T.C. SANAYİ VE TEKNOLOJİ BAKANLIĞI

#MİLLİ TEKNOLOJİ HAMLESİ

TÜBİTAK

YAZILIM TESTLERİ VE TEST ODAKLI GELİŞTİRME YAKLAŞIMININ KULLANIMI

ARAŞTIRMA SERİSİ - SAYI 8

BİLGEM

YAZILIM TEKNOLOJİLERİ ARAŞTIRMA ENSTİTÜSÜ

Simge ve Kısaltmalar

Kısaltmalar	Açıklama
TÜBİTAK	Türkiye Bilimsel ve Teknolojik Araştırma Kurumu
BİLGEM	Bilişim ve Bilgi Güvenliği İleri Teknolojiler Araştırma Merkezi
YTE	Yazılım Teknolojileri Araştırma Enstitüsü
SDLC	Software Development Life Cycle (Yazılım Geliştirme Yaşam Döngüsü)
XP	Extreme Programming (Ekstrem Programlama)
TDD	Test Driven Development (Test Odaklı Geliştirme)
BDD	Behavior Driven Development (Davranış Odaklı Geliştirme)
ATDD	Acceptance Test Driven Development (Kabul Testi Odaklı Geliştirme)
UI	User Interface (Kullanıcı Arayüzü)

Yazar
Özlem GÜNİNDİ

Yayın Koordinatörü
Elif ŞENYİĞİT

Editörler
Abdulkadir Taha YAMAÇ
Sevinç KARAKAŞ
Tuğçe YILMAZ

Tasarım
Şeyma KOÇER

İletişim: 0(312) 289 92 22 - yte.bilgi@tubitak.gov.tr

yte.bilgem.tubitak.gov.tr

İçindekiler

Önsöz

TÜBİTAK BİLGEM Yazılım Teknolojileri Araştırma Enstitüsü, 2012 yılından bu yana yazılım teknolojilerinde AR-GE faaliyetleri yürüten bir araştırma kuruluşudur. Araştırma faaliyetlerinde elde ettiği birikimini stratejik, hassas ve kritik projeler yürüterek kamu adına hayata geçirmekte; kurumlarımıza dijital dönüşüm, yazılım geliştirme teknolojileri ve kalite süreçleri konusunda danışmanlık vermektedir.

Araştırma Serisi ile TÜBİTAK BİLGEM YTE kurum içi çalışmaların yaygınlaştırılması ve sektörün erişimine açılması amaçlanmaktadır. Araştırma Serisi'nde yayınlanan çalışmalar TÜBİTAK BİLGEM YTE çalışanlarının projelerde elde ettiği bilgi birikimini paylaşmak adına derlenmiştir. Bu çalışmalar ile ülkemizin yazılım sektörüne katkı sağlanması hedeflenmektedir.

Giriş

Yazılım sektöründe şirketlerin ayakta kalabilmesi için ürün kalitelerini korumaları ve müşteri, kullanıcı memnuniyet oranlarını sürekli yüksek tutmaları gerekmektedir. Ürünün kalitesi, ürünün hatalardan arındırılmış olması şeklinde karşımıza çıkmaktadır. Şirketlerin, ürünü güncel ve kaliteli bir şekilde tutabilmesi onu sürekli olarak test etmelerinden geçer.

Yazılım testi, yazılım geliştirme yaşam döngüsünün (SDLC) en önemli aşamalarından biridir. Testler yazılımın, kendisinden beklenen özellikleri karşılayıp karşılamadığını incelemek amacıyla yapılır. Bu şekilde yazılımdaki hatalar bulunup düzeltilebilir ve gereksinimlere uygun hale getirilebilir. Yazılım testi hatayı erken tespit etme ve hata önleme faaliyetleri içerdiği için uzun vadede maliyeti düşürür. Aynı zamanda ürünün kalitesini yükseltir, müşterinin memnuniyetini ve güvenini kazanmayı sağlar. İyi yazılmış, okunaklı testler; incelediğimiz fonksiyona veya bileşene, hangi girdiler verildiğinde hangi çıktıların alacağını gösteren örnekler barındırır. Bu yüzden test edilen kodun da iyi bir dokümantasyonu sayılabilir.

Yazılım geliştirme yaşam döngüsü; gereksinim analizi, tasarım, geliştirme, test ve bakım olmak üzere birçok aşamadan oluşmaktadır. Her aşama da testten geçmektedir. Bu nedenle çeşitli test seviyeleri ve teknikleri vardır. Üzerinde çalışılan projelerde müşterilere, son kullanıcılara; kaliteli, hatadan arındırılmış ve müşteri memnuniyetinin yüksek seviyelerde tutulabildiği ürünler sunmak istenir. Bu nedenle ürün üzerinde, test seviyelerini ve tekniklerini uygulamak gerekmektedir. Yazılım testleri yazılıp, uygulanırken birçok farklı yaklaşım bulunmaktadır. Bu yaklaşımlardan bazıları test odaklı geliştirme, davranış odaklı geliştirme, kabul testi odaklı geliştirmedir.

Ekstrem programlama (XP) gibi çevik süreçlerde sistem hatalarının oluşmasını engellemek ve kaliteyi yüksek tutabilmek için test odaklı geliştirme yaklaşımı geliştirilmiştir. Test odaklı geliştirme yaklaşımı bir test türü değildir. Daha çok yazılım testlerini uygulamak için izlenen bir yoldur. Test odaklı geliştirme; kodlama, test ve tasarımın birlikte çalıştığı bir programlama tarzını ifade eder. Test odaklı geliştirme yaklaşımında önce test kodlarının yazılması daha sonra yazılım kodlarının yazılması gerekmektedir. Bu geliştirme metodunda testler hazırlanırken sistemin nasıl çalışması gerektiği düşünüldüğü için sadece gerekli olan yapılar için zaman harcanır. Bu çalışmada yazılım testleri, test yöntemleri ve yaklaşımları incelenerek, bir yazılım test yaklaşımı olarak TDD'nin yazılım geliştirme sürecinde nasıl uygulanacağı açıklanmıştır.

Yazılımda Test Kavramları

Uzun vadeli sürdürülebilirlik, rekabet üstünlüğü, kullanılabilirlik, müşteri beklentileri, veri güvenliği, maliyet kontrolü gibi nedenler yazılım ürünü için önemli faktörlerdir. Yazılım ürününün sadece anlık başarı odaklı değil, uzun vadeli başarı ve sürdürülebilirlik için kalite, güvenilirlik ve performans odaklı olması gerekmektedir. Bu nedenle ürün güvenilir, yüksek performanslı, düşük maliyetli olmalı ve müşteri memnuniyeti ve güvenilirliği sağlanmalıdır. Uygun bir şekilde test edilen yazılım ürünü sayesinde belirtilen özellikler sağlanabilir.

