T.C.

TRAKYA ÜNİVERSİTESİ

FEN BİLİMLERİ ENSTİTÜSÜ

İlişkisel Veritabanı Kullanılan Yazılımlarda Black-Box ve White-Box Test Yöntemleri ile Agile Metodolojiye Uygun bir Hibrit Test Metodu ve Uygulama Yazılımının Geliştirilmesi

Emin BORANDAĞ

Doktora Tezi

Bilgisayar Mühendisliği Anabilim Dalı

Danışman: Prof. Dr. Şaban EREN

2011

EDİRNE

T.C.

TRAKYA ÜNİVERSİTESİ

FEN BİLİMLERİ ENSTİTÜSÜ

İLİŞKİSEL VERİ TABANI KULLANILAN YAZILIMLARDA BLACK BOX VE WHITE BOX TEST YÖNTEMLERİ İLE AGILE METODOLOJİYE UYGUN BİR HİBRİT TEST METODU VE UYGULAMA YAZILIMININ GELİŞTİRİLMESİ.

Emin BORANDAĞ

DOKTORA TEZİ

BİLGİSAYAR MÜHENDİSLİĞİ ANA BİLİM DALI

Bu tez 01 / 07 / 2011 tarihinde aşağıdaki jüri tarafından kabul edilmiştir.

Prof. Dr. Şaban EREN
(Danışman)

Yrd.Doç. Dr. Erdem UÇAR
(Üye)

Doç. Dr. Yılmaz KILIÇASLAN
(Üye)

Yrd. Doç. Dr. Aydın CARUS
(Üye)

Yrd. Doç. Dr. Yılmaz GÖKŞEN
(Üye)

Doktora Tezi

Trakya Üniversitesi

Fen Bilimleri Enstitüsü

Bilgisayar Mühendisliği Anabilim Dalı

ÖZET

İlişkisel Veritabanı Kullanılan Yazılımlarda Black-Box ve White-Box Test Yöntemleri ile Agile Metodolojiye Uygun bir Hibrit Test Metodu ve Uygulama Yazılımının Geliştirilmesi

Günümüzde yazılım sistemleri, birçok iş alanının temel bileşenleri içersinde yer almaktadır. Büyüyen rekabet, gelişen teknoloji ve yazılım kuruluşlarının artan kabiliyetlerinin de etkisiyle gelişmiş yazılım sistemlerine her geçen gün daha çok ihtiyaç duyulmaktadır. Son yirmi yılda, yazılım geliştirmede kullanılan kalite sistemlerini ve süreçlerini değerlendirmek, yazılımda kalite sertifikasyonu sağlamak, süreçleri iyileştirmek ve yetenek belirlemek için çeşitli modeller geliştirilmiştir. Her ne kadar farklı yetenek ve özelliğe sahip modeller geliştirildiyse de yazılımda hata konusu hep önemli bir sorun olarak gündemde kalmıştır.

Bu çalışma, yazılım süreçlerini test süreci yönetimi başlığı altında ele alıp, agile metodolojiye uygun, ilişkisel veri tabanlarını kullanan yazılımlara uygulanabilecek, hibrit bir test metodu geliştirilmesini ve bu metodun geliştirilen yazılımlar üzerine uygulanmasını kapsamaktadır. Geliştirilen model ve uygulama, başta küçük ölçekli yazılım geliştirme firmaları olmak üzere, değişikliklerin fazla olduğu projelerde de kullanılabilir niteliktedir.

Bu tez 2011 yılında yapılmıştır ve 93 sayfadan oluşmaktadır.

Anahtar Kelimeler: Yazılım Testi, Yazılım Geliştirme Metotları, Agile Yazılım Geliştirme Yöntemi, Extreme Programlama.

Doctorate Thesis

Trakya University

Graduate School of Natural and Applied Sciences

Department of Computer Engineering

ABSTRACT

Development of a Hybrid Test Method Complied with the Agile Methodology and an Application Software by Using Black-Box and White-Box Test Methods for Software Using Relational Database

Nowadays, software systems are essential to many lines of business. The need for advanced software systems is growing with every passing day as a result of the effects of increasing competition, improving technology and the rising capabilities of software organizations. Various models have been developed in the last 20 years to evaluate the quality systems and processes that are used in software development, to refine the processes and to determine capabilities. Software errors remain an important issue, although models having different capabilities and features have been developed.

This thesis studies the development of a hybrid test method in accordance with agile methodology, which is then applied to software projects using a relational database. The thesis also covers the application of the method to software projects developed, considering the software processes within the context of test process management. The developed model is applicable to projects which undergo many changes, especially projects developed in small-scale software companies.

This thesis has been completed in 2011 and consists of 93 pages.

Keywords: Software Testing, Software Development Methodologies, Agile Software Development Methodology, Extreme Programming.

ÖNSÖZ

Başta tez çalışmam sırasında ilgi ve yardımlarını esirgemeyerek, evladına yardım eden bir baba gibi davranan tez danışmanım Yaşar Üniversitesi Fen-Edebiyat Fakültesi Dekanı Sn. Prof. Dr. Şaban EREN'e, tez izleme komitesinde yer alan değerli hocalarım Sn. Doç. Dr. Yılmaz KILIÇASLAN'a, Sn. Yrd. Doç.Dr. Erdem UÇAR'a, Sn. Yrd. Doç. Dr. Aydın CARUS'a ve Sn. Yrd. Doç. Dr. Yılmaz GÖKŞEN'e;

Her zaman sevgi ve desteklerini esirgemeden bana güç ve destek veren sevgili anne ve babama;

Bu çalışmanın gerçekleştirilmesi sırasında katkı veren çalışma arkadaşlarım Zeynep ÖNYURT'a, Fatih YÜCALAR'a, Önder ŞAHİNASLAN'a ve isimlerini ayrı ayrı sayamadığım, çalışmanın oluşması ve gelişimi sırasında değerli katkılarda bulunan, desteklerini gördüğüm değerli hocalarıma ve çalışma arkadaşlarıma sonsuz teşekkürlerimi sunarım.

