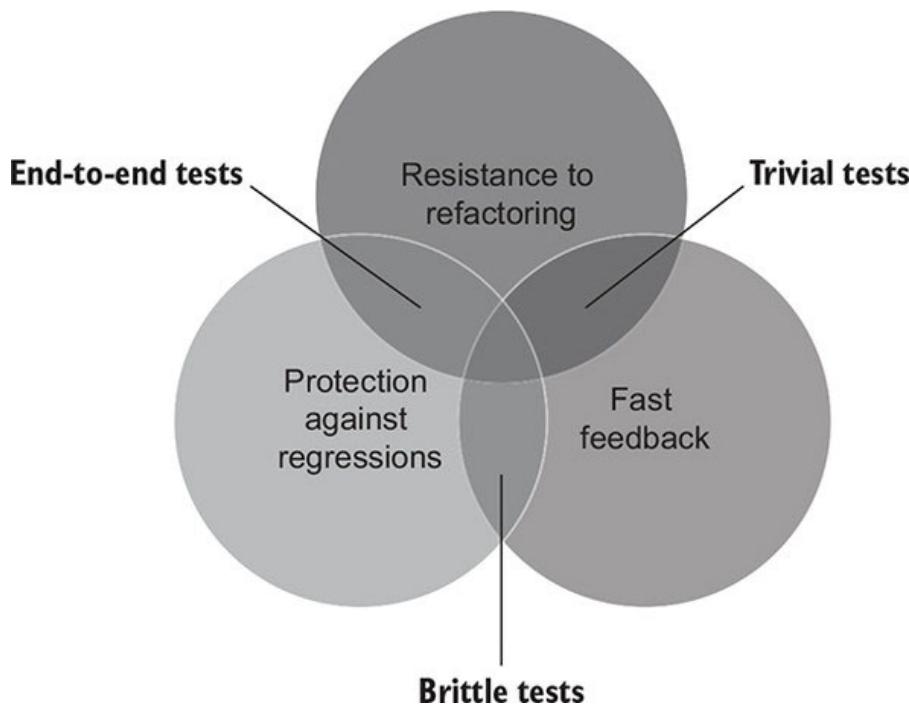
# YAZILIM KALİTE GÜVENCESİ VE TESTİ

DERS NOTU 6  
2022 GÜZ

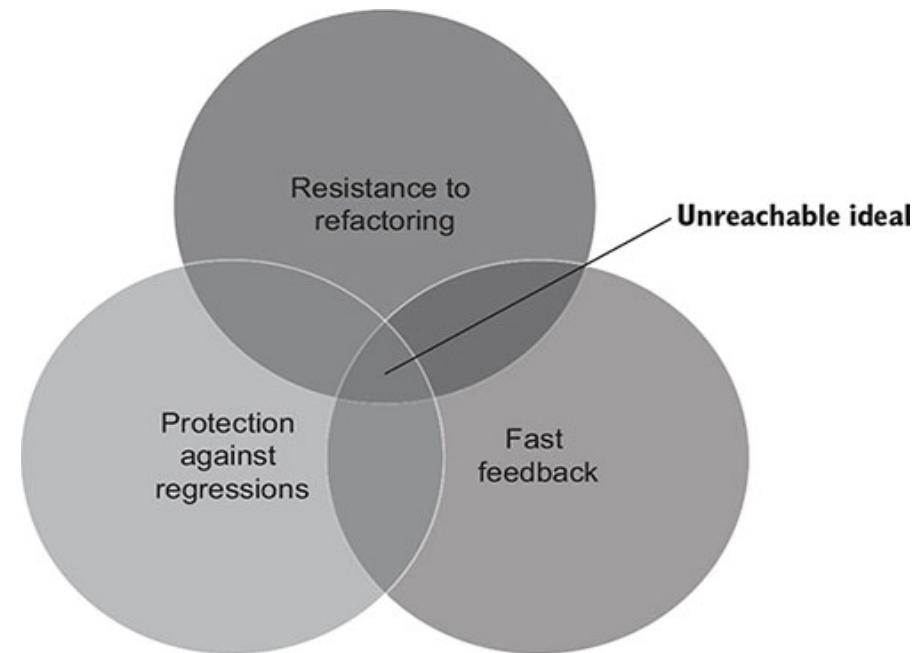
# Maksimum Sürdürülebililik



- Maksimum oranda sürdürülebilir ve yeniden düzenlemeye karşı direnç gösteren testler en iyi testlerdir.
- Test süresince bu iki özelliğin en üst düzeyde olması önemlidir.
- Regresyonları engellemeye çalışmak ve hızlı geri dönüşler arasında yapılan seçim sürdürülebilirliğin derecesini belirleyecektir.



- Brittle (fragile) testler hızlı çalışır ve regresyonlara karşı iyi koruma sağlarlar.
- Aynı zamanda yeniden düzenlemeye karşı çok az dirençlidirler.
- Yazılım kırılganlığı nedir?
  - ❖ Mevcut yazılımı onarmanın zorluğudur.
  - ❖ Güvenilir (reliable) görünmesine rağmen, küçük bir parça değiştirildiğinde başarısız olmasıdır.



Her üç özelliğin mükemmel olduğu ideal bir test mümkün değildir.

## Testi En Fazla Kırılgan (Brittle) Yapan Faktörler

- ❑ Statik veya uygun olmayan <threads> paylaşımı,
- ❑ Gerçekçi olmayan ya da yetersiz test verileri,
- ❑ Ayrıntılı mock planları,
- ❑ Test altında (fragile) kırılgan kod,
- ❑ Çok fazla uç durum (edge cases) veya fazla sayıda döngüsel karmaşıklık,
- ❑ Çok fazla sorumluluk,
- ❑ Sızdırılan soyutlamalar (leaky abstractions),
- ❑ Test kodun, test oturumları (sessions) arasında bağımsız olmaması,
- ❑ Sonuçları ortama göre değişebilen testler (örneğin performans zamanları)
- ❑ Birbirinden bağımsız çalışmayan testler.

## İyi Test ve CAP Teoremi

- ❑ İyi bir birim testinin üç özelliği arasındaki denge, CAP teoremine benzer.
- ❑ CAP teoremi, dağıtılmış bir veri deposunun aşağıdaki üç özellikten ikisinden fazlasını aynı anda sağlamasının mümkün olamayacağını ifade eder.
- ❑ **Consistency** (Tutarlılık): Her okuma en son yazılanı veya bir hatayı alır.
- ❑ **Availability** (Kullanılabilirlik): Her isteğin (request) bir yanıt aldığı (response) anlamına gelir (sisteme tüm düğümleri etkileyen sorunlar dışında).
- ❑ **Partition Tolerance** (Bölünebilme toleransı): Sistemin ağ bölümlenmesine rağmen çalışmaya devam ettiği anlamına gelir (ağ düğümleri arasındaki bağlantı kaybolur).

## CAP ile İyi Test Seçimi Benzerliği

- ❑ CAP teorisi ile iyi test arasında iki yönlü bir benzerlik görülmektedir.
- ❑ Öncelikle , üç özellikten ikisinde bir değişim- dokus mümkündür.
- ❑ İkincisi, büyük ölçekli dağıtılmış sistemlerdeki bölünebilme toleransı bileşeni ile ilgili müzakere yapmak bile söz konusu olmaz. Örneğin Amazon web sitesi gibi büyük bir uygulama tek bir makinede çalışması mümkün değildir.
  - ❖ Bu nedenle de tutarlılık ve kullanılabilirliği birlikte tercih etme seçeneği yoktur.
    - ✓ Sunucu ne kadar büyük olursa olsun, tek bir sunucuda depolanamayacak kadar çok veriye sahiptir.

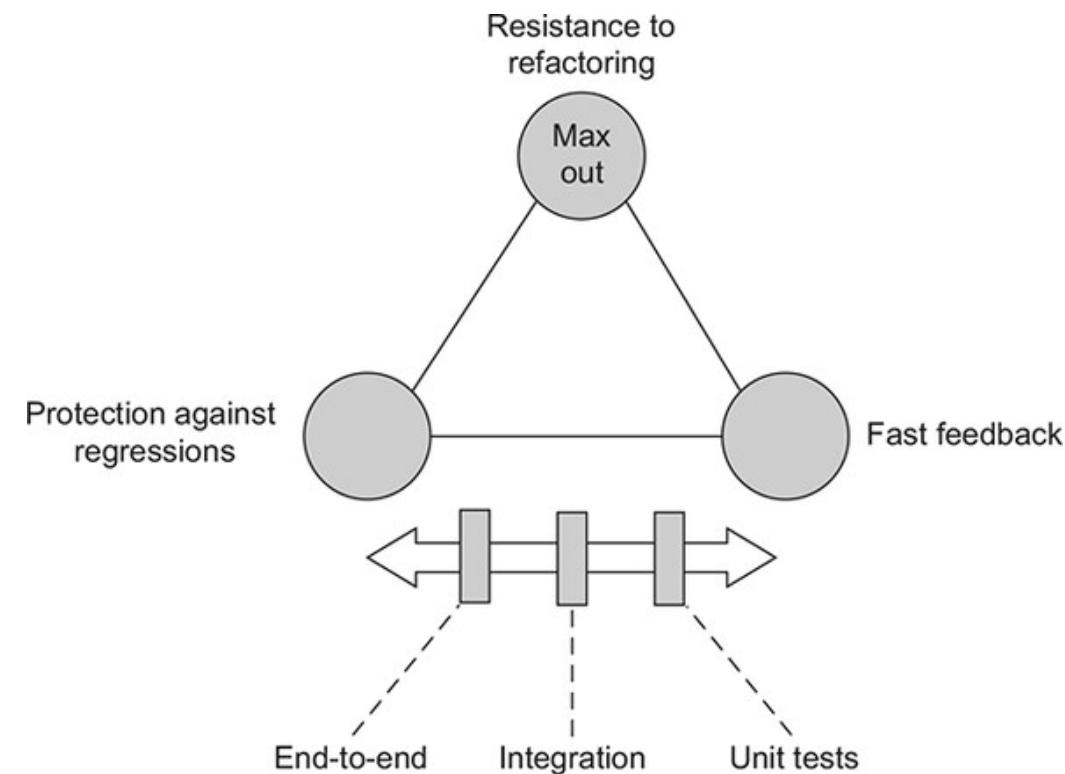
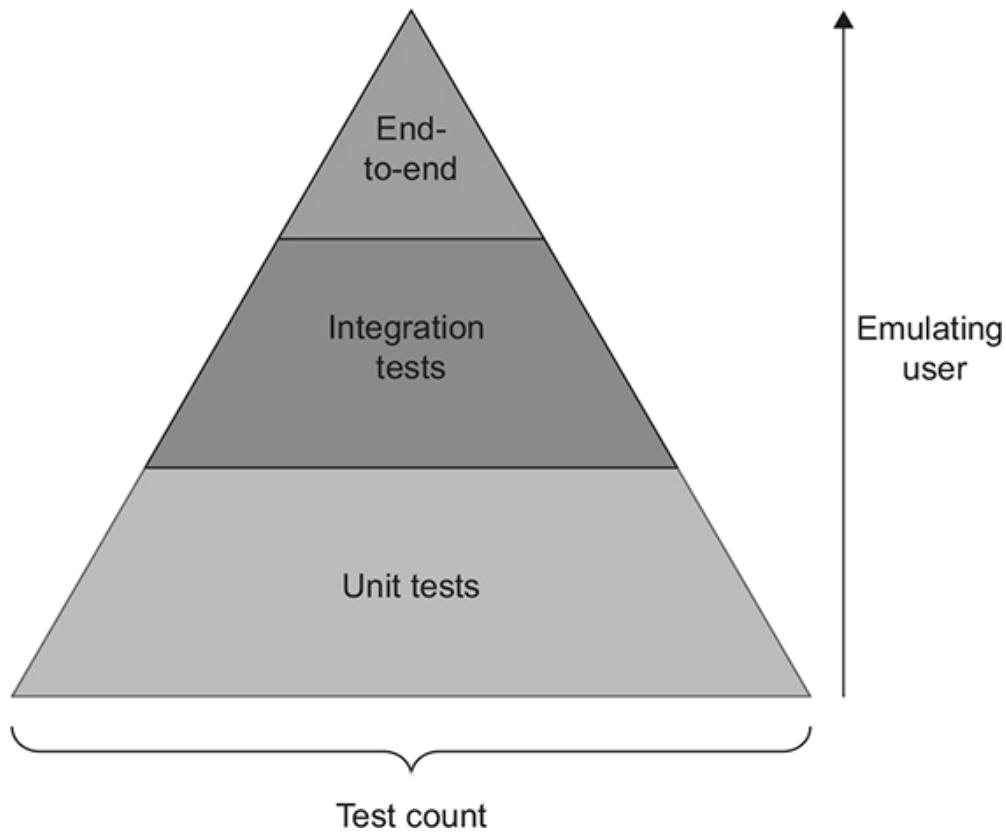
## CAP ile İyi Test Örneği

- ❑ Bu örnekte , aynı zamanda tutarlılık ve kullanılabilirlik arasında bir değişim tokusu gerekiyor.
- ❑ Sistemin bazı bölümlerinde, daha fazla kullanılabilirlik elde etmek için daha az tutarlılık seçilebilir.
  - ❖ Bir ürün kataloğu görüntülenirken, bazı bölümlerinin güncellliğini yitirmiş olmasının sorun oluşturmayıacağı kabul edilmiştir.
    - ✓ Senaryoda kullanılabilirliği yüksek önceliğe koyar.
  - ❖ Ama, bir ürün açıklaması güncellenirken tutarlılık, kullanılabilirlikten daha önemlidir
    - ✓ Ağ düğümleri üzerinde çalışırken, birleştirme çakışmalarını (merge conflicts) önlemek için bu açıklamanın en son hali konusunda ortak görüşe sahip olmaları gereklidir.