Yazılımda testin amaçları; ürünü oluştururken gereksinimlerin karşılandığından emin olmak, ürünün test işleminin tamamlandığını ve paydaşların beklediği şekilde çalıştığını onaylamak, ürünün kalite düzeyine karşı güven oluşturmak, hataları bulmak, yetersiz yazılım kalite riskini azaltmak olarak sıralanabilir.

Yazılım testleri geliştirilirken; test odaklı geliştirme, davranış odaklı geliştirme, kabul testi odaklı geliştirme gibi birçok farklı yaklaşım bulunmaktadır. Yazılım test yaklaşımlarını daha iyi anlayabilmek için test seviyeleri ve test teknikleri hakkında bilgi sahibi olmak faydalı olacaktır.

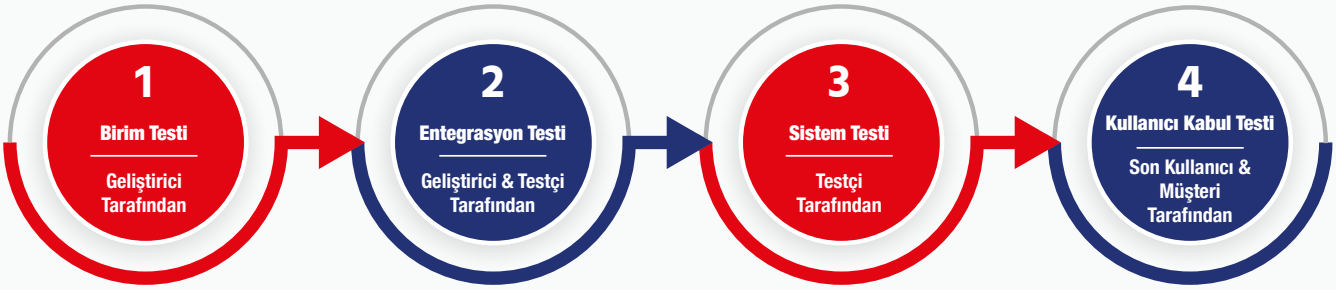
1. Yazılımda Test Seviyeleri

Test, bir sistemin veya sistemin bileşenlerinin belirtilen gereklilikleri karşılayıp karşılamadığını öğrenmek amacıyla yapılan bir değerlendirme sürecidir. Bu süreç yazılım geliştirme uzmanları tarafından başlayan ve son kullanıcıya kadar uzanan teknik seviyelerden oluşur.

Başlıca dört test seviyesi vardır:

- Birim testi (unit testing)
- Entegrasyon testi (integration testing)
- Sistem testi (system testing)
- Kabul testi (acceptance testing)

Şekil 1'de yazılım test seviyeleri gösterilmiştir.



Şekil 1. Test Seviyeleri

1.1. Birim Testi

Birim veya bileşen testi en temel test türüdür. Birim testi, yazılımın her bir bileşenini izole ederek işleyişini kontrol eder. Birim test genellikle yazılım geliştiriciler tarafından geliştirme ortamında yapılır. Bu tür testler yazılım geliştirme sürecinin en erken aşamalarında yapılmalı ve yazılım test ekibine bu şekilde teslim edilmelidir.

Birim testinin amacı, her bir modülü veya birimi izole ederek bu kısmın gereksinimler ve işlevsellik açısından doğru olduğunu göstermektir. Birim testi yazmak kodda yeniden düzenleme (refactor) işlemini yapmayı kolaylaştırır. Kodda değişiklik yapıldığında, birim test çalıştırılarak oluşturulan algoritmaya uygun bir şekilde çalışıp çalışmadığı kolaylıkla test edilir. Ancak birim testler tüm hataları ortaya çıkarmaz, çünkü her parça izole şekilde test edilmektedir.

Birim testin ne olduğunu anlamak için, Michael Feathers'ın birim testin "ne olmadığı" ile ilgili özetine bakılmalıdır:

Eğer bir testin,

- Veri tabanı ile etkileşimi varsa,
- Ağ üzerinden iletişimi varsa,
- Dosya sistemine müdahalede bulunuyorsa,
- Başka bir birim test ile aynı anda koşturulmaya çalışılıyorsa,
- Testin koşması için ortam değişkenleri ya da yapılandırılma dosyalarının değişmesi gerekiyorsa bu test birim test değildir.

Birim testi yazarken uymamız gereken bazı kurallar vardır:

- En küçük parçacık test edilmeli,
- Sadece bir senaryo test edilmeli,
- Test metodu ismi test edilen senaryonun yansıması olmalı,
- Test edilen kısım diğer kısımlardan bağımsız olmalı,
- Hızlı çalışabilmeli ve çabuk sonuçlar vermeli,
- Okunaklı, anlaşılabilir ve sürdürülebilir olmalıdır.

1.2. Entegrasyon Testi

Entegrasyon testi uygulamaya ait farklı bileşenlerin birbirleri ile etkileşiminin test edildiği test seviyesi olarak tanımlanır. Entegrasyon testleri, birden fazla birimi veya modülü gruplar halinde test ederek hataları oluştuğu konumda tespit eder ve hatalara ait esas sebeplerin kolaylıkla belirlenebilmesini sağlar. Ayrıca modüller arasındaki arayüzleri test edilerek ve farklı modüllerin birbileri ile iletişiminden kaynaklanan kritik hataları ortaya çıkarır.

Entegrasyon testinin amacı, bağımsız modüllerin diğer modüllerle entegrasyonları sağlandıktan sonra beklendiği gibi çalıştığından emin olmaktır. Entegrasyon noktalarının yoğun olduğu yazılım geliştirme projelerinde birim testler ile birlikte entegrasyon testi için kullanılan uçtan uca işlevsel iş akışı testleri kullanır. Birim test sırasında ayrı ayrı test edilen bileşenler birbirine entegre edildikleri zaman hataya sebep olabilirler. Entegrasyon testi, sistemin farklı bileşenlerinin (birimlerinin) birlikte doğru çalışıp çalışmadıklarını test etmeyi amaçlar.