İÇİNDEKİLER

ÖZET.. i

ABSTRACT ii

ÖNSÖZ iiii

İÇİNDEKİLER iv

SİMGELER DİZİNİ……vi

ŞEKİLLER DİZİNİ vii

ÇİZELGELER DİZİNİ viii

GİRİŞ……1

1. BÖLÜM3

YAZILIM SÜREÇ, KALİTE VE METODOLOJİLERİ 3

1.1. Yazılım 3

1.2. Yazılımların Tiplerine Göre Sınıflanması 3

1.3. Yazılım Mühendisliğinin Tarihsel Gelişimi 4

1.4. Yazılımın Gelişim Dönemleri 5

1.5. Yazılım Hatalari 6

1.6. Yazılım Proje Yönetimi 7

1.7. Yazılım Karmaşıklığı 8

1.8. Yazılım Mühendisliği Eğitiminin Bireye Kazandırdıkları 8

1.9. Yazılım Projelerinde Başarı…… 9

1.10.Yazılım Geliştirme Metodolojileri 10

1.11. Yazılımda Kalite 14

1.12. Süreç 17

1.13.Yazılım Süreç İyileştirme Modelleri 19

2. BÖLÜM23

YAZILIMDA TEST YÖNTEMLERİ……23

2.1. Birim Testi 24
2.2. Bütünleme Testi 25
2.3. Onaylama Testi 25
2.4. Metot Testi 26
2.5. White Box Test……..…26
2.6. Black Box Test.....33
3. BÖLÜM.....34
AGILE PROGRAMLAMA 34
3.1. Extreme Programlama –XP 35
3.2. XP'nin Bazı Avantajları ve XP Manifestosu 36
3.3. XP'nin Çalışma Mantığı.....38
3.4. XP Uygulamaları 40
3.5. XP ile Proje Başlangıcı.....43
3.6. XP ile Testlere Öncelik Vermek 44
3.7. XP ile İlgili Yapısal Bilgiler.....47
4. BÖLÜM.....50
4.1. Test Öncelikli Hibrit Yazılım Geliştirme Methodu.....50
4.2. Hibrit Test Aracı.....52
4.3 Geliştirilen Yazılım Projeleri 58
5. BÖLÜM.....76
SONUÇ VE DEĞERLENDİRME 76
KAYNAKLAR.....78
ÖZGEÇMİŞ.....83

SİMGELER DİZİNİ

Kısaltma		Türkçe Karşılığı	İngilizce Karşılığı
AQAP 150/160	:	Müttefik Kalite Güvence Yayınları 150/160	Allied Quality Assurance Publications 150/160
CMM	:	Yetenek Olgunluk Modeli	Capability Maturity Model
CMMI	:	Tümleşik Yetenek Olgunluk Modeli	Capability Maturity Model Integration
EVO	:	Yazılım Gelişimi	Software Evolution
IBM	:	International Business Machines	International Business Machines
IEEE	:	Elektrik ve Elektronik Mühendisleri Enstitüsü	The Institute of Electrical and Electronics Engineers
ISO	:	Uluslararası Standart Teşkilatı	The International Organization for Standardization
ISO 12207	:	Uluslararası Standart Teşkilatı 12207	International Standardization Organization 12207
ISO 15504 (SPICE)	:	Uluslararası Standart Teşkilatı 15504 (Yazılım Süreç İyileştirme ve Yetenek Belirleme)	International Standardization Organization 15504 (Software Process Improvement and Capability dEtermination)
MSF	:	Microsoft Çözüm Çerçevesi	Microsoft Solution Framework
RAD	:	Hızlı Uygulama Geliştirme	Rapid Application Development
RUP	:	Rasyonel Birleştirilmiş Süreç	Rational Unified Process
SEPG	:	Sistem Mühendisliği Süreç Grubu	System Engineer Process Group
XP	:	Extreme Programlama	Extreme Programming

ŞEKİLLER DİZİNİ

Şekil 1-1Çağlayan Modeli.....12

Şekil 1-2 Bohem Spiral Modeli.....13

Şekil 1-3 Süreç meta modeli 18

Şekil 1-4 CMMI'ın Yapısı 21

Şekil 2-1 Kara Kutu Testi.....33

Şekil 3-1Değişim Maliyeti ile XP in İlişkisi 37

Şekil 3-2 XP Çalışma Mantığı 39

Şekil 3-3 Proje Çevrimi 44

Şekil 4-1 Test Öncellikli Yazılım Akış Şeması.....52

Şekil 4-2 Uygulama Yazılımı Giriş Ekran Görüntüsü…….……...54

Şekil 4-3 Uygulama Yazılımı Projeler Yönetim Modülü Ekran Görüntüsü……..55

Şekil 4-4 Uygulama Yazılımı Müşteri Kartları Yönetim Modülü Ekran Görüntüsü..…56

Şekil 4-5 Uygulama Yazılımı Teknik Kartları Yönetim Modülü Ekran Görüntüsü..…57

Şekil 4-6 Uygulama Yazılımı Raporlama Modülü Ekran Görüntüsü……..……...58

Şekil 4-7 Personel Otomasyon Sistemi Ekran Görüntüsü……..…….62

Şekil 4-8 Optik Okuyucu Değerlendirme Sistemi Ekran Görüntüsü……..……67

Şekil 4-9 Muhasebe Otomasyon Sistemi Ekran Görüntüsü……..……..71

ÇİZELGELER DİZİNİ

Çizelge 1-1 Yazılımda Kalitenin Tarihsel Gelişimi 5

Çizelge 1-2 Yazılım Hatalarının Dağlımı.....6

Çizelge 1-3 Yazılım Hata Düzeltme Maliyetleri.....7

Çizelge 1-4 Proje Başarı Oranları.....9

Çizelge 2-1 Doğruluk Çizelgesi 30

Çizelge 2-2 Doğruluk Çizelgesi 32

Çizelge 3-1Extreme Programlama Tarihsel Gelişimi 36

Çizelge 3-2 XP Programlama pratikleri 38

Çizelge 4-1 Yazılım Geliştirme Araçları.....53

Çizelge 4-2 Personel Programı Bilgilendirme Tablosu.....63

Çizelge 4-3 Personel Otomasyon Sistemi-Ölçümler.....64

Çizelge 4-4 Optik Okuyucu Yazılım Projesi Bilgilendirme Tablosu.....67

Çizelge 4-5 Optik Okuyucu Sistemi Ölçümler.....68

Çizelge 4-6 Muhasebe Otomasyon Programı Bilgilendirme Tablosu.....72

Çizelge 4-7 Muhasebe Otomasyon Programı Ölçümler.....73

GİRİŞ

Yazılım teknolojilerinde ve yazılım geliştirme modellerinde yaşanan hızlı gelişme sonucunda, küçük, orta ve büyük ölçekli yazılım geliştirme firmaları tarafından çok farklı alanlarda kullanılabilen yazılımlar geliştirilmiştir. Bu yazılımlarda, kalite sertifikasyonu sağlamak, süreçleri iyileştirmek ve yetenek belirlemek için çeşitli yazılım geliştirme yöntemleri kullanılmaktadır. Son yıllarda geliştirilen yazılımlarda, müşteri gereksinimlerinin artmasına paralel olarak yazılım üretimindeki karmaşıklık da artmıştır. Karşılaşılan sorunlar neticesinde, yazılımların büyük bir kısmının etkin bir şekilde kullanılmadığı görülmektedir.