# İyi Test Nedir? (tekrar)

- Protection against regressions
- Resistance to refactoring
- Fast feedback
- Maintainability

# Test Piramiti ve Sürdürülebilirlik



- ❑ Piramitin farklı test türleri, hızlı geri bildirim ve regresyonlara karşı koruma arasında farklı seçimler yapar
- ❑ Uçtan uca testler, regresyonlara karşı korumayı destekler
- ❑ Birim testleri hızlı geri bildirimini vurgular
- ❑ Entegrasyon testleri diğer iki testin ortasındadır.

# Trivial Code and Test Example

## Protection Against Regressions

```
public class User  
{  public string Name { get; set; }  
}
```

[Fact]

```
public  
void Test()  
{  var sut = new User();  
  sut.Name = "John Smith";  
  Assert.Equal("John Smith",sut.Name);  
}
```

- Yeniden düzenlemeye (refactoring) karşı dirençlidir.
- Hızlı geri bildirim (feedback) sağlar
- Regresyonlara karşı koruması düşüktür.
- ????

## Tanımlamalar

### End to End Testing / E2E Test

- Uçtan Uca Test ile, uygulamanın akışının başlangıç noktasından bitiş noktasına kadar kusursuz çalışıp çalışmadığı test edilir.
- Uçtan uca testte yazılım, veri tabanları, ağlar, dosya sistemleri ve harici hizmetler gibi tüm bağımlılıklar ve bunların entegrasyonları birlikte test edilir.
- Bu özelliği ile Sistem Testinden ayrılır.
- Sistem testi aynı fonksiyonelliği farklı veri girişleri ile test ederek tepkileri değerlendirir.

### Integration Test

- Birim testlerin başarılı olarak tamamlandıktan sonra ilişkili modüllerin entegrasyonunun sağlanıp sağlanmadığının testidir..

```

public interface IRenderer
{
    string Render(Message message);
}

public class MessageRenderer : IRenderer
{
    public IReadOnlyList<IRenderer> SubRenderers { get; } {
        public MessageRenderer()
        {
            SubRenderers = new List<IRenderer>
            {
                new HeaderRenderer(),
                new BodyRenderer(),
                new FooterRenderer()    };    }

    public string Render(Message message)
    {
        return SubRenderers
            .Select(x => x.Render(message))
            .Aggregate("", (str1, str2) => str1 + str2);
    }
}

```

## Mesaj içeren bir HTML kodunun oluşturulması ve Testi

```

public class Message
{
    public string Header { get; set; }
    public string Body { get; set; }
    public string Footer { get; set; }    }

```

- ❑ **MessageRenderer sınıfı**, mesajın böлümleri üzerindeki bilginin aktarıldığı birden fazla «sub-renderer» içerir.
- ❑ Sonuçlar bir HTML belgesinde birleştirilir.
  - ❖ Sub-renderers (alt oluşturucular), ham metni HTML etiketleriyle düzenler (orchestrate).

# Orkestrasyon Örneği

```
public class BodyRenderer : IRenderer
{
    public string Render(Message message)
    {
        return $"<b>{message.Body}</b>";
    }
}
```

How can **MessageRenderer** be tested?

# MessageRenderer sınıfının testi

[Fact]

```
public void MessageRenderer_uses_correct_sub_renderers()
{
    var sut = new MessageRenderer();

    IReadOnlyList<IRenderer> renderers = sut.SubRenderers;

    Assert.Equal(3, renderers.Count);
    Assert.IsAssignableFrom<HeaderRenderer>(renderers[0]);
    Assert.IsAssignableFrom<BodyRenderer>(renderers[1]);
    Assert.IsAssignableFrom<FooterRenderer>(renderers[2]);
}
```

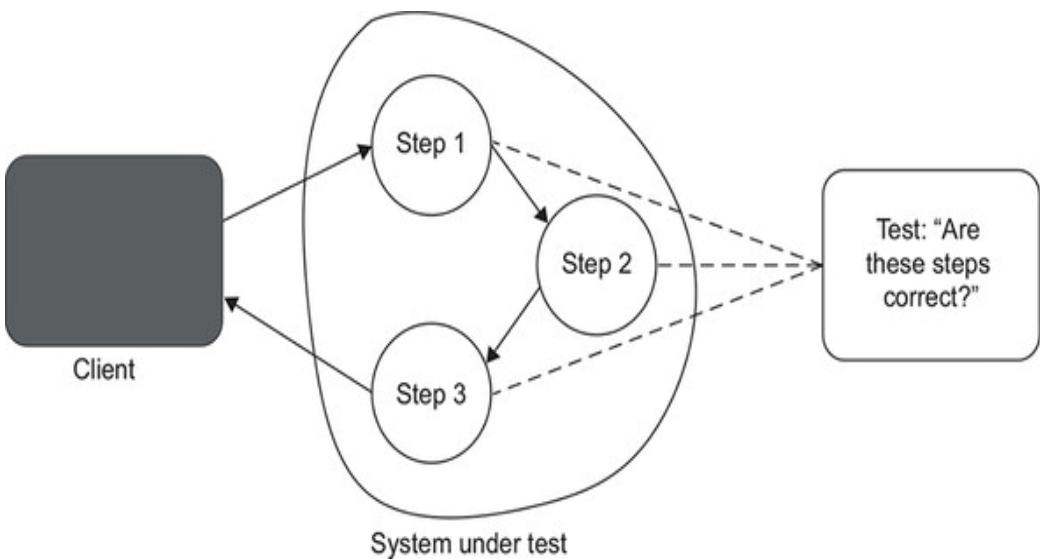
SUT 'nin uygulama ayrıntısına bağlı bir test

- Test doğru olarak yapılandırılmıştır. Yine de şu sorular sorulabilir.
- Sub-renderers tüm bekleneni verir mi?
  - Parçalar doğru sırada görünürler mi?
  - MessageRenderer'ın iletileri işleme biçimini doğru mudur?
- Doğru olduğu kabul edilmiştir.
- Test, MessageRenderer'ın gözlemlenebilir davranışını gerçekten doğrular mı?
  - Sub-renderers yeniden düzenirse veya herhangi bir parça yenisiyle değiştirirse ne olur?
  - Bu bir hataya yol açar mı?

## Hepsi mümkündür...

- ❑ Örneğin HTML dokümanının aynı kalması koşulu ile bir sub-renderer kompozisyonu değiştirilebilir.
- ❑ Örneğin BodyRenderer yerine BoldRenderер yazılabilir.
  - ❖ Yani böyle bir yer değiştirme yapılabilir.
- ❑ Örneğin tüm alt oluşturuculardan (sub-renderers) vazgeçip bunlar doğrudan MessageRenderер'da uygulanabilir
- ❑ Bunlardan biri yapıldığında sonuç değişimese bile test kırmızı sonuç verir.
  - ❖ Çünkü test SUT'un ürettiği sonuçla değil, SUT'nın uygulama ayrıntılarıyla eşleşir.

## Sonuç Olarak :



- Doğru olarak yapılandırıldığı kabul edilen ve SUT algoritmasıyla eşleşen test (önceki slaytta verilen) yandaki şekle bir örnektir.
- Verilmiş olan testin belirli bir sonucu verebilmek için SUT adımlarını izler.
  - ❖ Bu nedenle kırılgandır.
- Çünkü SUT uygulamasının herhangi bir yeniden düzenlenmesi, bir test hatası oluşturacaktır.
- SUT'nin uygulama ayrıntılarına bağlı testler, yeniden düzenlemeye (refactoring) dirençli değildir.

# Çözüm olabilir mi?

[Fact]

```
public void Rendering_a_message()
{
    var sut = new MessageRenderer();
    var message = new Message
    {
        Header = "h",
        Body = "b",
        Footer = "f"
    };

    string html = sut.Render(message);
    Assert.Equal("<h1>h</h1><b>b</b><i>f</i>", html);
}
```



Öğrenci Adı ve Soyadı	
Öğrenci Numarası	
Ders Adı	Yazılım Kalite Güvencesi ve Testi
Sınav Tarihi ve saatı	25.11.2022 11.00
Sınav Süresi	50 dakika
Sınav Salonu	Konferans Salonu
Sınav Tipi	Ara Sınav
Öğretim Elemanı Adı	Dr. Öğr. Üyesi Zeynep Altan

## CEVAPLAR

Soru Numarası	1	2	3	4	Toplam
Alınan Not					

1) "Mocks" ile "Stubs" birbirlerinin yerine kullanılabilecek test stilleri midir? Cevabınızı nedenleri ile açıklayarak verin. Cevabınız "test double" ile birlikte 4 farklı anahtar sözcük içermelidir. Kullandığınız anahtar sözcüklerin altı çizilmelidir. Cevabınızda kullanacağınız anahtar sözcükler "mock, stub ve ikinci sorudaki anahtar terim" olamaz (25 puan).

İletişim odaklı testler sınıfların içindeki

birim testlerdir. Test double olusturular. Bunlar ortak szelliklerdir. SUT, mocks kullanıldığındá kataphone fonksiyonları oluşturur. Test double refactoring hakimdeki pesabern etkileşimlerinin doğrulanması (verify) tahtları derak mocks kullanılmıştır.

Yeni fabe - positives düymelerin aksı tür.

dogru olmasının rağmen testin başarısız olmasıdır. Fab kodu Test - bubbles gerçek olde testlerin kırılgan (fragile)

yapır. Stubs diğerini gibi (mocks), nesnelerdir. Nesne kodunda fonksiyonların alınır. 3A'nın Assertion (sayı

doğrulanır (verify). gerçekleştirip gerçekleşmediğ sında bulunur. Fork derak antenini veren test sonra

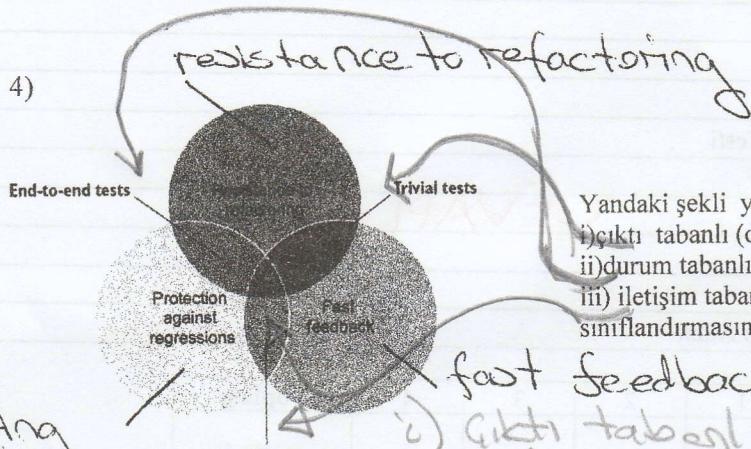
eklenir ve herhangi biri de testin başarısı belittirir. Yani testin başarısı verilen biri de testin başarısıdır.

2) Nesneye yönelik tasarımın temel ilkelerinden "loosely coupling" ile yazılım testi arasındaki ilişkiye nasıl açıklarsınız? (10 puan)

modülerin (sınıflar/bilesenlerin) arasında zayıf ilişkidir. Test lari de böyle bir ilişki kodun yeniden düzenlenmesinde (refactoring) önerilidir. Bilesenler arasında ilişk ının fazla olduğu refactoring> ictin direnci değiştirmeden fabe - positives doğurur. Yani kod doğru olmas na rağmen testlerde sorun olmaz. Clean code ictin istenmez. Refactoring ile kullanılmayan kodu

atmak kolaydır. Bu regresyona etki yani değişikliktir.

3) 11 Kasım tarihinde teslim ettiğiniz araştırma çalışmanızın sonucu 30 puan üzerinden değerlendirilmektedir.



Yandaki şekli yazılım testinin 3 farklı birim testi  
i) çıktı tabanlı (output-based) test,  
ii) durum tabanlı (state-based) test,  
iii) iletişim tabanlı (communication-based) test,  
siniflandırmasına göre ayrı ayrı nasıl değerlendirirsiniz? (25 puan)

i) çıktı tabanlı testlerde testin bir bir *side effect* tener etkisi, özellikle yoldur. SUT tamamlandığında bir değer sağlanır. Bu aktı (output) değeridir. Bu test te denilenektedir.  
< Veri toplanmadık deşam 82 şane alınmas False positives > Yalnız bir koruma seferidir (mantainance). Bütün sadece birlikte en iyidir.  
ii) Durum tabanlı testler *false positives* beraberdir. Bütün ile implementasyon arasında bağ fazla ise sizmalar mümkünür ve deşamlarına koruma artar. Bir durumun deşam ile doğrudur olucu. Bu nedenle yeniden däremeye fazla açık olmadığından sonra for la karsıtlaştırılır. Test final durumuna baktır.  
iii) Metrisin bâzı testte posisyon oradındaki etkileşimleri doğrulama için *mocks* ve *test stubs* kullanır. Yeniden däremeye kendi kuralıdır (BH + BH)

5) void soru5 (int a)

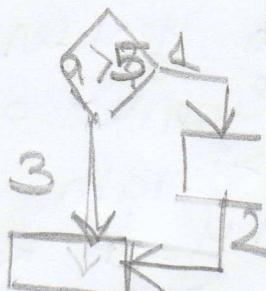
{

```
if (a>5)  
a=a*3;  
cout<<a;
```

}

kod parçası veriliyor.