Entegrasyon testlerinde test metodolojileri genellikle büyük ölçüde değişebilir, ancak bunlar iki temel test stratejisi çerçevesinde uygulanır:

- **Büyük Patlama Entegrasyon Testi (Bigbang Integration Test)**

Büyük patlama entegrasyon testi, yazılım ekiplerinin tüm bireysel modüller geliştirildikten sonra gerçekleştirebileceği bir entegrasyon testi türüdür. Büyük patlama testi gerçekleştirilirken, tüm modüller tek bir yazılım sistemi oluşturacak şekilde birleştirilir ve artımlı entegrasyon testinin tek seferlik yapısının aksine eş zamanlı olarak test edilir. Büyük patlama entegrasyon testi, bir hata ortaya çıktığında, hatanın yeri ve nedeni konusunda kafa karışıklığına daha az sebep olan daha küçük sistemler için uygundur.

- **Artımlı Entegrasyon Testi (Incremental Integration Test)**

Artımlı entegrasyon testi aşağıdan yukarıya (bottom-up testing) ve yukarıdan aşağıya (top-down testing) olmak üzere iki temel stratejiye göre gerçekleştirilir. Yukarıdan aşağıya entegrasyon, entegrasyon testinin sistem yığınının en üstünden yazılım mimarisinin her katmanına doğru gerçekleştirildiği bir test yaklaşımıdır. Testin kontrol akışı, kullanıcı arayüzü (UI) ile başlayıp yazılım veri tabanında sona erecek şekilde yukarıdan aşağıya doğru hareket eder. Aşağıdan yukarıya entegrasyon testi, mimarideki en alt modülden başlayarak ve yukarıya doğru çalışarak tek tek bileşenlerin test edildiği ve entegre edildiği bir süreçtir. Aşağıdan yukarıya entegrasyon testi, ekiplerin üst düzey modüller henüz geliştirme aşamasındayken teste başlamasına olanak tanır.

1.3. Sistem Testi

Sistem testi, eksiksiz ve entegre bir sistemde gerçekleştirilir. Gereksinimlere göre sistemin uygunluğunun kontrol edilmesi sağlanır. Bileşenlerin genel etkileşimi test edilir. Yük performans ve güvenlik testlerini içerir.

Sistem testi, sistemin şartnameye uygun olduğunu doğrulamak için yapılan son testtir. Bu yüzden test için hem fonksiyonel hem de fonksiyonel olmayan gereksinimler değerlendirilir. Fonksiyonel gereksinimlerle ilgili sistem testi, test edilecek sistem için en uygun kara kutu (black box) teknikleri kullanılarak başlar. Ardından, beyaz kutu (white box) tekniklerine geçilerek kara kutu testlerinin yakalayamadığı hatalar yakalanabilir.

Sistemin çalışacağı ortamla ilgili hata riskini en aza indirmek için sistem testinde test ortamı mümkün olduğunca canlı ortama yakın olmalıdır. Sistem testi, entegrasyon testinden sonra ve kabul testinden önce gerçekleştirilir. Sistem testleri, yazılım test ekibi tarafından düzenli olarak gerçekleştirilir ve sistemin geliştirme sırasında önemli aşamalarda olması gerektiği gibi çalıştığından emin olunur.

Test uzmanları, ayrı modüller ve bileşenler birbirine entegre edildikten sonra sistemin hem işlevsel hem de işlevsel olmayan gereksinimlerini değerlendirmek için sistem testlerini gerçekleştirir. Sistem testi sırasında bir yazılım yapısını tam olarak değerlendirmek için yazılım kodunun programlanması ve yapısı hakkında herhangi bir bilgiye ihtiyaç duymazlar. Bunun yerine test uzmanları, basitçe yazılımın performansını bir kullanıcının bakış açısından değerlendirmektedir.

1.4. Kullanıcı Kabul Testi

Kullanıcı kabul testleri, yazılımın gerçek dünya senaryolarına dayalı olarak test edilmesi için tasarlanmıştır. Bu nedenle, testler gerçek kullanıcılar tarafından gerçekleştirilir.

Kullanıcı kabul testleri aşağıda sıralanan adımları içermektedir:

- Test senaryolarının hazırlanması: Test senaryoları, yazılımın gerçek dünya senaryolarına dayalı olarak test edilmesini sağlar.
- Kullanıcı seçimi: Kullanıcı kabul testleri için uygun kullanıcılar seçilir.
- Test ortamının hazırlanması: Kullanıcı kabul testlerinin yapılacağı test ortamı hazırlanır.
- Test senaryolarının uygulanması: Kullanıcıların yazılımı gerçek kullanım senaryolarına göre kullanmalarını içerir.
- Geri bildirim toplama: Kullanıcı kabul testleri sırasında kullanıcılardan geri bildirimler toplanır. Bu geri bildirimler, yazılımın kullanılabilirliğini ve işlevselliğini değerlendirmeye yardımcı olur.
- Geri bildirimlerin analizi: Toplanan geri bildirimler analiz edilir. Bu analiz, yazılımın kullanılabilirliği ve işlevselliği hakkında daha detaylı bir görüş sağlar.
- Sorunların giderilmesi: Kullanıcı kabul testleri sırasında tespit edilen sorunlar giderilir.
- Son testlerin yapılması: Kullanıcı kabul testleri sırasında tespit edilen sorunlar giderildikten sonra son testler yapılır. Bu testler, yazılımın kullanılabilirliği ve işlevselliğini son kez kontrol etmek için kullanılır.

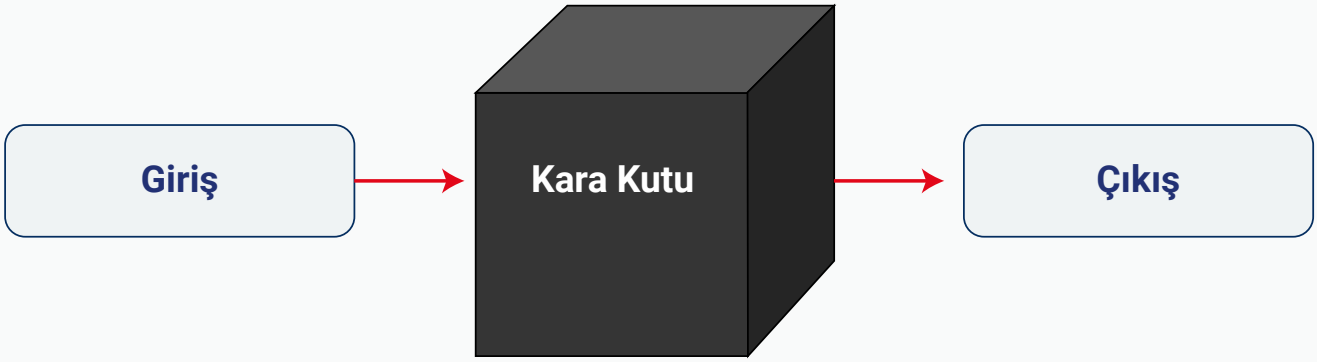
Kullanıcı kabul testleri, yazılım geliştirme sürecinde birçok avantaj sağlar. Bunlar; yazılımın gerçek dünya senaryolarına dayalı olarak test edilmesi, sorunların erken tespit edilmesi, daha yüksek müşteri tatmini ve daha iyi yazılım kalitesi sağlamasıdır.