Bu sorunların aşılması yönünde yapılan çalışmalar sonucunda agile olarak adlandırılan ve yazılım yaşam döngüsünü baştan aşağı değiştiren yöntemler geliştirilmiştir. Bu yöntemler arasında en yaygın olarak kullanılanı, Extreme Programming (XP[1])'dir. Özellikle son yıllarda kullanılan bu yöntem ile birlikte yazılım geliştirme işi kolaylaştırılmaya çalışılmaktadır.

2000 ile 2005 yıllarını kapsayan dönemde yapılmış bir çalışmada, yazılım testlerindeki kalitenin düşük olması nedeniyle Amerika'da 59,5 milyar dolar'ın boşa gittiği görülmüştür. Asıl düşündürücü olan kısım ise bu miktarın 22,5 milyar dolar'lık kısmının bazı basit yazılım test teknikleri ile önceden görülerek kurtarılabilecek olmasıdır (Schach, 2007).Bu kapsamda yazılımın test edilmesi; yazılımda var olan durum ile projenin isterlerini karşılaştırarak ikisi arasındaki farklılıkları bulmaktır. Bu amaçla yazılım özellikleri değerlendirilir ve incelenir. Bu işlemlerin yapılması için test aktivitelerinin belirlenip ona göre bir plan oluşturulması gerekir. Testlerin belirlenip uygulanması ile yazılımın kabulüne kadar geçen süre içinde takip edilen tüm yöntem ve yaklaşımlar test süreci yönetimi başlığı altında incelenmelidir.

---

[1]Extreme Programming'in kısaltması olarak XP kullanılmaktadır.

Bu çalışmada, küçük ve orta ölçekli yazılım organizasyonları tarafından çoğunlukla göz ardı edilen yazılım testinin, yazılım yaşam döngüsünün tüm safhalarında nasıl uygulanacağı ele alınmıştır. Bu kapsamda tezin amacı, yazılımın doğasında olan hata unsurunu azaltmayı sağlayabilecek; XP ile geliştirilen yazılım projelerinde kullanılabilecek bir hibrit test metodu geliştirmektir. Yöntemin kullanımı ile ilgili olarak bir yazılım aracı da geliştirilmiştir.

Çalışmalara öncelikle tez konusu ile doğrudan veya dolaylı olarak ilgisi bulunabilecek ulusal ve uluslararası kaynakların taranmasıyla başlanılmış ve bir referans eserler kütüphanesi oluşturulmuştur. Toplanan tüm eserler belirli kategoriler altında uygun formatta isimler verilerek konumlandırılmıştır. Sonraki aşamada her bir eserin ön incelemesi yapılarak hangi eserden ne ölçüde yararlanılıp yararlanılmayacağına yönelik bir karşılaştırma yapılarak önemli eserler belirlenmiş ve detaylı bir şekilde incelenmiştir. Yazılım süreç, kalite ve metodolojilerine ilişkin elde edilen bilgileri de içeren temel tanım ve kavramlarla ilgili açıklamalar çalışmanın ikinci bölümünde detaylı olarak verilmiştir. Ayrıca ikinci bölümde, yazılım geliştirmede etkin olan kriterler, standartlar, yaygın kullanılan metodolojilere ilişkin bilgiler, yazılım test teknikleri ve test yöntemleri ile ilgili bilgilendirmede bulunulmaktadır.

Çalışmanın üçüncü bölümünde Agile Programlama başlığı altında XP yazılım geliştirme yöntemi ele alınmış ve bu yönteme ilişkin kapsamlı bilgilendirme yapılmıştır. Dördüncü bölümde önerilen hibrit test metodu ve uygulama yazılımının geliştirilmesi anlatılmıştır. Çalışmanın son bölümünde ise elde edilen sonuçlara ve önerilere yer verilmiştir.

1. BÖLÜM

YAZILIM SÜREÇ, KALİTE VE METODOLOJİLERİ

Yazılım mühendisliği, müşterinin ihtiyaçlarını karşılayan, öngörülmüş bütçe ile zamanında teslim edilen ve hatasız bir yazılım geliştirmek için teknik yöntemler öngören bir mühendislik disiplinidir (Schach, 2007). Bu bakımdan yazılım mühendisliği bir yöntemler, teknikler ve araçlar kümesi olarak değerlendirilebilir. Yazılım üretiminde, yazılım yaşam döngüsü içerisinde yer alan safhaların yazılım mühendisliği ilkelerine göre gerçekleştirilmesi çok önemlidir (Sallie, 1993). Bu bakımdan yazılım mühendisliği, yazılım üretimindeki karmaşıklıkları gidermeyi hedefler.

Geçmişte kullanılan iş-akış şemaları gibi yöntemler, günümüzde, yazılım projelerinin boyutlarının büyümesinden dolayı yetersiz kalmış ve yerlerini daha gelişmiş metotlara bırakmıştır. Bundan dolayı yazılım üretim işi, tek kişinin başarabileceği boyuttan çıkmış ve artık takım çalışması biçimine dönüşmüştür. Yazılım mühendisliği bu temel olgular üzerine önem kazanmıştır. Bu bölümde yazılım ve yazılım mühendisliği konuları hakkında bilgiler verilmektedir.

1.1. Yazılım

Bir bilgisayar sisteminin istenilen işlemleri yapabilmesi için gerekli olan komutlar topluluğuna yazılım denir. En basit tanımıyla, bir sistemin donanım bileşenleri dışında kalan her şey yazılım olarak adlandırılır (Schach, 2007). En temel yazılım bileşenleri; yöntem, yazılımcı, mantık, bilgi ve isterlerdir (Alkan, 2004).

1.2. Yazılımların Tipine Göre Sınıflandırılması

Tek bir yazılım çeşidi ya da tek bir yazılım metodunu öğrenerek tüm yazılım çeşitleriyle ilgili bilgileri aktarmak mümkün değildir. Veritabanı ile ilgili yazılımlar olsun, grafiğin ön planda olduğu animasyon ile ilgili yazılımlar olsun hepsinin farklı bir

yapısı ve işleyişi vardır. Aşağıda farklı yazılım çeşitlerinden örnekler verilmiştir (Kuitunap, 2009).

- Animasyon yazılımları,
- Mesleki ve ticari yazlımlar,
- Sistem yazılımları,
- İşletim sistemleri,
- Derleyiciler,
- Editörler,
- Debug programları,
- Yapay zeka yazılımları.