İki farklı test senaryosu için ilgili değeri girerek "branch coverage" yaklaşımı ile kod parçasının testini gerçekleştirin (10 puan).



3 branch verdit.

$$a=6 \text{ iai} \frac{2}{3}=0,66$$

$$a=2 \text{ iai} \frac{1}{3}=0,33$$

Sekil ya da akillansız cevap

# Coverage Criteria

## GRAPH COVERAGE

- i) Structural Coverage Criteria  
&
- ii) Logic Coverage

# Graph Coverage Criteria

## ❑ Structural Coverage Criteria

Grafın sadece düğümleri ve kenarları olarak betimlenir.

## ❑ Data Flow Coverage Criteria

Değişkenler referans alınarak graf üzerinde açıklamalar yapılarak uygun metotlar uygulanır.

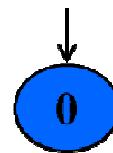
## Test Yolları ( Test Paths)

- Başlangıç düğümünden başlar ve final düğümünde biter
- Test yolları test durumlarının işleyişini simgeler
  - Bazı yollar pek çok test ile işlenir
  - Bazı yolların hiçbir test yaklaşımı ile kontrolü (execute) mümkün olmayabilir.
- Tüm test yolları tek bir düğüm ile başlar ve farklı bir düğümde sonlanır.

## Structural Coverage Node Coverage - NC

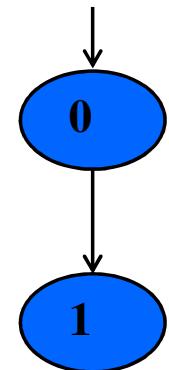
- TR (Test Requirement), G grafına ait erişilebilir her bir düğümdür.
- Sadece bir düğüme sahip bir grafikte herhangi kenar (edge) bulunmaz.

«0» uzunluklu yollardır



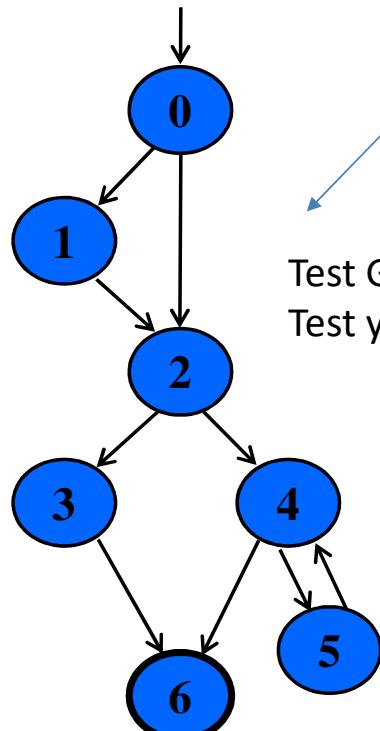
## Structural Coverage Edge Coverage -EC

- ❑ Edge Coverage , Node Coverage gerektirir.
- ❑ Edge Coverage , «Node Coverage» kapsar.
- ❑ «Edge Coverage » length up to 1 şeklinde tanımlanır.
- ❑ TR (Test Requirement) , G grafında 1 uzunluğuna kadar ulaşılabilen tüm yolları içerir.
- ❑ «Node Coverage» ile karşılaştırıldığında «Edge Coverage» daha güçlü test gerçekleştirir.



# Structural Coverage

## Node Coverage and Edge Coverage



### Node Coverage

$$TR = \{ 0, 1, 2, 3, 4, 5, 6 \}$$

Test Paths: [ 0, 1, 2, 3, 6 ] [ 0, 1, 2, 4, 5, 4, 6 ]

Test Gereksinimleri (Test Requirements):  
Test yollarının özellikleridir.

Test durumlarının işleyışı

### Edge Coverage

$$TR = \{ (0,1), (0,2), (1,2), (2,3), (2,4), (3,6), (4,5), (4,6), (5,4) \}$$

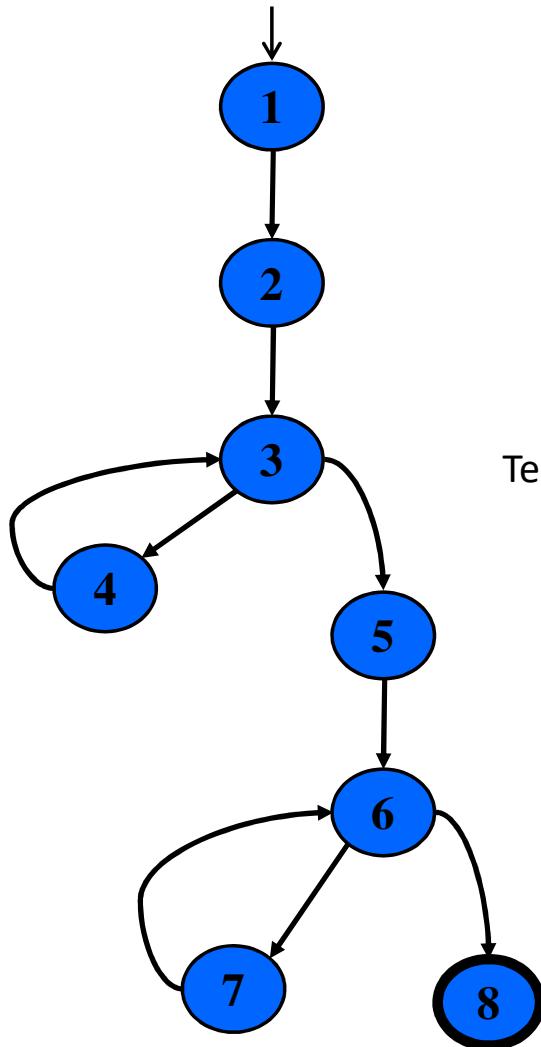
Test Paths: [ 0, 1, 2, 3, 6 ] [ 0, 2, 4, 5, 4, 6 ]

$$\text{path } (t_1) = [ 0, 1, 2, 3, 6 ]$$

$$\text{path } (t_2) = [ 0, 2, 4, 5, 4, 6 ]$$

$$T = \{ t_1, t_2 \}$$

# Structural Coverage - Edge Coverage



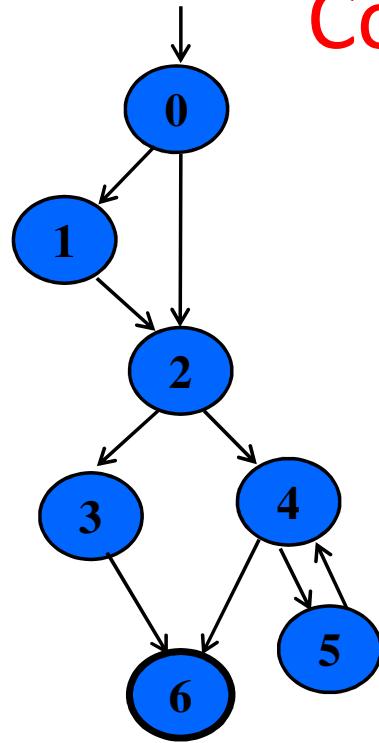
$TR = \{ [1, 2], [2, 3], [3, 4], [3, 5], [4, 3], [5, 6], [6, 7], [6, 8], [7, 6] \}$

Test Gereksinimleri (Test Requirements): Test yollarının özellikleridir.

Test Yollarından (Test Path) biri

Test Path: [ 1, 2, 3, 4, 3, 5, 6, 7, 6, 8 ]

## Complete Path Coverage

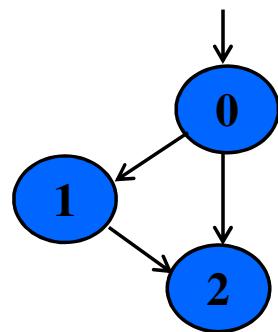


Complete Path Coverage (CPC) : TR contains all paths in G.

This is impossible if the graph has a loop

Test Paths: [ 0, 1, 2, 3, 6 ] [ 0, 1, 2, 4, 6 ] [ 0, 1, 2, 4, 5, 4, 6 ]  
[ 0, 1, 2, 4, 5, 4, 5, 4, 6 ] [ 0, 1, 2, 4, 5, 4, 5, 4, 5, 4, 6 ] .....

# Özetle



**Node Coverage (NC)** :  $TR = \{ 0, 1, 2 \}$

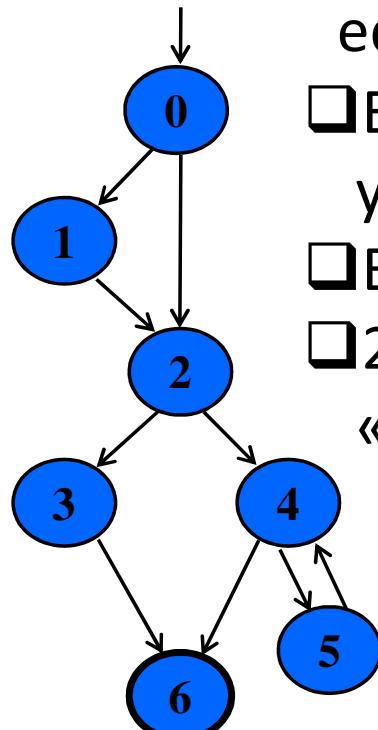
$T_1 = \{ t_1 \} = \text{Test Path} = [ 0, 1, 2 ]$

**Edge Coverage(EC)** :  $TR = \{ (0,1), (0,2), (1,2) \}$

$T_2 = \{ t_1, t_2 \} \text{ Test Paths} = [ 0, 1, 2 ] [ 0, 2 ]$

# Structure Coverage Edge-Pair Coverage

- Edge-pair coverage gerçekleştirmek için TR pairs of edges içerir.
- Edge-pair coverage için, «up to level 2» tanımlaması da yapılır.
- Bu yapıda «node» ve «edge coverage» içerilir.
- 2 uzunluğa kadar ifadesi, 2 «edge» ve daha az uzunluklu «edge» içeren grafları betimler.



$$TR = \{ [0,1,2], [0,2,3], [0,2,4], [1,2,3], [1,2,4], [2,3,6], [2,4,5], [2,4,6], [4,5,4], [5,4,5], [5,4,6] \}$$

Test Paths: [ 0, 1, 2, 3, 6 ] [ 0, 1, 2, 4, 6 ] [ 0, 2, 3, 6 ]  
[ 0, 2, 4, 5, 4, 5, 4, 6 ]

$$\begin{array}{ll} path(t_1) = [ 0, 1, 2, 3, 6 ] & path(t_2) = [ 0, 1, 2, 4, 6 ] \\ path(t_3) = [ 0, 2, 3, 6 ] & path(t_4) = [ 0, 2, 4, 5, 4, 5, 4, 6 ] \end{array}$$

$$T = \{ t_1, t_2, t_3, t_4 \}$$

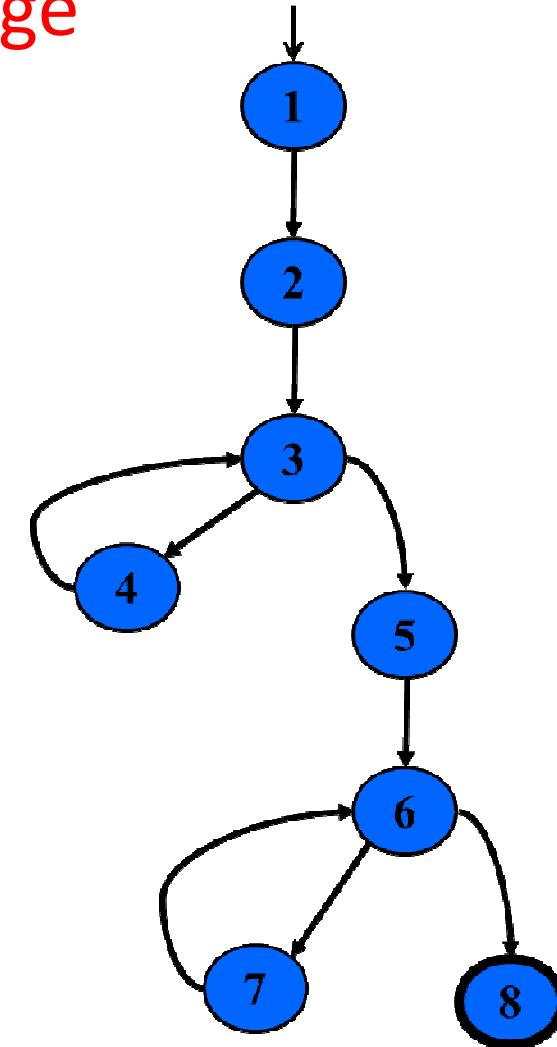
# Structural Coverage Criteria

## Edge-Pair Coverage

TR={ [ 1, 2, 3 ], [ 2, 3, 4 ], [ 2, 3, 5 ], [ 3, 4, 3 ],  
[ 3, 5, 6 ], [ 4, 3, 5 ], [ 5, 6, 7 ], [ 5, 6, 8 ],  
[ 6, 7, 6 ], [ 7, 6, 8 ], [ 4, 3, 4 ], [ 7, 6, 7 ] }