2. Yazılım Test Teknikleri

Yazılım test teknikleri, yazılım geliştirmede yazılan kodların kontrollerini yapmak ve fonksiyonların çalışma durumlarını sınamak için kullanılan testlerdir. Test teknikleri etkili bir şekilde test durumlarını tasarlamak için kullanılabilir.

2.1. Kara Kutu Testi

Test edilen yazılımın iç işleyişi düşünülmeden yapılan testlere kara kutu testleri denir. Kara kutu testinde bir yazılım parçası test ediliyorsa, test eden, bu yazılım parçasının girdisini ve buna karşılık sistem çıktısını bilir. Fakat bu çıktıya nasıl ulaşıldığıyla ilgilenmez. Çünkü kara kutu testinin amacı, verilen girdilerde istenilen çıktının elde edilmesidir.



Şekil 2. Kara Kutu Test Yaklaşımı

Kara kutu test tekniğiyle geliştirilen yazılım içerisinde şu hata türleri tespit edilmeye çalışılır:

- Doğru olmayan veya hiç işlevi olmayan işlevlerin tespiti
- Arayüz hataları
- Performans hataları
- Veri tabanlarına ulaşma hataları veya veri yapılarındaki hatalar
- İlklendirme veya sonlandırma hataları
- Sınır değer hataları

Avantajları:

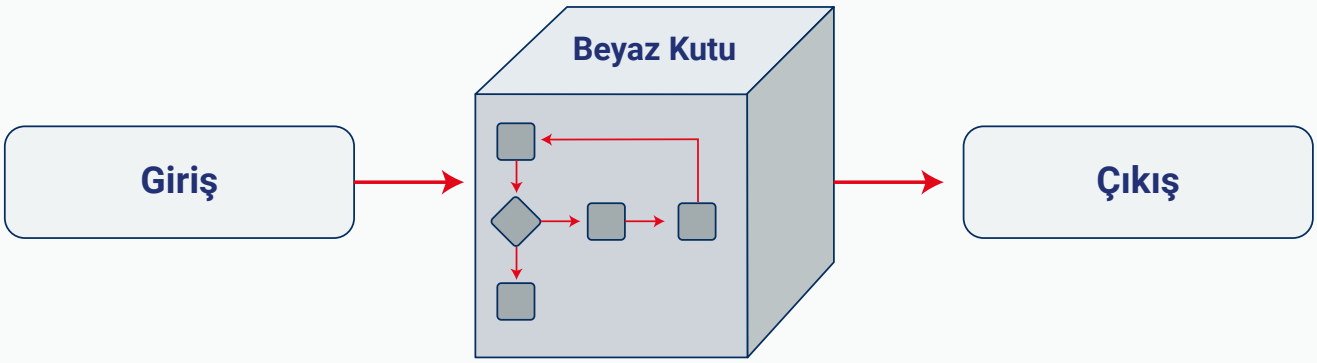
- Yazılımlarda hataların erken bulunması için etkin ve hızlı bir tekniktir.
- Test durumları yazılırken gereksinimlerden hareket edildiği için gereksinimlerdeki tutarsızlıkların ve belirsizliklerin belirlenmesinde önemlidir.
- Test uzmanlarının yazılımın ayrıntılarını bilmesine ihtiyacı yoktur.
- Test uzmanları ve geliştiriciler birbirinden bağımsız çalışabilirler.
- Test uzmanları yazılıma kullanıcı gözüyle baktığı için geliştiriciler tarafından fark edilemeyen pek çok olası hatanın ve eksikliğin bulunmasında yardımcı olur.

Dezavantajları:

- Kara kutu testleri yazılımın belirli bir kısmını hedef almadan yapıldığı için birçok hata tespit edilmeden kalabilir.
- Belirli sayıda girdi ile test yapıldığı için, fazla girdi ile deneme yapmak zaman alır.
- Anlaşılır olmayan gereksinimlerin test durumlarını tasarlamak ve testlerini yapmak bu teknikte zordur.

2.2. Beyaz Kutu Testi

Beyaz kutu testleri yazılımın iç yapısı bilerek tasarlanır. Bu nedenle beyaz kutu testlerini gerçekleştirenler genellikle sistemin iç yapısını bilen geliştiricilerdir. Beyaz kutu testiyle programın iç yapısındaki birimlerin içindeki hatalar araştırılır. Kaynak kod, beyaz kutu testlerinin en önemli girdisi olduğundan, koda ulaşım olmadan beyaz kutu testleri yapılamaz. Beyaz kutu test tekniğinde sınanan birimin veya modülün belirlenen girdiye beklenen çıktıyı nasıl verdiği, kod içinde hangi yollardan geçildiğini de bakılır.



Şekil 3. Beyaz Kutu Test Yaklaşımı

Avantajları:

- Kod içerisinde gizli kalmış mantıksal hatalar bulunur.
- Yazılan kodun optimizasyonuna katkıda bulunulur.
- Kaynak kodun analiz edilmesi ve bu analize göre testlerin gerçekleştirilmesiyle yazılım içerisindeki hatalar daha erken aşamada ve daha hızlı bulunur.
- Yazılım geliştiricilerin kod geliştirme yetenekleri desteklenir ve güçlendirilir.

Dezavantajları:

- Birim tümleştirme testleri test ekibi tarafından yapılacaksa, kodun iç yapısının bilinmesi gerektiği için maliyet artar.

3. Yazılımda Test Yaklaşımları

Test yaklaşımları; yazılımın kalitesini artırmak, yazılımın gereksinimlere uygunluğunu ve kodun güvenilirliğini sağlamak amacıyla uygulanır. Hangi yaklaşımın uygulandığı, projenin gereksinimlerine, ekibin tercihlerine ve projenin karmaşıklığına bağlı olarak değişir.