1.3. Yazılım Mühendisliğinin Tarihsel Gelişimi

Yazılım mühendisliğinin tarihi 1946 yılında ENIAC bilgisayarı ile başlamıştır. 1953 yılına gelindiğinde yazılım işinin merkezi İngiltere'den ABD'ye taşınmış ve çeşitli programlama dilleri üretilmiştir. Bu dillerden birisi de 1953 yılında üretilen FORTRAN programlama dilidir. Bu süre içerisinde programlama dillerini kullanışlı hale getirilmiş ve farklı programlama dilleri geliştirilmiştir. Ancak geliştirilen bu yazılımları üretmek için kullanılan metotlara ilk başlarda yeteri kadar önem verilmemiştir. 1962 yılında NASA tarafından uzaya gönderilen bir uzay aracının infilak etmesine yazılımlardan kaynaklanan bir hatanın neden olduğu saptanmış ve bu nedenle yazılım hataları üzerinde daha önemle ve özenle durulmaya başlanılmıştır. 1968 yılına gelindiğinde Almanya'nın Garmisch kasabasında düzenlenen bir NATO Konferansı'nda "yazılım mühendisliği" kavramı ilk kez kullanılmıştır. Bu kavramın tanımlanması ile birlikte yazılım işi, bir disiplin içerisinde değerlendirilmeye başlanmıştır. (Randell B, 1968) 1972 yılında IEEE tarafından yazılım mühendisliği kavramı kullanılmış, 1976 yılına gelindiğinde ise yazılım geliştirme standartları tanımlanmıştır. 1990'lı yıllardan itibaren bilgisayar mühendisliği disiplini içerisinde yer alan yazılım mühendisliği, daha sonraki yıllarda diğer mühendislik dallarından bağımsız bir hale gelmiştir. Yazılım mühendisliği, teknolojinin gelişimine büyük bir ivme sağlamıştır. Yaşamımız boyunca kullandığımız pek çok araç gerecin içerisinde yazılım teknolojisini görmek mümkün olmaktadır. Aynı şekilde yazılım mühendisliği, değişik sektörlerdeki (bankacılık, sağlık,

telekomünikasyon, hava sanayi, eğitim, finans, vb) işlerin daha kolay bir şekilde yapılmaları için yardımcı olmaktadır (Grad, 2010).

Çizelge 1-1'te yazılımda kalitenin tarihsel gelişimi yıllara göre verilmiştir. (Kalaycı, 2005)

Çizelge 1-1 Yazılımda Kalitenin Tarihsel Gelişimi(Kalaycı, 2005)

Yıl	Yazılımda Tarihsel Gelişim
1924	İlk modern istatiksel kalite kontrol –Shewhart
1940	Modern istatiksel kalite kontrol kullanılmaya başlıyor
1946	İlk Bilgisayar(ENIAC)
1950	Juran,Deming ve Feigenbaum'un önderliğinde Japonya
1960	Kalite kontrol ve yönetimi felsefesi.
1968	NATO Yazılım Mühendisliği Konferansı
1969	İlk Uluslararası Kalite Konferansı –Japonya Tokyo
1970	Yazılım Kalitesi terimi ilk kez kullanılıyor.
1980	Avrupa ve Amerikalı firmalarda toplam kalite çalışmaları başlangıcı

1.4. Yazılımın Gelişim Dönemleri

Yazılımın gelişimindeki tarihsel aşamaları dört grupta toplanabilir:

- **Birinci Dönem:** İlk bilgisayarların üretilmeye başladığı bu yıllarda bilgisayarların maliyetlerinin yüksekliğinden dolayı kişisel bilgisayar kullanımı mümkün değildi (Grad, 2010).
- **İkinci Dönem:** Veritabanı kullanımının da başladığı bu dönemde, bilgisayarların maliyet sorunu da belli oranda azalmıştır.
- **Üçüncü Dönem:** Bilgisayar ağ teknolojilerinin geliştiği bu dönemde, kişisel bilgisayarlar da kullanılmaya başlanmıştır.

- **Dördüncü Dönem:** Şu an içinde bulunduğumuz yılları da kapsayan bu dönemde bir işi yaptırmak için birden fazla bilgisayarın kullanılabildiği teknolojiler ortaya çıkmış, yazılım üretiminde karşılaşılan zorluklar nedeniyle yazılımda kalite anlayışı önem kazanmaya başlamıştır. Bu amaçla çeşitli kuruluşlar kendi yazılım standartlarını geliştirmişlerdir (Aspray, 1996).

1.5. Yazılım Hataları

Yazılım geliştirme sürecinde oluşan hatalar yazılımın geliştirme süresini uzatmanın yanında, bulundukları yere göre maliyetleri değişmektedir. Örneğin, ürün teslimi sırasında ortaya çıkan bir hatanın maliyeti, analiz sırasında ortaya çıkan bir hataya göre çok daha fazladır. Geliştirilen yazılımlardaki tüm hataların bulunması oldukça zordur. Bu nedenle yazılımlardaki tüm hataların bulunup, daha sonra sistemin işletime alınması gibi bir durum söz konusu değildir. Yapılan testler ile hatalar en aza indirilmeye çalışılır. Çizelge 1-2'de oluşan hataların dağılımları verilmiştir (Gerald, 2007).

Çizelge 1-2 Yazılım Hatalarının Dağılımı(Gerald, 2007)

Hata Türleri	**Hata Yüzdesi (%)**
Analiz Sırasında Oluşan Hatalar	%55
İşlevsel Tasarım Sırasında Oluşan Hatalar	%25
Kodlama	%15
Belgeleme	%5

Yazılım geliştirme sürecinin ileriki safhalarında ortaya çıkan hataların düzeltilme maliyeti, erken safhalarda ortaya çıkan hataların düzeltilme maliyetinden daha fazladır. Yazılım geliştirme safhalarına göre oluşan hataların maliyetleri Çizelge 1-3'de verilmiştir (Gerald, 2007).