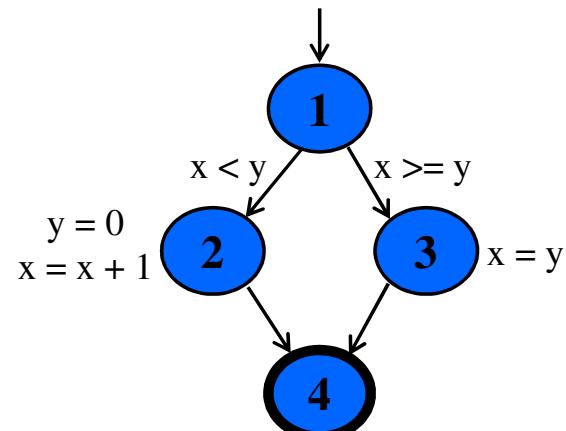
### Test Paths

- i. [ 1, 2, 3, 4, 3, 5, 6, 7, 6, 8 ]
- ii. [ 1, 2, 3, 5, 6, 8 ]
- iii. [ 1, 2, 3, 4, 3, 4, 3, 5, 6, 7, 6, 7, 6, 8 ]

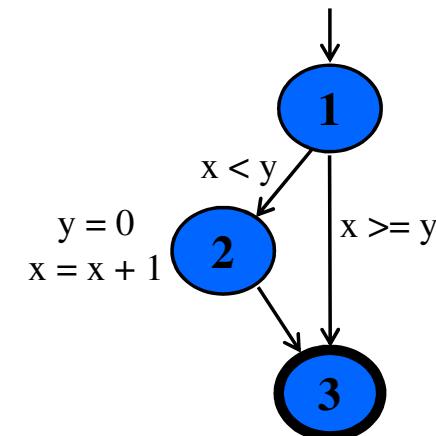


# Control Flow Graph : if Statement

```
if (x < y)
{
    y = 0;
    x = x + 1;
}
else
{
    x = y;
}
```



```
if (x < y)
{
    y = 0;
    x = x + 1;
}
```

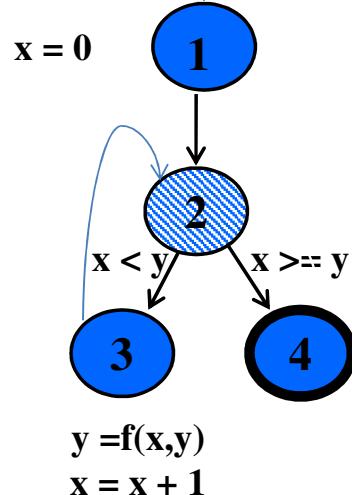


# Control Flow Graph : while Statement & for Loop

```

x = 0;
while (x < y)
{
    y = f (x, y);
    x = x + 1;
}

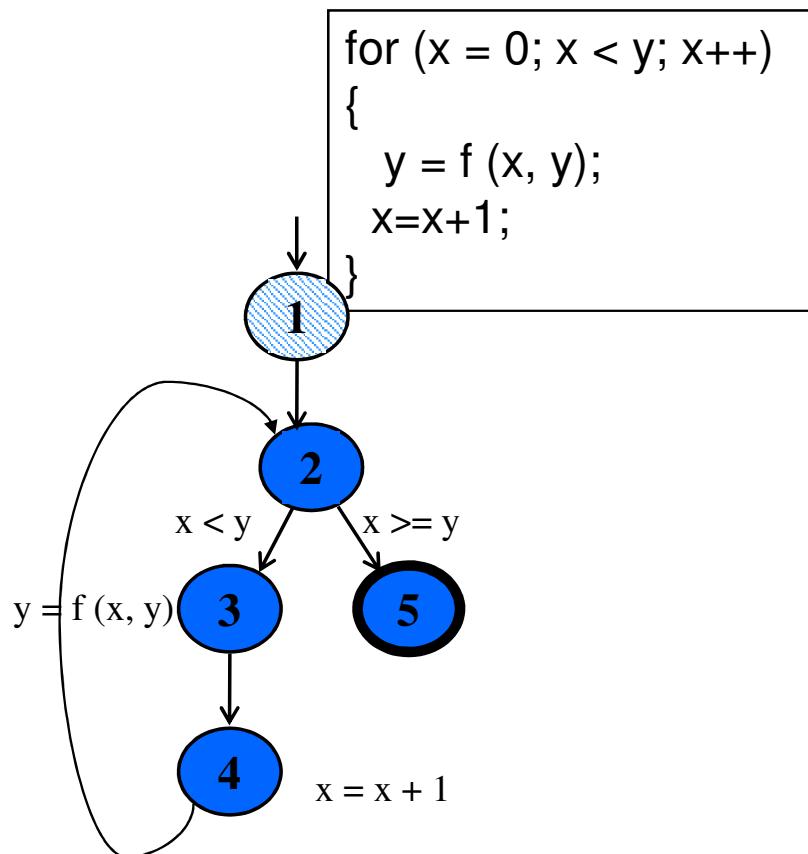
```



```

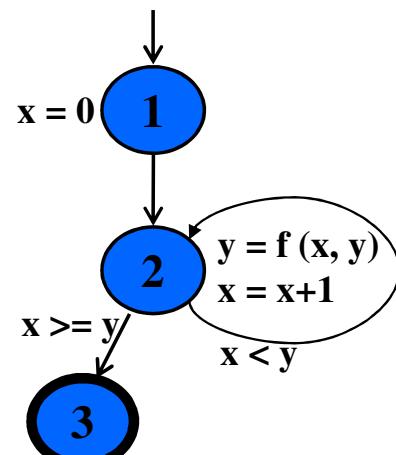
for (x = 0; x < y; x++)
{
    y = f (x, y);
    x=x+1;
}

```

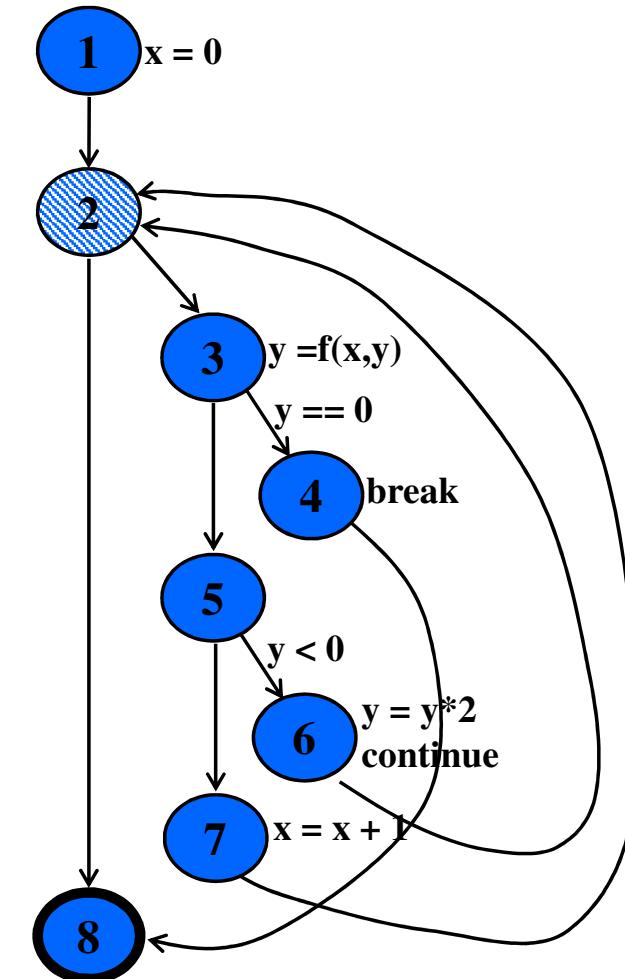


# Control Flow Graphs : do loop, break and continue

```
x = 0;
do
{
    y = f (x, y);
    x = x + 1;
} while (x < y);
println (y)
```



```
x = 0;
while (x < y)
{
    y = f (x, y);
    if (y == 0)
    {
        break;
    } else if y < 0)
    {
        y = y * 2;
        continue;
    }
    x = x + 1;
}
print (y);
```



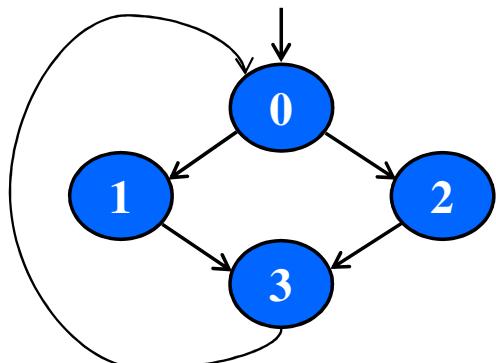
# Simple Paths

**Simple Path** : A path from node  $n_i$  to  $n_j$  is simple if no node appears more than once, except possibly the first and last nodes are the same

«Simple path» belirlenen yol sadece başlangıç ve final düğümleri tekrarlanabilir, onun dışında hiçbir düğüm tekrarı olamaz.

«Simple path» diğer tüm alt yolları (subpaths) içerir.

A loop is a simple path if it has no internal loops and it includes all other subpaths

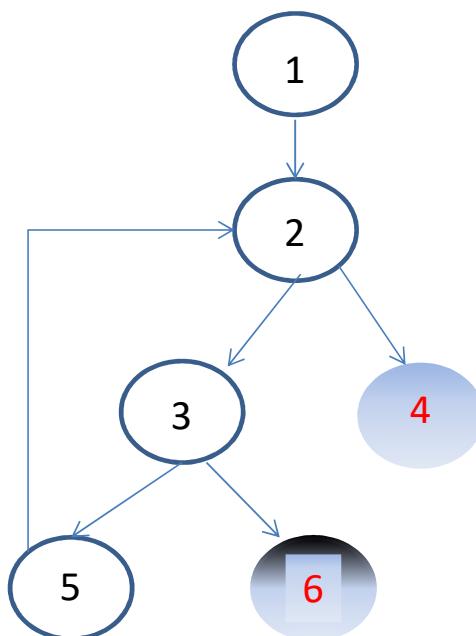


**Simple Paths** : [ 0, 1, 3, 0 ], [ 0, 2, 3, 0 ], [ 1, 3, 0, 1 ],  
[ 2, 3, 0, 2 ], [ 3, 0, 1, 3 ], [ 3, 0, 2, 3 ],  
[ 1, 3, 0, 2 ], [ 2, 3, 0, 1 ],  
[ 0, 1, 3 ], [ 0, 2, 3 ], [ 1, 3, 0 ], [ 2, 3, 0 ], [ 3, 0, 1 ], [ 3, 0, 2 ],  
[ 0, 1 ], [ 0, 2 ], [ 1, 3 ], [ 2, 3 ], [ 3, 0 ],  
[ 0 ], [ 1 ], [ 2 ], [ 3 ]

# Structural Coverage Criteria

## Simple Paths

Simple Paths = { [1],[2], [3], [4] ,[5] [6],  
[1 ,2], [2,3], [2,4] [3,5] [3,6] [5, 2],  
[1, 2,3] , [1, 2,4] , [2, 3,5], [2,3,6], [3, 5, 2], [5, 2, 3], [5, 2, 4],  
[1, 2, 3, 6], [1, 2, 3, 5], [2, 3, 5, 2], [3, 5, 2, 3],  
[3, 5, 2, 4], [5, 2, 3, 6], [5, 2, 3, 5] }



# Structural Coverage Criteria

## Prime Paths

- ❑ Prime paths are *maximal length simple paths*
  - ❖ A simple path from node  $n_i$  to  $n_j$  is a *prime path* if it is simple and does not appear as a proper subpath of any other simple path.
- ❑ Prime paths do not have any internal loops,
  - ❖ But the entire path may be a loop.

## Prime Paths and Prime Path Coverage

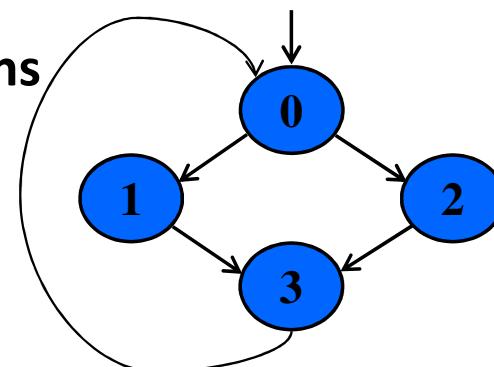
- ❑ Prime path (PP) gives strong coverage of loops without requiring an infinite number of paths
- ❑ In prime path coverage, tests must tour **each prime path** in the graph  $G$ .
- ❑ Prime path coverage requires touring **all subpaths** of length 0 (all nodes), of length 1 (all edges), length 2, 3, etc.
- ❑ Prime path **subsumes node coverage, edge coverage and edge-pair coverage**
- ❑ In prime path coverage, tests must tour each prime path in the graph  $G$ .