Test odaklı geliştirme, davranış odaklı geliştirme ve kabul testi odaklı geliştirme üç önemli yaklaşımdır.

3.1. Test Odaklı Geliştirme (Test Driven Development - TDD)

TDD, kodun ne yapacağını belirlemek ve doğrulamak için test senaryolarının geliştirildiği bir yazılım geliştirme yaklaşımıdır. Test odaklı geliştirme; kodlama, test ve tasarımın birlikte çalıştığı bir programlama tarzını ifade eder.

Test odaklı yazılım geliştirme süreci, kod yazılmadan önce birim testlerin yazılmasını içerir. Yazılım geliştirici için bu süreç, bir problemi çözmek için belirli bir algoritma yazılmadan önce kodun analizinin yapılmış olması ve davranışını bilmesi gerektiği anlamına gelir. Bu süreç ile test yazılırken, test bir işlevin çıktısı olan bazı koşullardan geçecektir. Ayrıca test, söz konusu kodun tüm satırlarını kapsayacak şekilde yazılmalıdır. Bu da kod kapsamının yazılan tüm kod satırlarının yüksek bir yüzdesi olduğundan emin olunması gerektiği anlamına gelir.

3.2. Davranış Odaklı Geliştirme (Behavior Driven Development - BDD)

Davranış odaklı geliştirme ilk olarak 2006 yılında Dan North tarafından ortaya atılmıştır. BDD, TDD gibi prensip olarak öncelikle test kodları yazılsın daha sonrasında proje kodu yazılsın anlayışını benimsemektedir. Yazılım süreçlerinin daha test odaklı gitmesini sağlayan bir yaklaşımdır. Ancak testler, esas olarak sistem davranışına dayanmaktadır. BDD, TDD'nin karmaşıklığını gidermek amacıyla çıkmıştır.

İkisi arasındaki en büyük fark;

- TDD, diğer sınıflara (class) bağımlılığı az olan, sınıfların sorumluluklarını ayırarak tekrar etmeyen bir kod yazılmasını sağlar.
- BDD ise, bir sistemin davranışını, bu davranışın nasıl geliştirildiğine dair ayrıntılara yer vermeden tanımlanmasına izin verir.

TDD bazı koşulları ileri sürerken, BDD sistemin davranışını açıklamak için gerçek gereksinimlerin gerçek zamanlı örnekleri kullanılarak düz metin olarak yazılır. Bu sayede yazılan bu metinler ile uygulamanın güncel bir dokümanının oluşması da sağlanmış olur.

Test senaryolarını yazarken Given, When, Then gibi komutlar kullanılır.

- **Given:** Belirli bir senaryo yazılır.
- **When:** Bir eylem gerçekleşir.
- **Then:** Çıktı sağlanır.

Uygulama için kabul testleri yazmak amacıyla kullanılan BDD çerçevesine dayalı bazı araçlar vardır. Bunlardan en bilineni Cucumber'dir. Cucumber ile iş analistleri, geliştiriciler ve test uzmanları için kolayca okunabilir ve anlaşılabilir formatta işlevsel doğrulamanın otomasyonu yapılabilir.

3.3. Kabul Testi Odaklı Geliştirme (Acceptance Test Driven Development - ATDD)

Gereksinimlerin ve kabul kriterlerinin doğrulanması bu yaklaşımın ana odağıdır. İletişim, işbirliği ve berraklık esası ile hareket edilir. Tüm paydaşların (geliştiriciler, analistler, son kullanıcılar vs.) katılımı ile kabul kriterleri tanımlanır. Kabul testleri, yazılım bitmiş sayılmadan önce mutlaka başarılı şekilde geçmesi gereken test seti olarak tanımlanır. Geleneksel olarak yazılım geliştirme sürecinin sonunda çalıştırılır.

Ortak olarak tanımlanan kabul testi, geliştirmeye başlamadan önce otomatize edilir. Tüm proje üyelerinin nelerin tamamlanması gerektiğini anlamasını sağlar. ATDD yaklaşımında otomatize edilmiş testler, geleneksel olarak sonra çalıştırılma yerine proje boyunca çalıştırılabilir.

Doğası gereği davranış odaklı geliştirmeye benzer, ancak kabul testi odaklı geliştirme, esas olarak sistemin işlevsel davranışını tahmin etmeye odaklı, yazılımın gereksinimlerini karşılayan koda odaklanır. Amaç, uygulanabilir testler yapmaktır. Kodda değişiklik yapıldığından otomatik olarak yürütülür. BDD gibi, testler düz metin olarak yazılır.

ATDD'nin avantajları şu şekilde sıralanabilir:

- Gereksinimler, herhangi bir belirsizlik olmadan çok net bir şekilde analiz edilir.
- Tüm ekip genelinde iletişimin iyi olmasını sağlar.
- Kabul testleri, yazılımı ulaşması gerektiği noktaya doğru yönlendirir ve bir kılavuz görevi görür.
- ATDD'nin otomasyonu geri bildirim süresini azaltabilir.

3.4. Test Yaklaşımları Arasındaki Farklar

- TDD daha tekniktir ve özelliğin uygulandığı aynı dilde yazılır. Örneğin Java ile uygulanıyorsa, JUnit testleri yazılır. BDD ve ATDD ise basit İngilizce dilinde metin olarak yazılır.
- TDD yaklaşımı, bir özelliğin uygulanmasına odaklanır. BDD, özelliğin davranışına odaklanır. ATDD ise gereksinimleri yakalamaya odaklanır.

- TDD'yi uygulamak için teknik bilgiye sahip olmamız gerekir. BDD ve ATDD ise herhangi bir teknik bilgi gerektirmez. BDD ve ATDD'nin faydası, bu özelliği geliştirmeye hem teknik hem de teknik olmayan kişilerin katılabilmesidir.
- TDD geliştiğinden, iyi tasarım için alan sağlar ve gereksinim karşılama yönüne odaklanır. BDD ve ATDD ise kullanıma uygun olan farklı bir yönüne odaklanır.

Bundan sonraki bölümde TDD detaylı olarak açıklanmış ve uygulama yöntemine yer verilmiştir.