Çizelge 1-3 Yazılım Hata Düzeltme Maliyetleri. (Gerald, 2007)

Yazılım Geliştirme Safhaları	Hata Düzeltme Maliyeti ($)
Çözümleme	1$
Tasarım	5$
Kodlama	10$
Test	25$
Kabul	50$
İşletim	100$

1.6. Yazılım Proje Yönetimi

Yazılım ve yazılım mühendisliği kadar önemli olan bir diğer konu da projenin iyi bir şekilde yönetilmesidir. Kendi yapısını koruyan sağlam bir yazılımın proje kısmında insan ve süreç konularına odaklanılması gerekmektedir. Yazılım proje yönetiminin ilgilenmesi gereken birinci konu insandır. Yazılımın üretilmesi aşamasında çok yoğun olarak kullanılan insan faktörü, proje yöneticileri tarafından ihmal edilmemesi gereken bir konudur (Cockburn,2004). Yazılımı üretecek kişiler, müşteriler ve proje yöneticilerinin; sürekli bir araya gelerek, proje kapsamı ve o an gelinen durumu konuşmaları gerekmektedir. Yönetici projenin erken safhasında yeteri kadar müşteri ile bir araya gelmez ise ileriki safhalarda çözüm üretememe riski ile yüzleşmek zorunda kalabilir. Sorunlar işin ikinci kısmıdır. Erken sorun tespiti yazılım yöneticisinin hatayı daha kısa sürede çözmesini sağlayacaktır. Süreç kısmında ise süreçler belgelenmeli, süreçlerin girdileri ve süreç sonucunda çıkacak bilgiler irdelenmelidir. Proje yöneticisi işlerini yapabilmek için uygulamadan gelen deneyime, sorunlar için kullanılacak teknik araçlara ve yöntemlere sahip olmalıdır. Bu görevler yeteri kadar iyi bir şekilde yapılmaz ise sorunlara yanlış çözümler üretilmesi gibi bir durum ile karşılaşabilir (Andres, 2004).

1.7. Yazılım Karmaşıklığı

Bilgisayar sistemlerinin yapılarının zaman içerisinde gelişmesi sistemlerin karmaşık bir yapıya bürünmelerini sağlamıştır. Taleplerin artması ile birlikte daha işe özel ve daha karmaşık yazılımlar çıkmıştır. Ortaya çıkan gereksinimleri karşılamak giderek güçleşmiş ve işin içinden çıkılması imkânsız bir hal almıştır. Bu durumun üstesinden gelmek için yazılım karmaşıklığı konusu ele alınmıştır.

Yazılım karmaşıklığı, iki farklı boyutta ele alınır. Bunlardan birincisi yönetimsel karmaşıklık, ikincisi ise teknik karmaşıklıktır. Üretilecek yazılımın türüne göre kabaca bir karmaşıklık hesabı yapılabilir. Yazılım karmaşıklığının en önemli nedeni kod miktarındaki artış olup kod miktarındaki bu artış yazılımda hata oluşma riskini arttırır. Zaman içerisinde artan isterlerle birlikte daha da kendini gösteren kod artışı, yazılımda ortaya çıkması olası hataların sayısını arttırmaktadır. Yazılım hatalarını mümkün olduğu kadar azaltmak için bütünleşme, test ve bakım kısımlarına özen gösterilmesi gerekmektedir (Sommerville,2006).

1.8. Yazılım Mühendisliği Eğitiminin Bireye Kazandırdıkları

Yazılım mühendisliği eğitiminin doğru kurumlardan doğru şartlarda alınması halinde şu sonuçlara ulaşılacaktır:

- Yazılımı kullanacak kişilerden gelen isterleri analiz ederek, bu isterlere uyan çözümler tasarlayabilmek,
- Yazılımı kullanacak kişilerin belirlediği proje bitirme tarihi, proje bütçesi ve yazılım kullanılabilirliği konularında bir işbirliği sağlayabilmek
- Mühendislik etiğine uygun yasalar çerçevesinde belirlenmiş kurallara dikkat etmek ve aynı zamanda müşterinin beklentilerine ekonomik bir çözüm üretebilmek
- Yazılımın yaşam döngüsü içerisindeki bütün aşamalarını gerçekleştirmek (Sarıdoğan, 2004).

1.9. Yazılım Projelerinde Başarı

Yazılım projelerinde başarı ilkeleri temel olarak, müşteri gereksinimlerini yerine getirme, projenin zamanında ve bütçesi içerisinde bitirme şeklinde tanımlanmıştır. Yazılım mühendisliğinin ilk çıktığı 1960'lı yıllardan yazılımda kalite sorusunun tartışılmaya başlandığı 1970'li yıllara kadar üretilen projelerin başarı oranları Çizelge 1-4'de verilmiştir.

Çizelge 1-4 Proje Başarı Oranları (Shach, 2007)

Tamamen Başarılan Proje	Kısmen Başarılan Proje	Başarısız Olan Proje
%5	%40	%55

Çizelge1-4'de gösterildiği üzere firmaların çoğu belli metot ve metodolojileri kullanmadığı veya tam olarak bu yöntemlere başvurmadığından proje geliştirmeye çalıştıklarında büyük sorunlarla karşılaşmışlardır. Bunun sonucu olarak da ilk kez 1968'de söz edilen "Yazılım Krizi" sözcüğü çok sık kullanılır olmuştur. Bu sorunun farkına varılmış ve 1984 yılında Carnegie Mellon Üniversitesi bünyesinde Yazılım Mühendisliği Enstitüsü (Software Engineering Institute) tarafından oluşturulan bir çalışma ekibi ile o güne kadar oluşturulmuş projeler incelenmiştir. Bu ekip, başarılı olan projelerin çoğunda yer alan ortak değerleri bir araya getirerek "En İyi Pratikler (Best Practices)" adı altında bir rapor hazırlamışlardır. Yazılım projeleri için gerekli olan "En İyi Pratikler" aşağıda listelenmiştir (Mead,Tobin,Couturiaux,1996).

- Erken tanı, erken çözüm maliyeti düşürür.
- Ürün değil, süreç önemlidir. Süreç iyileştirilmelidir.
- Sürekli iyileşme, gelişme yolu açık olmalıdır.
- Standartlar ve ölçütler kullanılmalı. (ölç, referanslı çalış)
- Müşteri tatmin araştırması yap.
- Proje kestirim araçları kullan.
- Süreçlerini iyi belgele.
- Metodoloji kullan ve iyi belgele.
- Kritik tasarım değerlendirmesi yap.

- Kod denetlemesi (walkthrough, inspection) yap.
- Konfigürasyon yönetimi aracı kullan.
- Hataları ve güvenilirliği ölç, kaydet, analiz yap,
- Tahmin modelleri kullan.
- Temel nedenlere (root cause) in.
- Proje sonrası değerlendirme yap.

1.10.Yazılım Geliştirme Metodolojileri

Yazılım geliştirmek için bazı metot ve metodolojiler kullanmamız gerekmektedir. Metot ve metodoloji kullanmak En İyi Pratiklerden biridir (Jones, 2000). Metot ve metodolojiler yapılan işleri belli bir proje yapısı içersinde süreçlere ayırarak neyin ne şekilde yapılması gerektiğini bize gösterir, aynı zamanda yazılımı düzgün bir biçimde gerçekleştirmemizi sağlar (Kemerer, 1994). Aşağıda, kullanılan bazı yazılım geliştirme metodolojileri yer almaktadır.