## Prime Paths

Prime Paths : [ 0, 1, 3, 0 ], [ 0, 2, 3, 0 ], [ 1, 3, 0, 1 ],  
[ 2, 3, 0, 2 ], [ 3, 0, 1, 3 ], [ 3, 0, 2, 3 ],  
[ 1, 3, 0, 2 ], [ 2, 3, 0, 1 ]

Prime paths are maximal length simple paths

Simple Paths : [ 0, 1, 3, 0 ], [ 0, 2, 3, 0 ], [ 1, 3, 0, 1 ],  
[ 2, 3, 0, 2 ], [ 3, 0, 1, 3 ], [ 3, 0, 2, 3 ],  
[ 1, 3, 0, 2 ], [ 2, 3, 0, 1 ],  
[ 0, 1, 3 ], [ 0, 2, 3 ], [ 1, 3, 0 ], [ 2, 3, 0 ], [ 3, 0, 1 ], [ 3, 0, 2 ],  
[ 0, 1 ], [ 0, 2 ], [ 1, 3 ], [ 2, 3 ], [ 3, 0 ],  
[ 0 ], [ 1 ], [ 2 ], [ 3 ]



## Simple Paths & Prime Paths Example

Len 0

[0]  
[1]  
[2]  
[3]  
[4]  
[5]  
[6] !

Len 1

[0, 1]  
[0, 2]  
[1, 2]  
[2, 3]  
[2, 4]  
[3, 6] !  
[4, 6] !  
[4, 5]  
[5, 4]

Len 2

[0, 1, 2]  
[0, 2, 3]  
[0, 2, 4]  
[1, 2, 3]  
[1, 2, 4]  
[2, 3, 6] !  
[2, 4, 6] !  
[2, 4, 5] !  
[4, 5, 4] \*  
[5, 4, 6] !  
[5, 4, 5] \*

Len 3

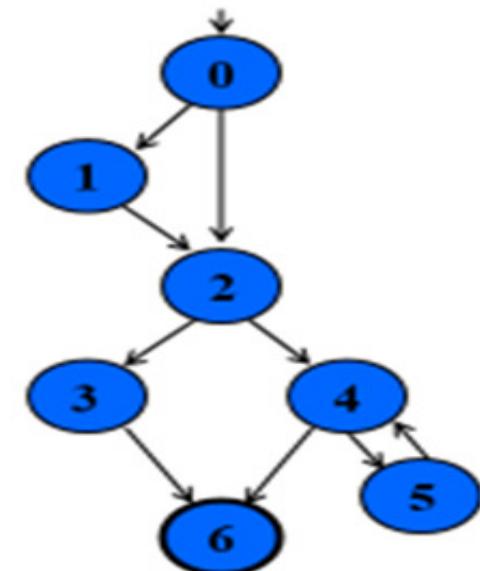
[0, 1, 2, 3]  
[0, 1, 2, 4]  
[0, 2, 3, 6] !  
[0, 2, 4, 6] !  
[1, 2, 3, 6] !  
[1, 2, 4, 5] !  
[1, 2, 4, 6]

Len 4

[0, 1, 2, 3, 6] !  
[0, 1, 2, 4, 6] !  
[0, 1, 2, 4, 5] !

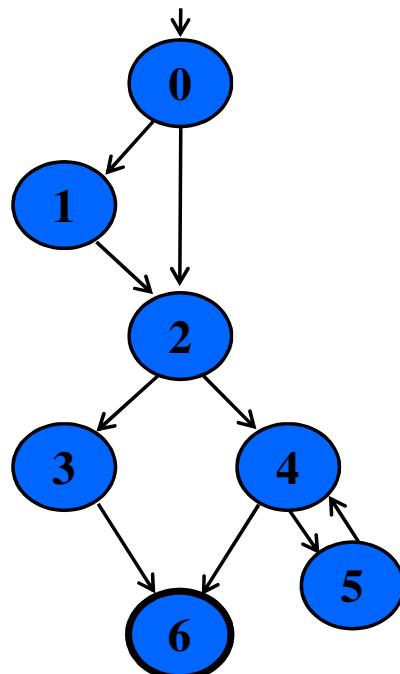
! means path terminates

\* means path cycles



# Simple Paths & Prime Paths Example

! ve \* ile işaretlenmiş olan tüm «simple paths» için, bu yolların «subpath» olduğu genişletilmiş yollar tamamlandığında, bu «subpaths» birer «prime path» olarak alınacaktır.



## Prime Paths

- [ 0, 1, 2, 3, 6 ]
- [ 0, 1, 2, 4, 5 ]
- [ 0, 1, 2, 4, 6 ]
- [ 0, 2, 3, 6 ]
- [ 0, 2, 4, 5 ]
- [ 0, 2, 4, 6 ]
- [ 5, 4, 6 ]
- [ 4, 5, 4 ]
- [ 5, 4, 5 ]

Execute loop 0 times

Execute loop more than once

# Graph Coverage for Source Code

Data Flow Graph Coverage for Source Code

# Yazılım Kalite Güvencesi ve Testi Araştırma & Uygulama Ödevi 2

23 Aralık Cuma derste elden teslim edilecektir. Ödev aynı zamanda Beykent Pusulaya yüklenenecek (22 Aralık 23.59) ve değerlendirmeler burada ilan edilecektir. Ders başlamadan imza karşılığında çıktı teslimi zorunludur.

1. Functional Testing using Selenium
2. Performance Testing using JMeter
3. Functional Testing using UFT
4. Test Management using Jira
5. Defect Management using Bugzilla
6. Functional Testing using TestComplete
7. Mobile Testing using Appium
8. Database Testing
9. Acceptance Testing using Cucumber

- Yukarıdaki 9 farklı otomatik test yaklaşımından herhangi üçünü (1 -3 içerisinde ilk seçiminiz, 4-6 içerisinde 2. seçiminiz, 7-9 arasından 3. seçiminiz olacak şekilde) inceleyiniz ve temel özellikleri ile anlatınız (50 puan).
- Cevabınızda hangi üçlü sınıflandırma içerisinde seçim yaptığınızı açık olarak belirtiniz.
- Seçmiş olduğunuz test yaklaşımının açıklayıcı anlatımlarından sonra birer örnek üzerinde testlerini gerçekleştiriniz ve süreci açıklayınız (50 puan).
- Bu çalışma final sınavında 20 puan üzerinden değerlendirilecektir.

# Tests with Data Flow Criteria

- ❑ Data flow criteria require tests that tour subpaths from specific definitions of variables to specific uses

*Belirli değişken tanımlarından bu değişkenlerin kullanıldığı yerlere erişimleri test eder.*

- ❖ Nodes where a **variable is assigned** a value are called **definitions** (or *defs*)

*Değer ataması yapılmış düğümler *defs* olarak adlandırılır.*

- ❖ Nodes where the value of a **variable is accessed** are called **uses**.

*Bir değişken değerine erişen düğümler *uses* olarak adlandırılır.*

## def-use · Definition

A *def* is a location in the program where a value for a variable is stored into memory (assignment, input, etc.)

*def bir programda bellekte depolanmış bir değere ait yerleşimdir.*

❑ A *use* is a location where a variable's value is accessed.

*use bir değişkenin değerine erişildiği yerleşimdir.*

# Data Flow Coverage for Source

❑ **def** : A location where a **value** is **stored** into memory

- ❖ x appears on the left side of an assignment ( $x = 44$ )
- ❖ x is a formal parameter of a method (when method starts)
- ❖ x is an input to a program

❑ **use** : A location where variable's **value** is **accessed**

- ❖ x appears on the right side of an assignment ( $x=x+1$ )
- ❖ x appears in a **conditional test**
- ❖ x is an actual parameter to a method
- ❖ x is an output of the program
- ❖ x is an output of a method in a return statement

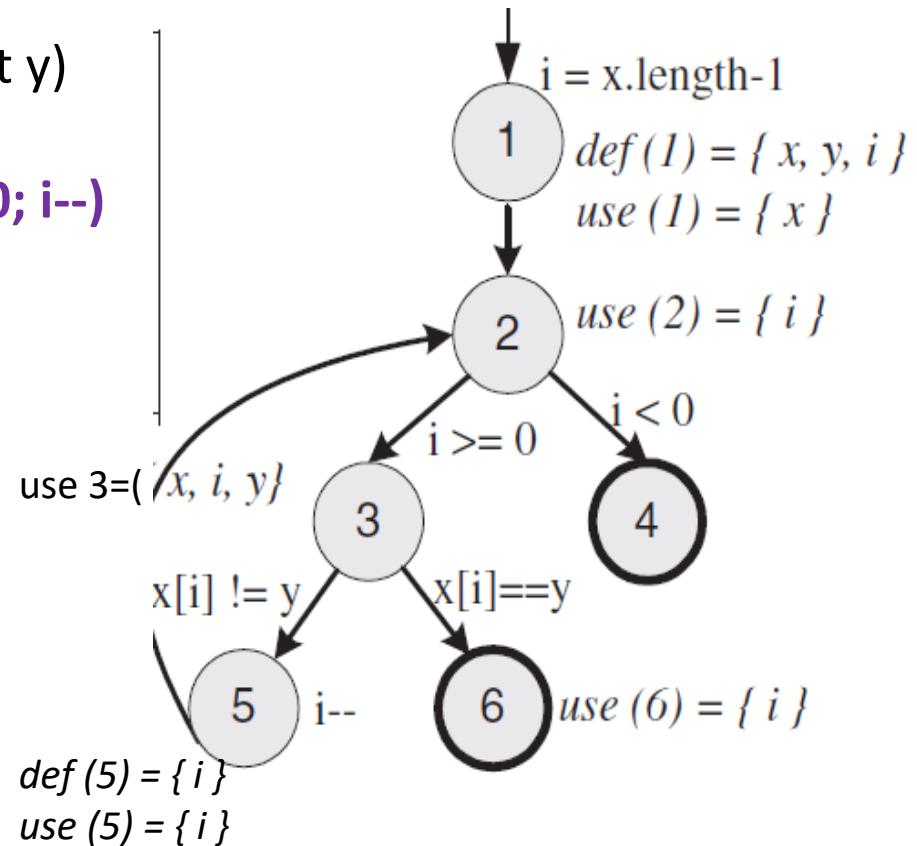
## DU – Pair Def- Use çifti

- ❑ def ve use aynı düğümde ise, DU- pair oluşturur
- ❑ Bu durum bir döngü oluşur.
- ❑ Aynı düğüm içerisinde use kullanımından sonra def oluşursa düğüm bir döngüdür.

# Örnek 1 :

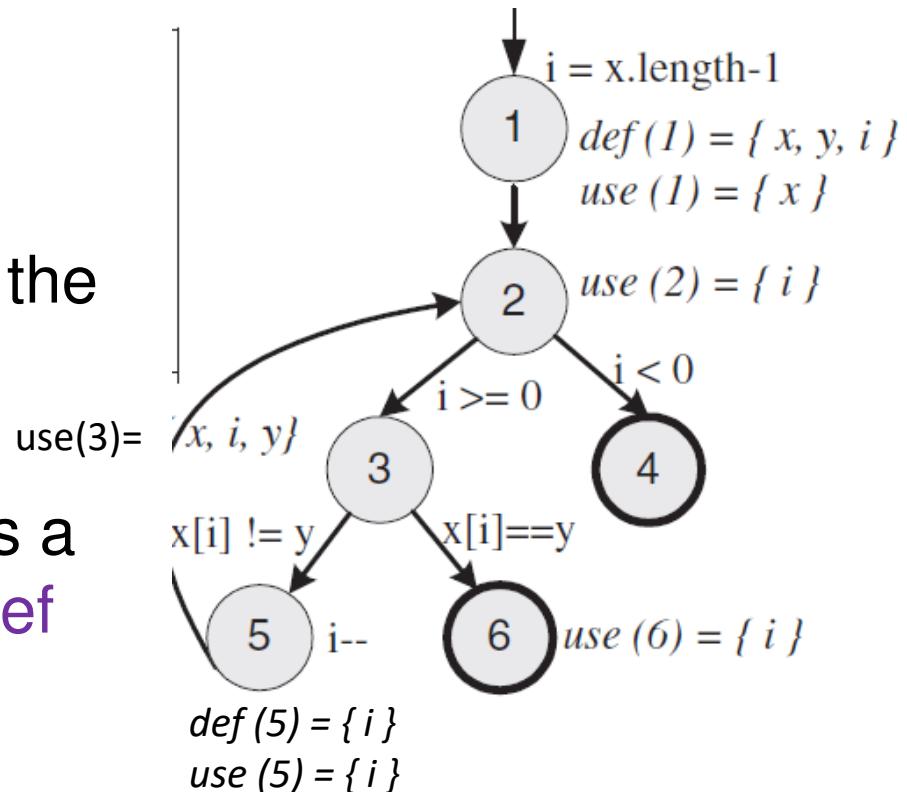
## Data Flow Covarege

```
public int findLast (int []x, int y)
{
    for (int i = x.length-1; i>=0; i--)
    {
        if (x[i] == y)
            return i;
    }    return -1;
```