Test Driven Development Yaklaşımı

TDD, Kent Beck tarafından XP'nin bir parçası olarak bulunmuş olan bir programlama tekniğidir. Tarih açısından bakıldığında 1994'te Kent Beck, Smalltalk için SUnit test çerçevesini (framework) yazmıştır. 1998 yılında XP'de "testlerin çoğunu ilk sırada yazmalıyız" diye ifade ettiği Önce Test (Test First) kavramını ortaya atmıştır. 2002 Kasım'ında piyasaya çıkan "Test Driven Development: By Example" isimli kitabında Önce Test (Test First) kavramını Test Driven Development olarak daha da detaylı bir teknik olarak ele almıştır.

Kent Beck "Test Driven Development: By Example" isimli kitabında test odaklı geliştirmeyi şu şekilde tanımlamaktadır:

"Test odaklı geliştirme, herhangi bir yazılım mühendisinin takip edebileceği, basit tasarımı ve güven veren test paketlerini teşvik eden bir dizi tekniktir. "

Kent Beck aynı kitapta TDD için şu iki kuralı tanımlamaktadır:

1. Herhangi bir kod yazmadan önce çalışmayan bir test yazın.
2. Tekrarlanmış kodu kaldırın.

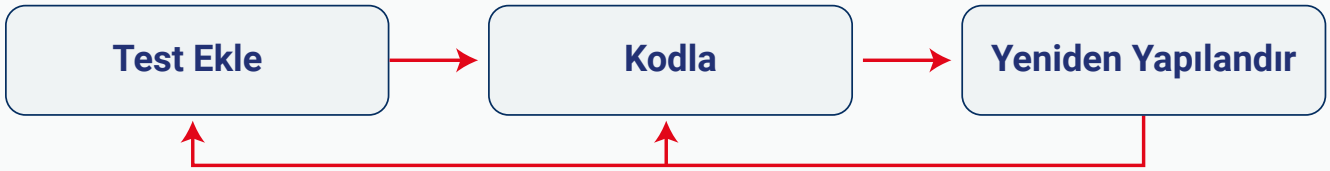
Michael Feathers "Working Effectively with Legacy Code" adlı 2005 yılında yayınlanan kitabında, testin gerekliliğinden şöyle bahsetmiştir:

"Testi olmayan kod, kötü koddur. Ne kadar iyi yazılmış olduğu önemli değil; ne kadar güzel veya nesne yönelimli veya iyi kapsülleme yapılmış olduğu da önemli değil. Testlerle, kodumuzun davranışını hızlı ve doğrulanabilir bir şekilde değiştirebiliriz. Onlar olmadan, gerçekten kodumuzun daha iyi mi yoksa daha kötü mü hale geldiğini bilmiyoruz."

TDD geliştirme süreci, çevik (agile) süreçler ile geliştirme sürecine dahil olmuştur. Daha önce şelale (waterfall) modeli yönetim sürecindeki aşamalar Şekil 4'te sunulmuştur.

Şekil 4. Şelale (Waterfall) Modeli Aşamaları

Waterfall modeli yönetim sürecinde ilk önce program için gerekli tasarım oluşturulur. Sonra bu tasarım uygulanır. Son aşama olarak sistem hatalarını bulabilmek için testler yapılır. TDD bu süreci tam tersine çevirmektedir. Şekil 5'te TDD aşamalarına yer verilmiştir.



Şekil 5. TDD Aşamaları

TDD'nin temelinde yatan ise kodu yazmaya başlamadan önce ona ait bir test yazmak ve daha sonra bu testi karşılayacak kadar yeterli kodu yazmaktır. Bu yöntemdeki ilk adım kodun başarısız olmasına yetecek kadar kısa bir test yazmaktır. Daha sonra bu testleri koşturarak başarısız olduğundan emin olduktan sonra, ikinci adım kodu güncelleyerek bu testin geçmesini sağlamaktır. Testleri tekrar koştuktan sonra eğer kodumuz halen başarısız ise kodu güncelleyip yeniden test etmek gerekir. Kod gözden geçirilir dizayn ve gereksinimlere uymayan kısımları düzeltilir.

1. Uygulama Yöntemi

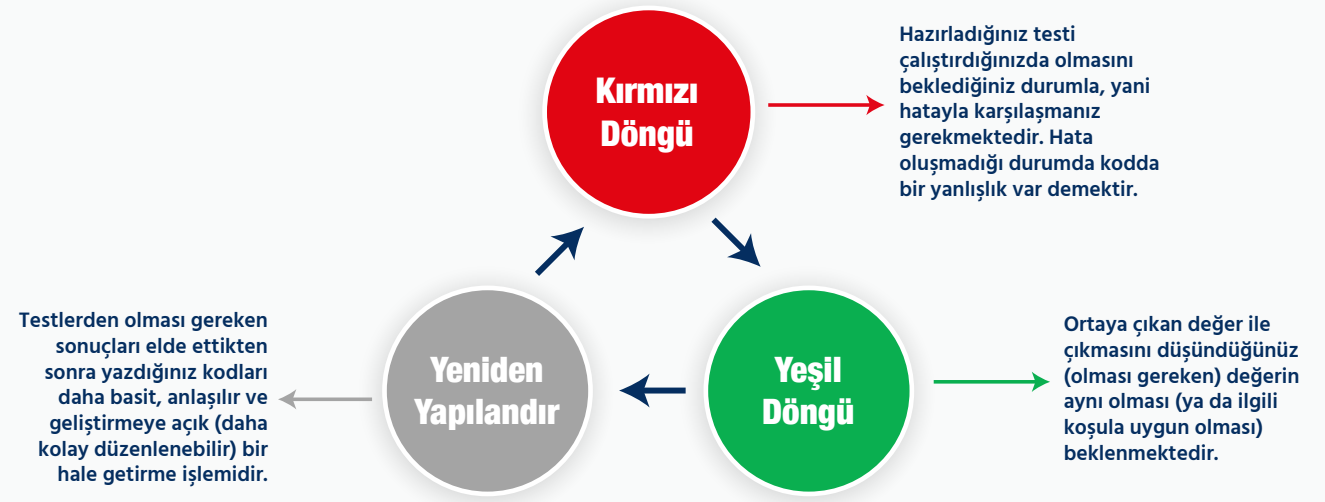
TDD geleneksel yazılım tarzını tamamen tersine çevirir ve yazılıma testler ile başlanır. Kent Beck'in tanımladığı gibi herhangi bir satır program kodu oluşturmadan önce bir test sınıfı oluşturularak yazılım işlemine başlanmaktadır. Bu süreci daha iyi anlayabilmek için kırmızı döngü (red cycle), yeşil döngü (green cycle) ve yeniden yapılandırma (refactoring) kavramları anlaşılmalıdır.