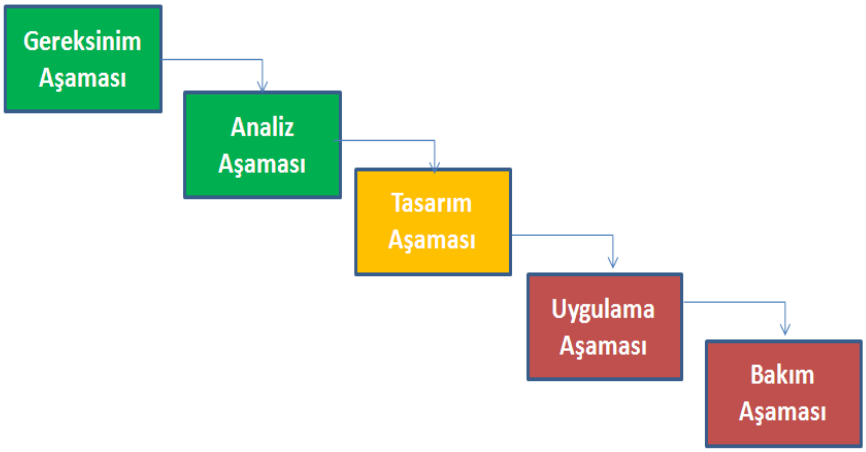
1. Çağlayan Modeli
2. Evrimsel-Ağaç (Evaluation-Tree) Modeli
3. RAD(Rapid Application Development)Hızlı Uygulama Geliştirme
4. Spiral Model (Barry Bohem tarafından ortaya atılan)
5. EVO Modeli
6. Agile Metotlar (Extreme Programming, Scrum, …vb. yöntemler)
7. Yapısal Metotlar (Jackson Structured Method, … vb. metotlar)
8. Tekrarlamalı ve Artırımlı (Iterative and Incremental) Yaşam Döngü Modeli
9. Süreç Yaklaşımı (İnce, 2004).

Günümüzde geliştirilmiş yüzden fazla metodoloji bulunmaktadır. Geliştirilen bu metodolojilerde genelde çağlayan ya da spiral model temel alınmıştır. Diğer bir taraftan özel kuruluşlarda uzmanlar tarafından geliştirilmiş, birçok kuruluş tarafından benimsenmiş ve uygulanmakta olan çeşitli metodolojiler vardır. Bir metodolojide bulunması gereken temel bileşenler ya da özellikler aşağıda listelenmiştir (Kan, 2002).

- Ayrıntılı bir süreç modeli,
- Ayrıntılı süreç tanımları,
- İyi tanımlanmış üretim yöntemleri,
- Süreçler arası arayüz tanımları,
- Ayrıntılı girdi tanımları,
- Ayrıntılı çıktı tanımları,
- Proje yönetim modeli,
- Konfigürasyon yönetim modeli,
- Maliyet yönetim modeli,
- Kalite yönetim modeli,
- Risk yönetim modeli,
- Değişiklik yönetim modeli,
- Kullanıcı ara yüz ve ilişki modeli,
- Standartlar.

Bir kuruluşun kendi metodolojisini geliştirmesi ve uygulaması oldukça zaman alıcı ve uzmanlık gerektiren bir konudur. İzleyen kısımda yazılım geliştirme metodolojilerinden bahsedilmiş ve konuya açıklık getirmesi açısından örnekler verilmiştir.

Çağlayan Modeli: En eski, en tanınmış, en temel modeldir. Bu modelde oluşturulacak sistemlerin her birini bir proje olarak ele almak gerekmektedir. Bu model bazı hükümet standartlarına girmiştir. Baştan tanımlanmış sistemler için daha çok tercih edilmektedir. Yazılımın üretilmesi aşamasında, yazılımcı ile müşterinin çok sık bir araya gelmediği durumlarda uygulanan bir modeldir. Müşterinin istekleri farklılaşmayacaksa yani müşteri istekleri esnek değilse bu modelin kullanılması uygundur. Burada "Requirements Paradox"'a dikkat edilmelidir. Yazılımın sağlıklı ve kaliteli olması için isterlerin sabit ve kararlı olması gerekir (Kan, 2002). Şekil 1-1'de Çağlayan Modeli gösterilmektedir. Bu modelde yaşam çevrimi, bir yazılım ürününün başlatıldığı andan ürünün kullanımdan kalktığı ana kadar geçen süre olarak adlandırılır. İsterlerin alınması ve planlama, sistem incelemesi, tasarım, gerçekleştirim ve bakım olarak 5 alt başlığı vardır.



Şekil 1-1Çağlayan Modeli(Kan, 2002)

Gereksinim Aşaması: Proje ile ilgili gereksinimler belirlenir. Yeni oluşturulacak sistemin amaçları, hedefleri ve oluşturulacak sistemin özellikleri belirlenmeye çalışılır (Sahach, 2007).

Analiz Aşaması: Oluşturulacak sistem ile ilgili detaylı bir inceleme ve analiz yapılır. Sistemin çözümlemesi yapılır. Kullanıcının yazılım bittikten sonra yapacağı işler modellenir. Kullanıcı istekleri detaylı olarak gözden geçirilir (Sarıdoğan, 2004).

Tasarım Aşaması: Proje için önce genel bir tasarım gerçekleştirilmesiyle başlar. Daha sonra bu genel tasarım yapısı bozulmadan projenin alt kısımları için daha detaylı bir tasarım hazırlanır. Tasarımları hazırlamak için eldeki imkânlar değerlendirilir. Sahip olunan bilgi birikimine bakılır. Tasarımlar oluşturulurken yazılımcı tarafındaki kişilerle birlikte müşteri temsilcilerinin de olması projenin daha sağlam temeller üzerine oturtulmasına yardımcı olacaktır (Sahach, 2007).

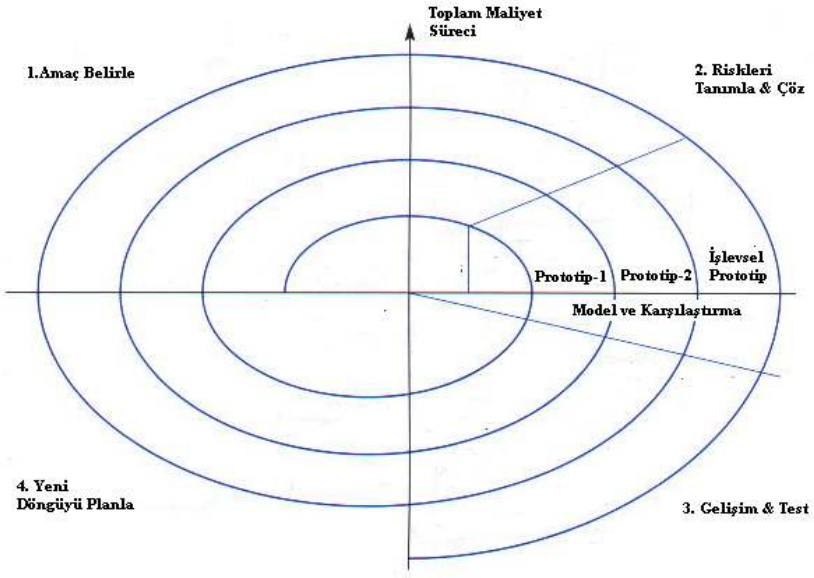
Uygulama Aşaması: Sistem incelemesi ve sistem tasarımından gelen bilgi birikimi ile sistem gerçekleştirimine ait bir alt yapı oluşturur. Sağlanan bu alt yapı ile sistem gerçekleştirilmesi işlemine geçilir. Burada belli kurallara göre, sistemin kaynak

kodları yazılır. Sistem testi ve kabul testleri de bu aşama içerisinde yapılmaktadır (Sahach, 2007).