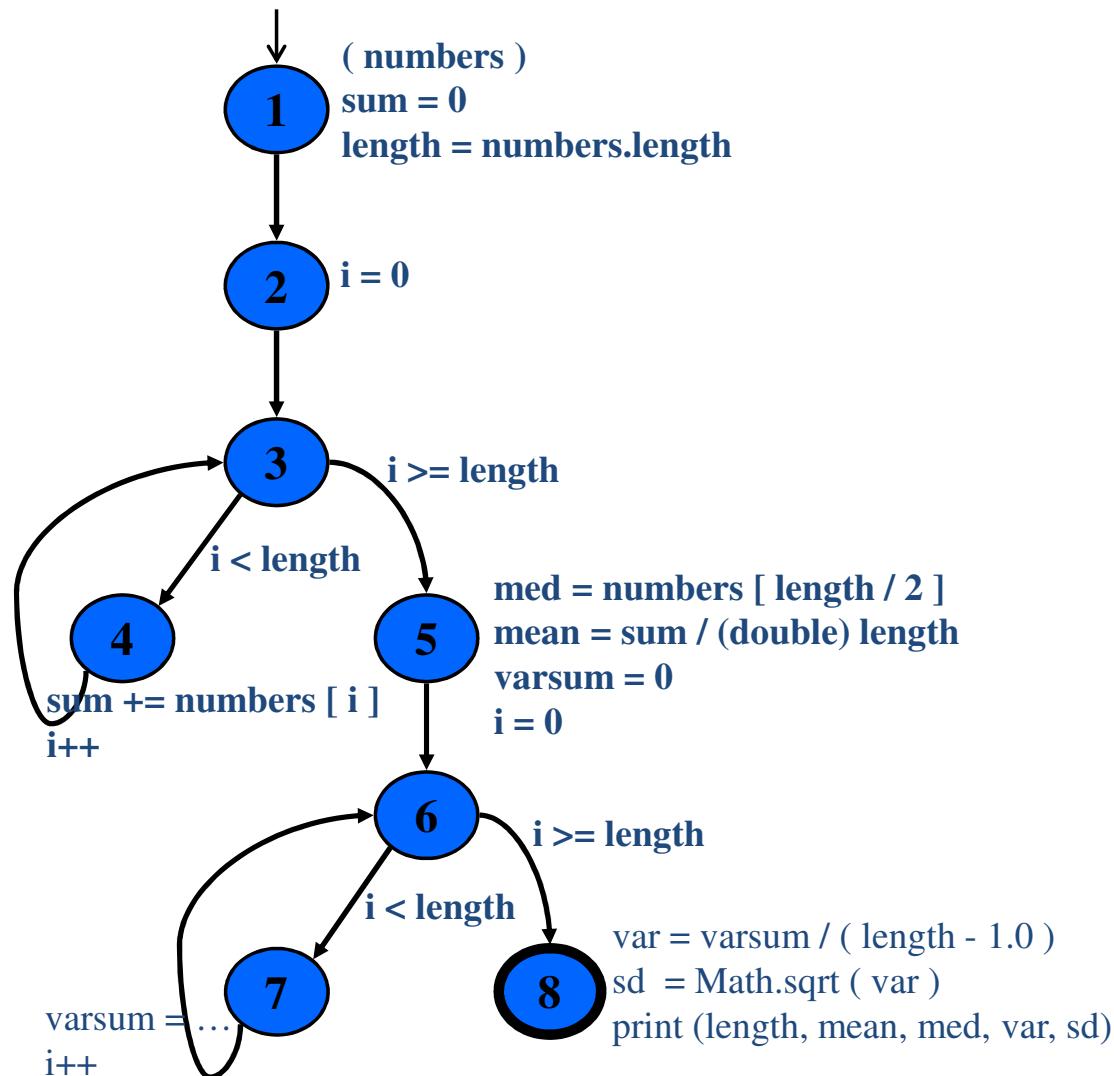
## Def-Use Pairs =(def location, use location, variable)

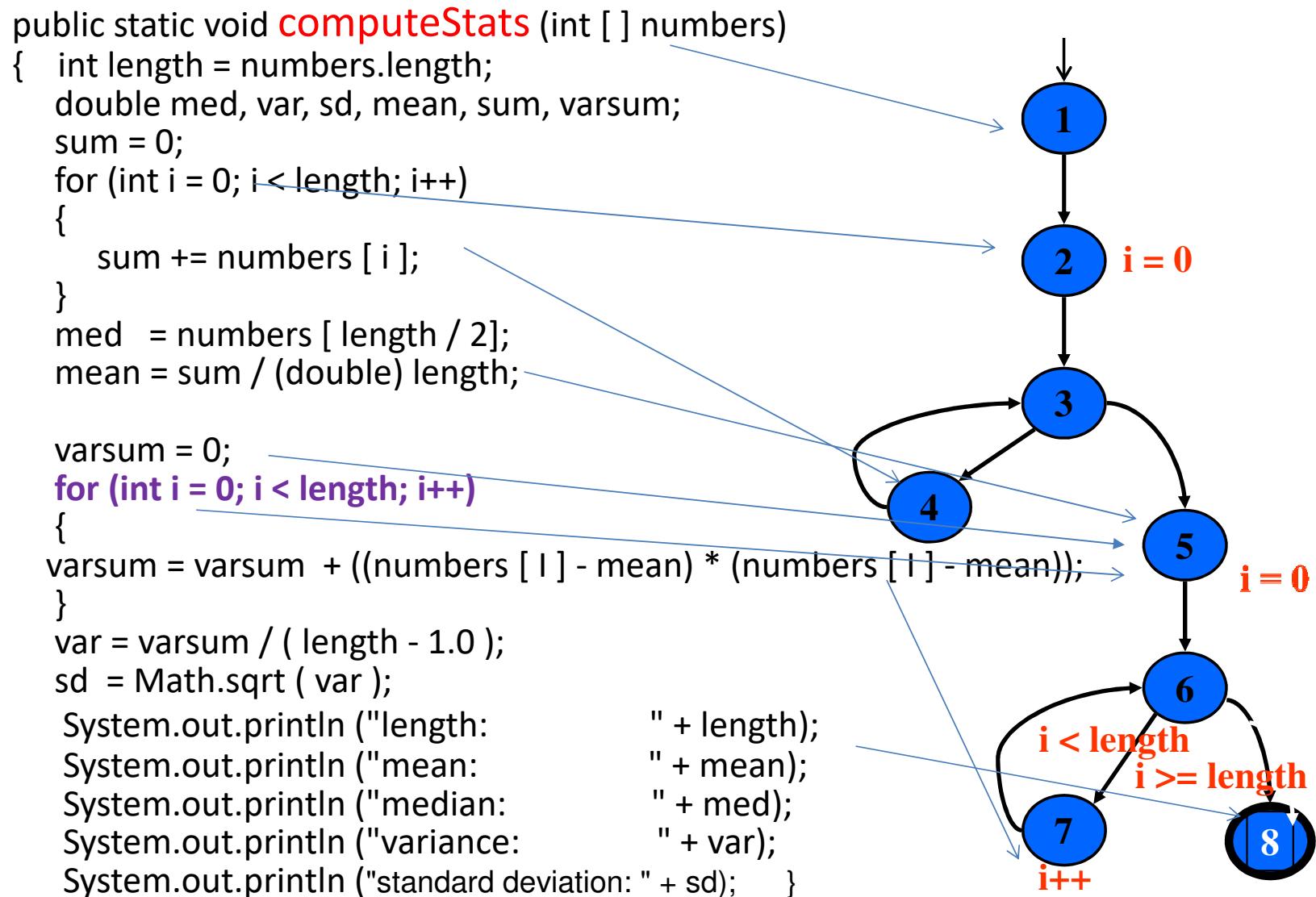
- Nodes 4 and 6 are final nodes, corresponding to the `return` statements.
- Node 2 is introduced to capture the `for` loop; it has no executable statements.
- DU (def-use) pairs are shown as a variable name followed by the def node, then the use node.



Def -Use Pairs = { (1, 1,x), (1,3,x), (1,3,y), (1, 2,i),  
(1, 3,i), (1,5,i), (1,6,i), (5, 2,i), (5, 3,i), (5, 6,i), (5, 5,i)}

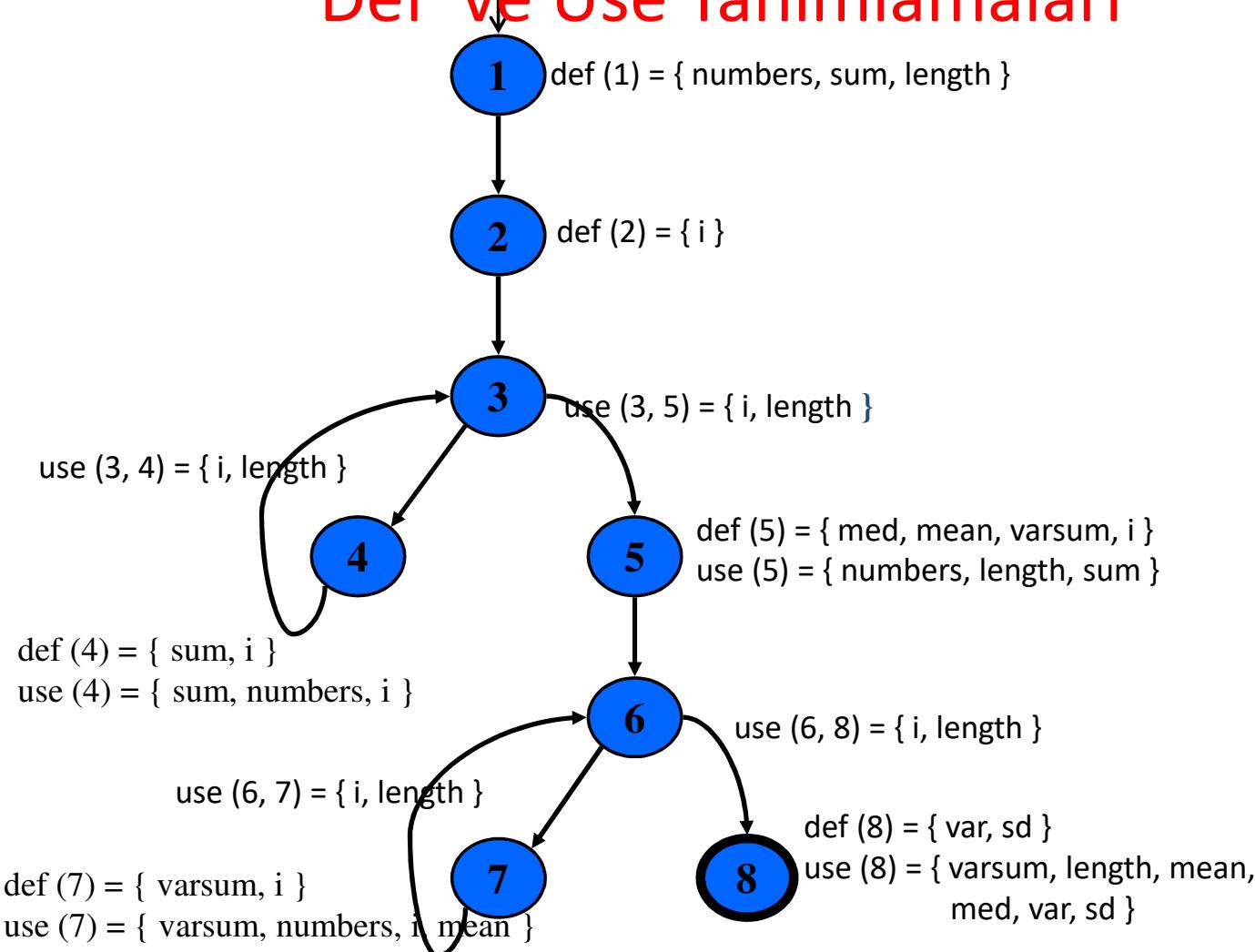
## Örnek 2: Data Flow Coverage





# Control Flow Graph ->Data Flow Graph

## Def ve Use Tanımlamaları



# Def ve Use Tablosu

Node	Def	Use	Edge	Use
1	{ numbers, sum, length }	{ numbers }	(1, 2)	
2	{ i }		(2, 3)	
3			(3, 4)	{ i, length }
4	{ sum, i }	{ numbers, i, sum }	(4, 3)	
5	{ med, mean, varsum, i }	{ numbers, length, sum }	(3, 5)	{ i, length }
6			(5, 6)	
7	{ varsum, i }	{ varsum, numbers, i, mean }	(6, 7)	{ i, length }
8	{ var, sd }	{ varsum, length, var, mean, med, var, sd }	(7, 6)	
			(6, 8)	{ i, length }

# Değişkenler : Def & Use Pairs

variable	DU Pairs	
numbers	(1, 4) (1, 5) (1, 7)	Defs come <u>before</u> uses, do not count as DU pairs
length	(1, 5) (1, 8) (1, (3,4)) (1, (3,5)) (1, (6,7)) (1, (6,8))	
med	(5, 8)	
var	(8, 8)	defs <u>after</u> use in loop, these are valid DU pairs
sd	(8, 8)	
mean	(5, 7) (5, 8)	
sum	(1, 4) (1, 5) (4, 4) (4, 5)	No def-clear path ... different scope for i
varsum	(5, 7) (5, 8) (7, 7) (7, 8)	
i	(2, 4) (2, (3,4)) (2, (3,5)) (2, 7) (2, (6,7)) (2, (6,8)) (4, 4) (4, (3,4)) (4, (3,5)) (4, 7) (4, (6,7)) (4, (6,8)) (5, 7) (5, (6,7)) (5, (6,8)) (7, 7) (7, (6,7)) (7, (6,8))	No path through graph from nodes 5 and 7 to 4 or 3