Kırmızı döngü (red cycle) yani hatalı sonuç döndüren test kavramı; bu aşamada yazılım geliştirici kendisine verilmiş gereksinimlerden sıradaki gereksinime ait birim testi yazar. Yazılan test, daha öncesinden yazılmış olan tüm birim testler ile çalıştırılır. En son yazılan birim teste ait uygulama (implementasyon) henüz gerçekleşmediği için ilgili test başarısız sonuçlanır.

Yeşil döngü (green cycle) yani başarılı sonuç döndüren test kavramı; yazılım geliştirici gereksinimi karşılayacak yazılımın uygulamasını en az kodla yapar. Yazılan kodun sadece şimdiye kadar birim testi yazılmış gereksinimleri karşılaması beklenir. Gelecekte yazılımı yapılacak durumların kodlaması önceden yazılmaz ve düşünülmez. Uygulaması tamamlandıktan sonra yazılımın gereksinimi karşılayıp karşılamadığını kontrol etmek için tüm birim testler çalıştırılır. Beklenen durum tüm birim testlerin başarıyla geçmesidir. Bu testler başarıyla çalıştığı takdirde yazılım geliştirici bir sonraki aşamaya geçebilir. Aksi takdirde gereksinim ile yazılım arasında uyumsuzluk olduğunu gösterir.

Yeniden yapılandırma (refactor) yani tasarımı iyileştir kavramı; yazılım geliştirici bu aşamada testlerinde değişiklik yapmadan kodunu yeniden gözden geçirerek tasarımını iyileştirmeye çalışır. Bu kapsamda yazılım geliştirici kullandığı metod, değişken ve sınıf isimlerini düzenleyebilir. Var olan kod tekrarlarını giderir ve kodunu daha test edilebilir hala getirir. Böylece kodun bağımlılığını azaltırken amaca yönelik metod ve sınıfların ortaya çıkmasını sağlar. Bu aşamada dikkat edilmesi

gereken nokta değişikliklerin adım adım yapılmasıdır. Ayrıca her bir adımda yazılımın derlenebiliyor olmasına özen gösterilmelidir. Bununla birlikte yazılım geliştirici her değişiklik adımında tüm testleri çalıştırarak testlerin başarıyla geçtiğinden emin olmalıdır. Testler geçmediği sürece değişiklikler yapılmamalıdır.



Şekil 6. TDD Yaşam Döngüsü

Kent Beck "Test Driven Development: By Example" isimli kitabında TDD için yapılması gereken adımları şu şekilde sıralar:

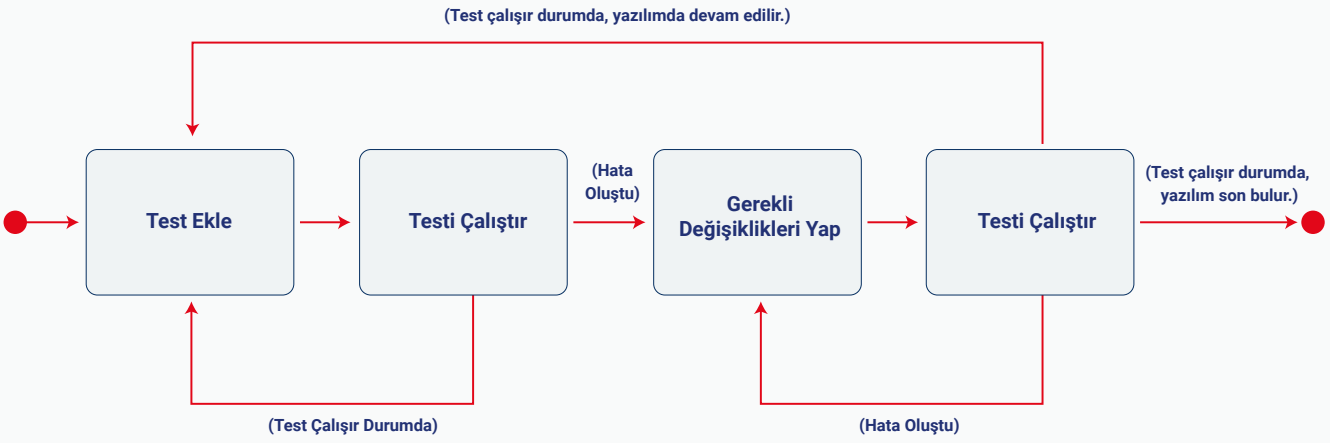
a) Hızlı bir şekilde test ekleyin.

b) Tüm testleri çalıştırın ve yenisinin başarısız olduğunu görün.

c) Biraz değişiklik yapın

d) Tüm testleri çalıştırın ve hepsinin başarılı olduğunu görün.

e) Tekrarı kaldırmak için yeniden yapılandırın.



Şekil 7. TDD İçin Yapılması Gereken Adımlar

"Test ekle – kodla - yeniden yapılandır" olarak tanımlayabileceğimiz TDD sürecinde, sadece test metodlarının öngördüğü sınıflar oluşturulur. Burada dikkat edilmesi gereken nokta, gerekli sınıfların en basit şekilde oluşturulmalarıdır. Yazılım geliştirici testi bırakıp, testin gerek duymadığı sınıfları oluşturmamalıdır. Test edilen sınıf metodları ilk etapta null değerini geri verecek ya da hiçbir şey yapmayacak şekilde programlanır. Test çalıştırıldığında hata verecektir, çünkü kullanılan metodlar hiçbir şey yapmamaktadır. Bu noktada test çalışacak şekilde kod üzerinde değişiklik yapılır. Yine metodların en basit şekilde uygulanmalarına dikkat edilir. Bu işlemler ve yeni testler program için gerekli tüm sınıflar oluşturulana kadar devam eder. Testler ve gerekli sınıflar yavaş yavaş oluştukça, test edilen sınıflar üzerinde gerekli değişiklikler yapılarak, istenilen tasarım oluşturulur. Bu yeniden yapılandırma işlemi (refactoring) mevcut testlerin desteğiyle çok daha kolay bir hal alır, çünkü yapılan her değişikliğin ardından testler yardımıyla yapılan değişiklerin yan etkileri kolayca tespit edilebilir. Buradan, yapılandırma işlemlerinin sağlıklı yapılabilmesi için mutlaka birim testlerin olması gerektiği sonucu çıkmaktadır. Yazılım son bulduğunda sistemde bulunan her sınıf ve metod için testler var olacaktır.