Bakım Aşaması: Yazılım test aşamasından sonra ortaya çıkabilecek eksiklikler ve aksaklıkların çözümünün yapıldığı safhadır. Bu safhada yeni gelen isterler varsa incelenip yeniden bir yaşam döngüsü içerisinde değerlendirilir (Sahach, 2007).

Evrimsel-Ağaç Modeli: Evrimsel-ağaç modeli, çağlayan modelinin geliştirilmiş bir şeklidir. Gerçekleştirilecek büyük bir proje küçük parçalara ayrılır. Parçalara ayrılan sistem aşama aşama gerçekleştirilir. İsterler, analiz, tasarım, kodlama, sınama aşamaları paralel veya teker teker gerçekleştirilir. Her parça için küçük çapta bir çağlayan modeli uygulanır (İnce, 2004).

Bohem Spiral Modeli: Çağlayan modelinin ve prototipleme yaklaşımının birleşmiş şekli olarak düşünülebilir. Proje dört ana başlığa ayrılır ve her başlık kendi içerisinde daha detaylı bir bakış açısıyla çözülmeye çalışılır. Şekil 1-2'de Bohem Spiral modeli gösterilmektedir.



Şekil 1-2 Bohem Spiral Model (Group, 1998)

Her bir aşama için planlama, geliştirme, risk çözümleme, üretim ve kullanıcı değerlendirmesi yapılır. Risk analizi, gözlem, yaşam döngüsü planı, seçeneklerin

değerlendirilmesi kısımları vardır. Her aşamada bir prototip oluşturulur. Bu prototipler geliştirilerek son prototip ortaya çıkartılır (Group, 1998).

EVO Modeli: Her hafta yapılacak işler belirlenir. Hafta başı isterlerin analizi yapılır, görevler belirlenir. Hafta sonuna kadar verilen bu görevler gerçekleştirilmeye çalıştırılır. Hafta sonunda da sonuçların doğrulanması ve değerlendirilmesi işleri yapılır. Yeni gelen isterler doğrultusunda EVO modeli tekrar kullanılır. Her çevrim sonucunda mutlaka bir ürün teslim edilir. Yazılım geliştirilmesi için tanımlanan temel çevrim bir ila üç hafta arasıdır. Tanımlanan stratejik hedeflerin gerçekleştirilmesi için ise bu süre bir ila üç ay arası, planlanan organizasyon hedeflerin tamamlanması için ise; üç ila on iki ay arasıdır.

EVO modelinde; yazılım geliştirilmesi planlanırken, zaman kutulama ve özellik kutulama işlemleri yapılır. Bu sayede hangi zaman dilimi içerisinde hangi özelliklerin yapılacağı belirlenmiş olur. Zaman kutulamada belirtilen zaman diliminde hangi özelliklerin gerçekleştirileceği, özellik kutulamada ise istenilen özelliklerin gerçekleşmesi için gereken zaman belirlenir (Deursen, 2009).

1.11. Yazılımda Kalite

Yazılımlar, özel hayatımızda ve iş hayatımızda yerini giderek arttırmaktadır. Bu durum daha geniş kapsamlı yazılımlar üretme ihtiyacını ortaya çıkartmıştır. Yazılımlardaki artışla birlikte üretilen yazılımların daha karmaşık bir yapıda sahip olmaları, yazılımların nasıl ve ne şekilde üretileceği sorusunu akla getirmiştir. 1968 yılında ilk kez ortaya çıkan yazılım mühendisliği kavramı ile birlikte o günden bu güne yazılımların nasıl ve ne şekilde yapılacağı sorusu üzerinde durulmuştur (Randell, 1968).

Yazılım üretimi işi için; yazılım geliştirme metodolojileri, programlama dilleri, çeşitli programlama araçları geliştirilmiştir. Bu gelişmelerin sağlanmasında pek çok mesleki kuruluş ve akademik kurum ortaklaşa çalışmıştır. Yazılım mühendisliğinin temelleri, yazılım mühendisliğinin ürettiği ürünlerin niteliklerini anlatan teorik, bilimsel, matematiksel yöntemlerden ve ana ilkelerden oluşur. Buradaki ana nokta

kaynakları, belirlenmiş bir amaca dönüştürmek için mühendislik tasarımı ve mühendislik bilimi uygulanarak en uygun modellemenin yapılabilmesidir (Grad, 2010).

Yazılımda kalite ve süreç konusunun iyi anlaşılabilmesi için,yazılımda kalite ve süreç ile ilgili kavramlara bakmak gerekmektedir. Yazılım kalitesini belirleyen temel özellikleri iki belli başlı kısma ayırabiliriz.

1. Yazılımın kullanımına ilişkin özellikler
2. Yazılımın geliştirilmesine ilişkin isterler

Yazılımın kullanımına ilişkin özellikler: Yazılımın kullanımındaki en önemli özellik, kullanıcının isteklerinin, kullanıcının isteğine uygun bir şekilde sağlanmasıdır (İnce, 2004). Gerçekleştirilecek yazılımın etkin bir şekilde kullanılması için bazı özelliklere sahip olması gerekmektedir. İzleyen kısımda bu özelliklerden bahsedilmişdir.