# Değişkenler: Def Use Paths

variable	DU Pairs	DU Paths	variable	DU Pairs	DU Paths
numbers	(1, 4)	[ 1, 2, 3, 4 ]	mean	(5, 7)	[ 5, 6, 7 ]
	(1, 5)	[ 1, 2, 3, 5 ]		(5, 8)	[ 5, 6, 8 ]
	(1, 7)	[ 1, 2, 3, 5, 6, 7 ]	varsum	(5, 7)	[ 5, 6, 7 ]
length	(1, 5)	[ 1, 2, 3, 5 ]		(5, 8)	[ 5, 6, 8 ]
	(1, 8)	[ 1, 2, 3, 5, 6, 8 ]		(7, 7)	[ 7, 6, 7 ]
	(1, (3,4))	[ 1, 2, 3, 4 ]		(7, 8)	[ 7, 6, 8 ]
	(1, (3,5))	[ 1, 2, 3, 5 ]	i	(2, 4)	[ 2, 3, 4 ]
	(1, (6,7))	[ 1, 2, 3, 5, 6, 7 ]		(2, (3,4))	[ 2, 3, 4 ]
	(1, (6,8))	[ 1, 2, 3, 5, 6, 8 ]		(2, (3,5))	[ 2, 3, 5 ]
med	(5, 8)	[ 5, 6, 8 ]		(4, 4)	[ 4, 3, 4 ]
var	(8, 8)	<i>No path needed</i>		(4, (3,4))	[ 4, 3, 4 ]
sd	(8, 8)	<i>No path needed</i>		(4, (3,5))	[ 4, 3, 5 ]
sum	(1, 4)	[ 1, 2, 3, 4 ]		(5, 7)	[ 5, 6, 7 ]
	(1, 5)	[ 1, 2, 3, 5 ]		(5, (6,7))	[ 5, 6, 7 ]
	(4, 4)	[ 4, 3, 4 ]		(5, (6,8))	[ 5, 6, 8 ]
	(4, 5)	[ 4, 3, 5 ]		(7, 7)	[ 7, 6, 7 ]

# Graph Coverage for Design Elements

Data Flow Graph Coverage for Source Code

# Graph Coverage for Design Elements

- ❑ Veri soyutlama ve nesneye yönelik yazılım için modülerlik ve yeniden kullanım büyük önem taşır.
- ❑ Burada tasarım elemanlarına yönelik çözümlemelerin testi de önemlidir.
  - ❖ Önceki test yöntemlerinde «modularity» ve «reuse» nerededir? sorusuna integrasyon testi cevap olabilir.
  - ❖ Zira her iki özellik te bağımsız test edilmelidir.

# Structural Graph Coverage for Design Elements

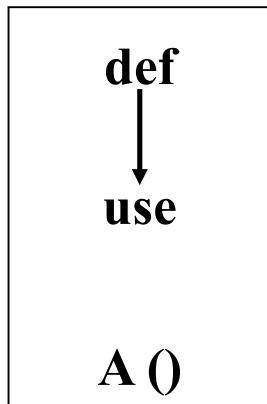
- ❑ Graph coverage for design elements is based on couplings between software components.
- ❑ *Coupling* measures the dependency relations between two units by reflecting their interconnections
  - ❖ Faults in one unit may affect the coupled unit.
- ❑ *Coupling* provides summary information about the design and the structure of the software.

# DU –Çiftlerinin Tasarım Elemanlarında Testi

- ❑ «Data flow testing criteria» temelinde «defs to uses» değerlerinin kontrolü vardır.
- ❑ def ve use aynı düğümde bulunuyorsa, DU\_pair gerçekleşmesi için önce use gerçekleşir, daha sonra def süreci başlatılır.
  - ❖ Böylece def ve use tanımlamalarının bir döngü (loop) içerisinde yapıldığı ifade edilir.
- ❑ «Data flow criteria» sınıflar arasında uygulandığında yapı karmaşıklaşmıştır.
  - ❖ Fonksiyonlar arasındaki parametre geçişlerinde isimler değişir.
  - ❖ Veriyi paylaşmanın farklı yolları olabilir.
    - ✓ def- use çiftlerini bulmak zorlaşabilir.
    - ✓ hangi def –use çiftlerinin test edileceğini bulmak güçleşir.

# def-use pairs as Normal Coupling Data Flow Analysis

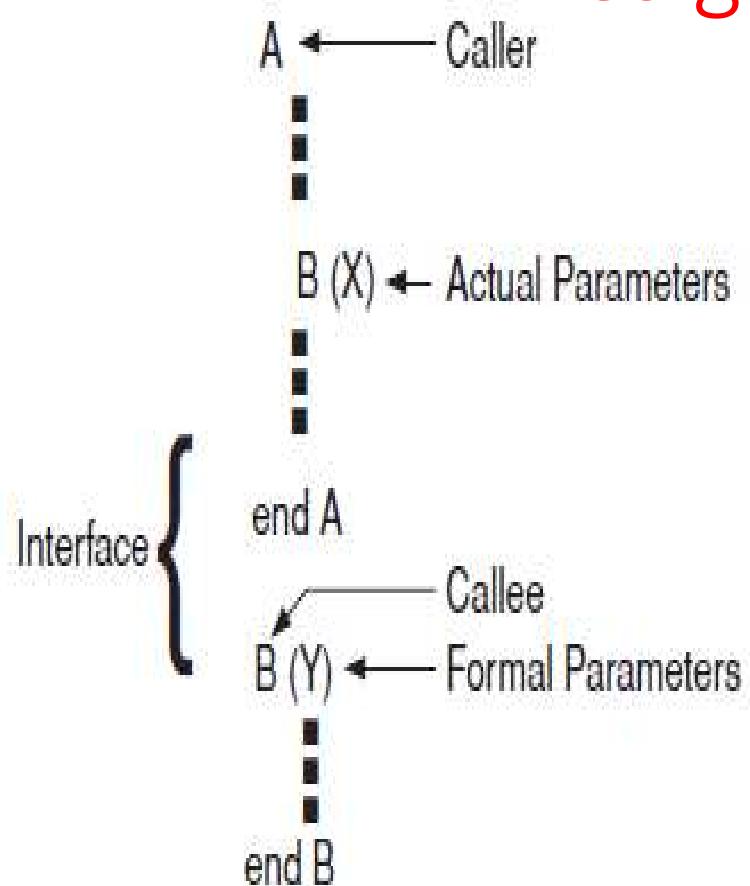
Classic Data Flow Coverage for design elements



intra-procedural data flow  
(within the same unit)

- ❑ The method A() contains **def** and **use**.
- ❑ Here the **variable** is omitted
- ❑ It is assumed that all **du** pairs refer to the **same variable**.

# Parameter Coupling between Design Elements



caller:

```
...
foo(actual1, actual2);
...
```

callee:

```
void foo(int formal1, int formal2) {
    ...
}
```

Caller : An unit that invokes another unit

Callee : The unit that is called

Callsite : Statement or node where the call appears

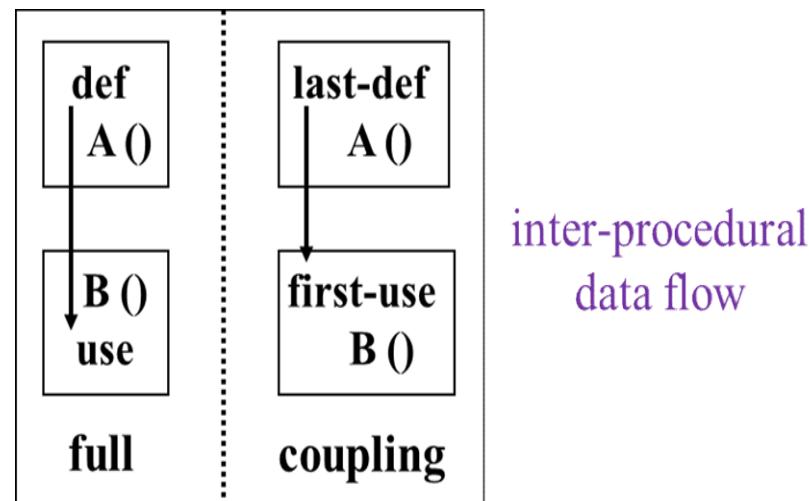
Actual parameter : Variable in the caller

Formal parameter : Variable in the callee

# def-use pairs

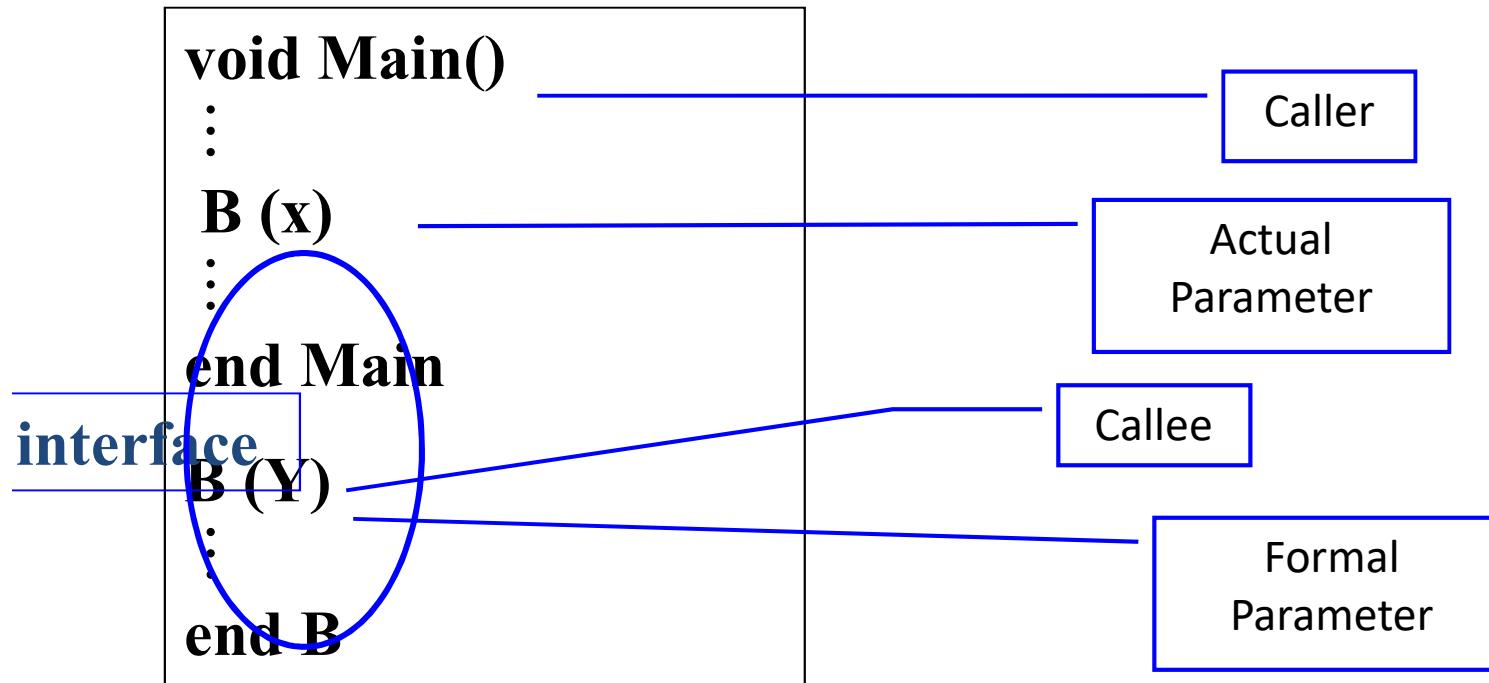
## Another classic Data Flow Coverage

All du pairs between a caller A() and a callee B()



# Data Flow Graph Coverage for Design Elements

- ❑ This technique can be limited to the **unit interfaces**
- ❑ Data flow is concerned only with **last-defs before calls** and **returns** and **first-uses after calls** and **returns**.
  - ❖ The last-defs before a call are locations with defs that reach uses at **callsites**
  - ❖ The last-defs before a return are locations with defs that reach a **return statement**.



- Tüm **def-use pairs** bulmak ve test etmek hem zordur, hem de pahallıdır.
- **Def-use pairs** teknigi arayuzler (unit interface) üzerinde sınırlanırılacaktır.
- Bir integration testidir.
  - ❖ Pek çok olasılık vardır.
  - ❖ Sadece arayüz (interface) ile sınırlanırsa amaca ulaşılacaktır.

# Nesneye Yönerek Temel Özellik Coupling testi

«Coupling» özelliğini gerçekleştiren farklı tanımlamalar vardır.