"Test ekle – kodla – yeniden yapılandır" adımları döngü içerisinde sürekli devam eder. Yazılım müşterinin tüm isteklerini karşılayabilecek seviyeye gelene kadar uygulama sürekli bir yapılandırma işlemine tabi tutulur.

Sonuç ve Öneriler

Bir yazılımı test etmek önemlidir. Çünkü test metodojileri yazılımın gerçekten olması gerektiği gibi çalışmasını sağlar. Bu yöntemlerin nasıl çalıştığını anlamak; geliştiricilerin ve yazılım geliştirmeye dahil olan diğer bireylerin, amaçlarına hizmet etmek için hangi yaklaşımın en iyi sonucu vereceğini anlamalarına yardımcı olabilirler. Yazılımda hataların analizi ve çözümleri şirketlere büyük miktarlarda zaman, para ve prestij kaybettirmektedir. Dolayısıyla test yazmanın ek bir yük olmadığının farkına varılmalı ve test yazma süreçlerine önem verilmelidir. Bu çalışmada yazılım testleri, test yöntemleri ve yaklaşımları incelenerek, bir yazılım test yaklaşımı olarak TDD'nin yazılım geliştirme sürecindeki önemi vurgulanmış, nasıl uygulanacağı açıklanmıştır.

Yazılım geliştirme sürecinde yazılım test yaklaşımı olarak TDD kullanıldığında,

- Doğru uygulandığı takdirde son satırına kadar test edilmiş bir yazılım sisteminin oluşmasını sağlar.
- İyi ve zaman içinde gerekli değişikliklere ayak uyduracak bir tasarım oluşturmak kolaylaşır.
- Yazılım geliştirici geliştirme sırasında, sistemin nasıl çalışması gerektiğini hayal ettiği için sadece gerekli sınıf ve metodları oluşturur.
- Yazılım geliştiricinin "belki ilerde kullanılır, bu metodu eklemekte fayda var" tarzı düşünmesini engeller. Böylece TDD proje maliyetini düşürür, çünkü sadece gerekli sınıf ve metodlar için zaman harcanır.
- Testler oluşturulurken, oluşturulan sınıfların kullanımı ve performansı hakkında da düşünülür. Bu sayede de yazılım tasarımında iyileştirme yapılır.
- Test kapsama alanı geniş olur, neredeyse her kod satırı test metodları tarafından çalıştırılır.
- Testler koda olan güveni artırır ve kodun yeniden yapılandırılmasıyla oluşabilecek hatalar testler tarafından yakalanabilir.
- Yazılım geliştirici daha test edilebilir kod yazmaya çalışır. Bir kodun test edilebilirliği ne kadar fazla ise kodun içerisindeki sınıfların birbirleriyle olan bağlılığı o kadar azalır. Böylece yazılım geliştirici, nesne yönelimli programlama hedeflerinden biri olan yüksek kohezyon (high cohesion) ve gevşek bağımlılık (loose coupling) durumunu sağlamış olur.

Görüldüğü üzere yazılım geliştirme sürecinde yazılım test yaklaşımı olarak TDD'nin kullanılmasının bir çok avantajı vardır. Her yazılım projesinin gereksinimleri farklıdır ve TDD'nin uygulanabilirliği projenin özelliklerine ve ihtiyaçlarına bağlı olarak değişebilir. Pojenin büyüklüğü, geliştiren ekibin yetkinliği, proje yönetim metodu gibi konular düşünülerek TDD'nin kullanılıp kullanılmayacağına karar verilebilir. Projeyi geliştiren ekip TDD'yi benimsemeye istekliyse ve bu yaklaşımı kullanmaya motive ise projenin TDD ile yürütülmesi daha kolay olacaktır.

Kaynakça

1. https://dergi.bmo.org.tr/teknoloji/test-gudumlu-programlama
2. https://talentgrid.io/tr/test-driven-development-nedir/
3. https://medium.com/devopsturkiye/yaz%C4%B1l%C4%B1m-geli%C5%9Ftirirken-neden- test-yazmal%C4%B1y%C4%B1m-b58da8124f5c
4. https://hasandogn.medium.com/test-driven-development-tdd-nedir-b78c7a6ef10b
5. https://acokgungordu.medium.com/yaz%C4%B1l%C4%B1m-geli%C5%9Ftirme-s%C3%BCrecinde-test-tdd-atdd-ve-bdd-aras%C4%B1ndaki-farklar-8332c20aaec7
6. https://www.guru99.com/test-driven-development.html
7. https://medium.com/@nerobianchi/tdd-nedir-965c1a26e68f
8. http://www.kurumsaljava.com/2008/11/26/test-gudumlu-yazilim-test-driven-development- tdd/
9. https://keytorc.com/blog/entegrasyon-testi-nedir/
10. https://www.zaptest.com/tr/entegrasyon-testi-nedir-turler-surec-ve-uygulamaya-derinlemesine-bakis
11. https://www.zaptest.com/tr/sistem-testi-nedir-yaklasimlar-turler-araclar-ipuclari-ve-puf-noktalari-ve-daha-fazlasina-derinlemesine-bir-bakis
12. https://muzafferkaleli.com/2019/11/24/agile-software-testing/
13. https://tr.linkedin.com/pulse/kullan%C4%B1c%C4%B1-kabul-testleri-nedir-digital-vizyon-akademi
14. Beck K. (2002) Test-Driven Development By Example, Addison-Wesley. ISBN: 978-0321146533
15. Acar Ö. (2013), Pratik Agile: Extreme Programming
16. Feathers, M. (2004). Working Effectively with Legacy Code. Prentice Hall. ISBN: 9780131177055
17. Yamuç A ,Çakın A. Web Uygulamaları Geliştirilmesinde Test Güdümlü Programlamanın Yeri: Bir Örnek Durum Çalışması, İstanbul
18. Nalbant E. ,Yazılım Yaşam Döngünde Testin Önemi ve Bir Test Otomasyon Projesinin Gerçekleştirilmesi, İstanbul

T.C. SANAYİ VE
TEKNOLOJİ BAKANLIĞI

#MİLLİ TEKNOLOJİ HAMLESİ

TÜBİTAK

İşçi Blokları Mahallesi Muhsin Yazıcıoğlu Caddesi No:51/C 06530 Çankaya/ANKARA

+90 (312) 289 92 22 - yte.bilgem@tubitak.gov.tr

TÜBİTAK - BİLGEM Yazılım Teknolojileri Araştırma Enstitüsü (YTE)