**Parameter Coupling:** relationships between caller and callees; passing data as parameters;

**Shared Data Coupling:** one unit writes to some in-memory data, another unit reads this data;

**External Data Coupling:** one unit writes data e.g. to disk, another read reads the data.

# Coupling Data-flow

- ❑ We are only testing variables that have a definition on one side and a use on the otherside
- ❑ we do not impose test requirements in the following case:

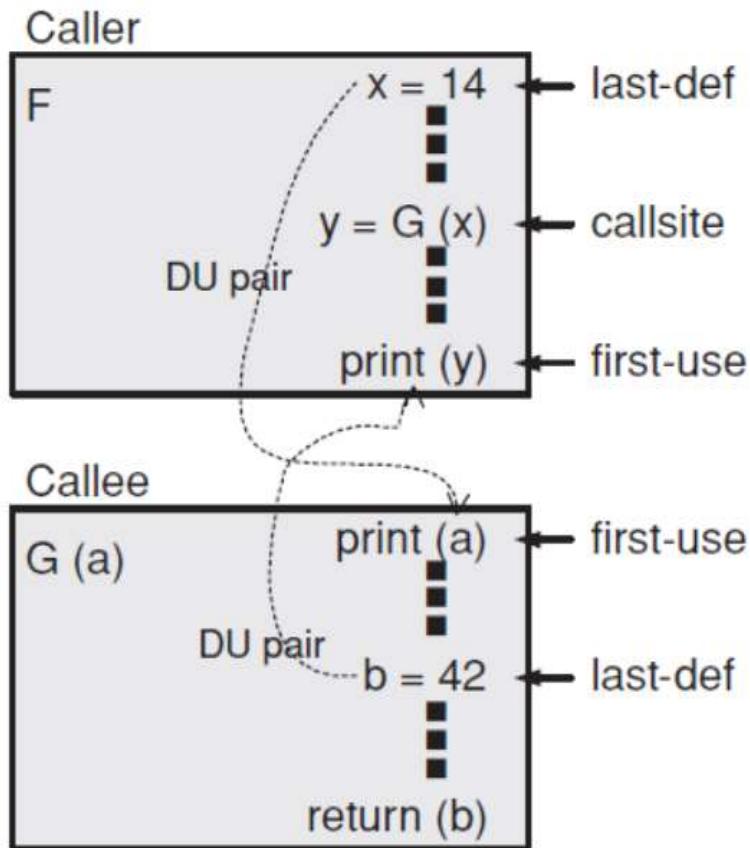
```
foo(20, 17);

foo(int x, int y) {
    return x * 5; // no use of y, so no test requirement
}
```

# Inter-Procedural DU Pairs

- ❑ If we focus on the interface, then we just need to consider the last definitions of variables **before calls and returns** and first uses inside units and **after calls**
- ❑ Last-def : The set of nodes that define a variable **x** and has a def-clear path from the node through a call site to a use in the other unit
  - Can be from caller to callee (parameter or shared variable) or from callee to caller as a return value
- ❑ First-use : The set of nodes that have uses of a variable **y** and for which there is a def-clear and use-clear path from the call site to the nodes

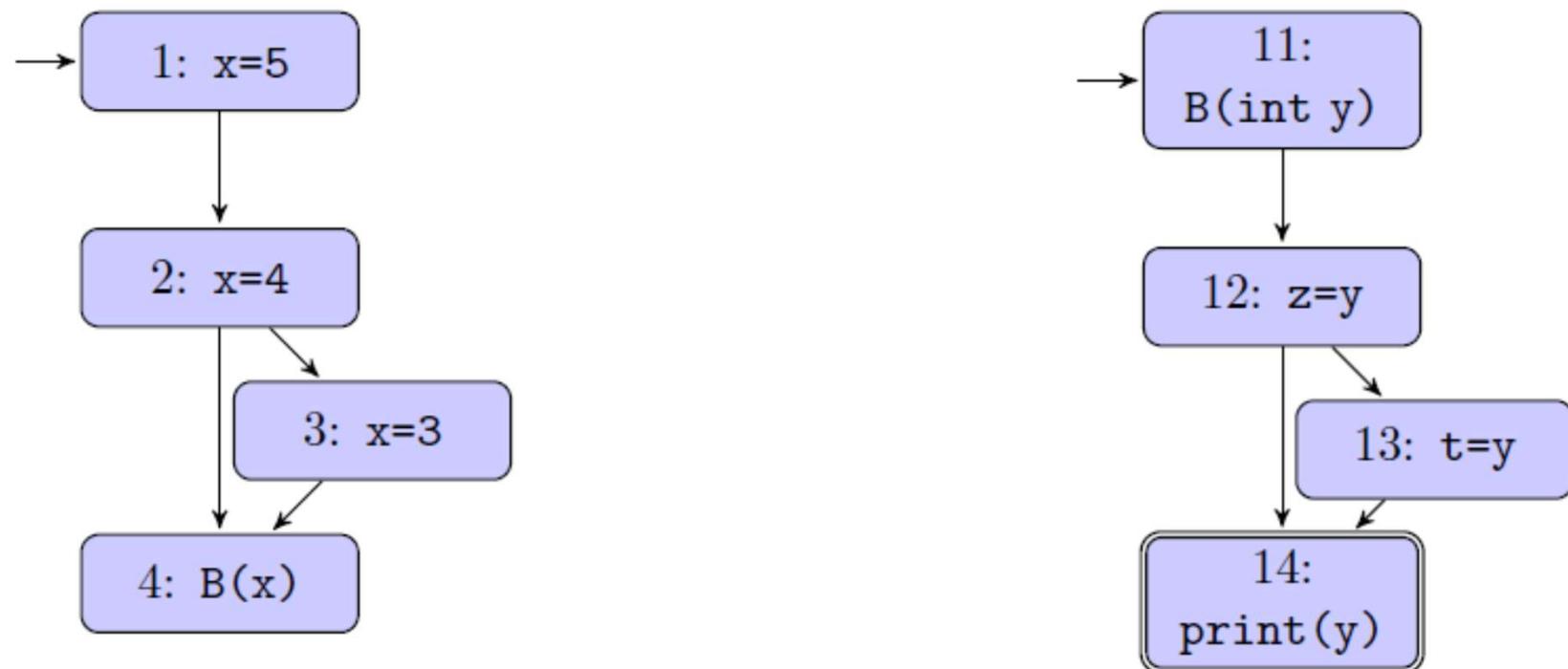
# Fonksiyonlar arası DU çiftleri last-def & first-use



The **callsite** has **two du-pairs**.

- $x$  in  $F()$  is *passed to a in  $G()$*
- $b$  in  $G()$  is *returned and assigned to y in  $F()$* .

# du-pairs between Callers and Callees



The last-defs are 2, 3

The first-uses are 12, 13

# Important for Coupling Data-Flow

- We are only testing variables that have a definition on one side and a use on the other wise
- We do **not** impose test requirements in the following case:

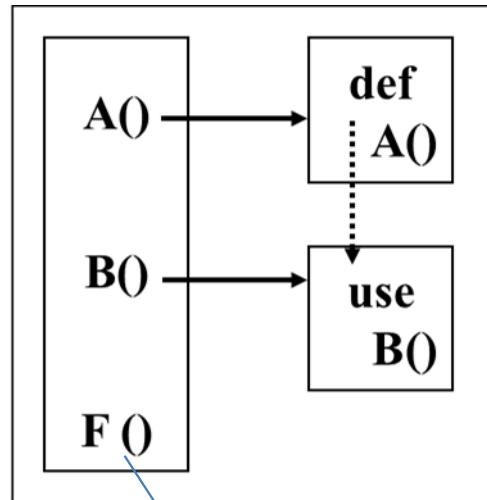
```
foo(20, 17);

foo(int x, int y) {
    return x * 5; // no use of y, so no test requirement
}
```

# def-use pairs in Object-Oriented Software

In object oriented software du- pairs are usually based on class and state variables defined for class.

- A **coupling method** F() calls two methods A() and B().
  - ❖ A() defines a **variable** and B() uses it.
- For the **variable references to the same**, both A() and B() must be called through the same instance context, or object reference.
  - ❖ For example if the callers are o.A() and o.B(), they are called through the instance context of o.
- If the calls are **not through the same instance** context, the definition and use will be different instances of the variable.



coupling method

# def-use pairs in Object-Oriented Software

In object oriented data flow testing

A() and B() could be in the **same class** , or they could be in **different classes** and accessing the **same global variables**

These are the advanced topics for **inheritance** and **polymorphism**.

## Java Example.

```
class M {  
    int x;  
    void foo() { x = 5; }  
    int bar() { return x; }  
}
```

```
void notCouplingMethod() {  
    M m = new M(), n = new M();  
    m.foo(); print (n.bar());  
}
```

- ❑ Because **m** and **n** are different objects, there is **no du-pair** here.
- ❑ We have a **last-def** of **m.x** and a **first-use** of **n.x**
  - ❖ they are different objects, it is possible but they are not **def-use pairs**.

# Inheritance, Polymorphism & Dynamic Binding and Problems for Data Flow Coverage

- Additional control and data connections make data flow analysis more complex
  - ❖ The defining and using units may be in different call hierarchies
- When inheritance hierarchies are used, a *def* in one unit could reach uses in any class in the inheritance hierarchy
- With dynamic binding, *the same location can reach different uses depending on the current type of the using object*
- The same location can have *different definitions or uses at different points* in the execution

# Example: Quadratic root program for two numbers

```
1 // Program to compute the quadratic root
2 import java.lang.Math;
3
4 class Quadratic
5 {
6     private static float Root1, Root2;
7
8     public static void main (String[] argv)
9     {
10        int X, Y, Z;
11        boolean ok;
12        int controlFlag = Integer.parseInt (argv[0]);
13        if (controlFlag == 1)
14        {
15            X = Integer.parseInt (argv[1]);
16            Y = Integer.parseInt (argv[2]);
17            Z = Integer.parseInt (argv[3]);
18        }
19        else
20        {
21            X = 10;
22            Y = 9;
23            Z = 12;
24        }
```

```
25        ok = Root (X, Y, Z);
26        if (ok)
27            System.out.println
28                ("Quadratic: " + Root1 + Root2);
29        else
30            System.out.println ("No Solution.");
31    }
32
33 // Three positive integers, finds quadratic root
34 private static boolean Root (int A, int B, int C)
35 {
36    float D;
37    boolean Result;
38    D = (float) Math.pow ((double)B,
39                          (double)C - 4.0 * A * C );
40    if (D < 0.0)
41    {
42        Result = false;
43    }
44    Root1 = (float) ((-B + Math.sqrt(D)) / (2.0 * A));
45    Root2 = (float) ((-B - Math.sqrt(D)) / (2.0 * A));
46    Result = true;
47    return (Result);
48 } //End method Root
49
50 } // End class Quadratic
```

```
1 // Program to compute the quadratic root for two numbers
2 import java.lang.Math;
3
4 class Quadratic
5 {
6     private static float Root1, Root2;
7
8     public static void main (String[] argv)
9     {
10        int X, Y, Z;
11        boolean ok;
12        int controlFlag = Integer.parseInt (argv[0]);
13        if (controlFlag == 1)
14        {
15            X = Integer.parseInt (argv[1]);
16            Y = Integer.parseInt (argv[2]);
17            Z = Integer.parseInt (argv[3]);
18        }
19        else
20        {
21            X = 10;
22            Y = 9;
23            Z = 12;
24        }

```

shared variables

last-defs

```

25      ok = Root (X, Y, Z);
26      if (ok)
27          System.out.println
28              ("Quadratic: " + Root1 + Root2)
29      else
30          System.out.println ("No Solution.");
31      }
32
33 // Three positive integers, finds the quadratic root
34 private static boolean Root (int A, int B, int C)
35 {
36     float D;
37     boolean Result;
38     D = (float) Math.pow ((double)B, (double)2-4.0)*A*C);
39     if (D < 0.0)
40     {
41         Result = false;
42         return (Result);
43     }
44     Root1 = (float) ((-B + Math.sqrt(D)) / (2.0*A));
45     Root2 = (float) ((-B - Math.sqrt(D)) / (2.0*A));
46     Result = true;
47     return (Result);
48 } /*End method Root
49
50 } // End class Quadratic

```

**first-use**

**first-use**

**last-def**

**last-defs**

The diagram illustrates the flow of variable usage from their first definition to their final use. It uses blue boxes to identify regions and blue ovals to highlight specific code points. Arrows show the flow from 'first-use' to 'last-def' and 'last-defs'.

- first-use:** Points to the first use of 'ok' at line 25 and the first use of 'Root1 + Root2' at line 28.
- first-use:** Points to the first use of 'D' at line 36 and the first use of 'Result' at line 41.
- last-def:** Points to the definition of 'Result' at line 41.
- last-defs:** Points to the definitions of 'Root1' at line 44 and 'Root2' at line 45.

# The call to `root()` on line 34 in `main` passes three parameters

- Each of the variables X, Y, and Z (*actual parameters*) have three **last-defs** in the **caller** at lines 15, 16, 17, lines 21, 22, and 23 and *they are used in the callee*
  - ❖ The variables X, Y, and Z are mapped to **formal parameters** A, B, and C in `Root()`.
  - ❖ All three variables (A, B and C) have a **first-use** at line 38.
- The class variables Root1 and Root2 are defined in the **callee** and *they are used in the caller*.
  - ❖ Their **last-defs** are at lines 44 and 45 and the **first-use** is at line 28.
- The value of local variable Result is returned to the caller, with two possible **last-defs** at lines 41 and 46, **first-use** at line 26

# Quadratic – Coupling DU-pairs

Pairs of locations: method name, variable name,  
statement

(main (), X, 15) – (Root (), A, 38)

(main (), Y, 16) – (Root (), B, 38)

(main (), Z, 17) – (Root (), C, 38)

(main (), X, 21) – (Root (), A, 38)

(main (), Y, 22) – (Root (), B, 38)

(main (), Z, 23) – (Root (), C, 38)

(Root (), Root1, 44) – (main (), Root1, 28)

(Root (), Root2, 45) – (main (), Root2, 28)

(Root (), Result, 41) – ( main (), ok, 26 )

(Root (), Result, 46) – ( main (), ok, 26 )