- **Kolaylık:** Yazılım kullanıcının tüm isterlerini karşılayabilmelidir. Ama bu isterleri karşılarken müşterinin bu işleri kolay menülerle basit bir şekilde yapmasına imkân sağlamak gerekmektedir. Menülerin estetik bir görünüme sahip olması, yazılımın kolaylığını ve karakteristiğini artıracaktır.
- **Doğruluk:** Yapılan yazılımların, müşterilerin ihtiyacı olan verileri düzgün bir şekilde oluşturarak hatasız bir şekilde müşteriye vermesidir. Bu sayede müşteriler bu yazılımı güvenle kullanabilirler.
- **Sağlamlık:** Yazılım oluşturulurken her türlü şart ve koşullar imkânlar dâhilinde ele alınmalı ve farklı şartlar altında yazılımın yapısının düzgün bir biçimde korunması sağlanmalıdır.
- **Verimli çalışma:** Yazılım çalıştırıldığında istenilen cevaplar kısa sürede elde edilmeli ve bu sonuçları elde etmek için kullanılan kaynaklar mümkün olduğunca az olmalıdır.
- **Korunmalı olma:** Yazılımı geliştiren kişilerin bilgisi olmadan yazılım ya da yazılımın ihtiyacı olan veriler üzerinde değişiklik yapılmamalı, veriler elektronik ortamda korunmalı.

- **Ekonomiklik:** Yapılacak ya da yapılan yazılımlar müşteriye büyük bir külfet getirmeden, sürekli olarak kullanılabilir olmalı ve imkânlar dâhilinde değişik donanımlarla da kullanılmalıdır.

Yazılımların Değerlendirilmesi: Yazılımların değerlendirilmesiyle ilgili olarak dikkat edilmesi gereken ana özellikler aşağıda listelenmiştir. Üretilen yazılımların bu özellikleri ve bilgileri sağladığından emin olunması gerekmektedir (İnce, 2004).

- **Denetlenebilirlik:** Yazılımın standartlara ne derece uygun olduğunun istenilen zamanda sorgulanmasıdır.
- **Doğruluk:** İsterleri eksiksiz yerine getirme kapasitesidir. Bu özellik aynı zamanda başarı için de önemlidir.
- **Hassaslık:** İşlemler sonucunda çıkan verilerin sayısal olarak doğruluk derecesi.
- **Veri Aktarımı:** Sistemin kullanacağı verilerin dış ortamdan düzgün bir biçimde alınıp alınamadığı konusunu kapsar.
- **Verilerin Yaygınlığı:** Yazılım karmaşasını engellemek için verilerin standart bir yapıda olması gerekmektedir. Standart veri tiplerinin kullanılması şarttır.
- **Bütünlük:** İstenen bütün işlerin bir arada gerçekleştirilmesidir.
- **Büyüklük:** Yazılımı oluşturan kaynak kod satır sayısı, modül sayısı, öz kaynak gereksinimi gibi değerlerdir.
- **Tutarlılık:** Yazılımın üretim aşamasında tasarım ve belgeleme için aynı tekniklerin kullanılması.
- **Hata Dayanıklılığı:** Oluşan bir hatanın hangi sistemlerden etkilendiğinin tespiti ve hatanın büyüklüğüne karşı sistemin sağlamlığıdır.
- **Genişleyebilirlik:** Yazılım mimarisinin istekleri karşılama derecesidir.
- **Müşteri Memnuniyeti:** Yapılan yazılımın müşterinin ihtiyaçlarını ne derece karşıladığı ve bu ihtiyaçların karşılanmasından sonraki müşteri tatminidir.
- **Belgeleme:** Yazılım üretimi aşamalarında belgelemenin açık ve anlaşılır bir şekilde yapılıp yapılmadığıdır.
- **Değişik Yapılarda Çalışabilmesi:** Üretilen yazılımın mümkün mertebe değişik sistemler üzerinde çalışabilmesidir.

- **İzlenebilirlik:** Programın kendi hatalarını kendi mekanizması ile gösterebilmesidir.
- **Modülerlik:** Bir bütün olan programın kendi içyapısının işlevsel olarak ayrılabilir olması. Bu özellik sistem içerisinde sağlanıyorsa, belli bölümlerdeki yeni istekler için sadece o bölümle ilgili yerlerin değiştirilmesi gerekir. Bu sayede isterlerin daha kolay bir şekilde gerçekleştirilmesi sağlanabilir.
- **Kullanım Kolaylığı:** Kullanıcı ara yüzlerinin ve programın kullanımının kolay olmasıdır.
- **Bakım Kolaylığı:** Sisteme sonradan ilave edilecek geliştirici ve iyileştirici kısımların kısa sürede yapılabilmesidir.

1.12. Süreç

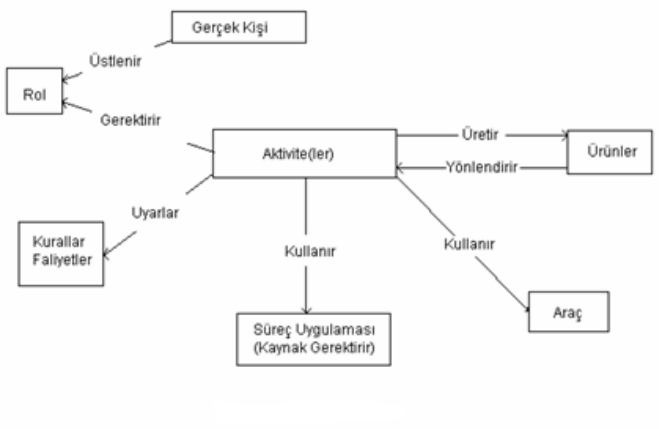
Süreç, kısa tanımıyla bir işi yapma yoludur. Genelde bir ya da birkaç adımdan oluşur. Bir süreci iyi tanımlamak, o sürecin amacını, ürünlerini, aktörlerini tanımlayarak olur. Eğer bir süreç yeteri kadar iyi bir şekilde tanımlanmaz ise hatalar belirlenemez. Başka bir deyişle süreç olgunlaştırılmaz ise ürün kalitesi sadece tesadüflere kalır. Süreç yaklaşımı özellikle yazılım için daha da geçerli ve önemlidir. Çünkü sürecin doğruluğu sadece geliştirilen yazılımın çalışmasıyla anlaşılabilir. Sürecin amacı, bir standardı oluşturmaktır. Yazılım süreçlerini iyi bir şekilde belgelemek gerekmektedir. Bu belgeleme, yazılı veya grafiksel olabilir. Bir süreç başka bir yerde tekrardan oluşabilir; süreçler tekrarlı olabilir. Bir diğer unsur da yazılımların oluşturulmasında süreç olgunluğudur. Süreç için çok önemli olan "süreç yönetimi" de bir başka önemli konudur. Süreç yönetimi ve kalite yönetimi birbirini sürekli olarak desteklemesi gereken unsurlardır. Bir ürünün yazılım için kalitesi onu incelemekle belirlenmez, büyük oranda onu üreten organizasyon ve süreçlerine bağlıdır (Sarıdoğan, 2004).

Süreç teknolojisinin amacı bir süreç modelini ortaya koymak ve tüm yazılım üretimini bu modele göre değerlendirmektedir. Süreçler sürekli gözden geçirilmeli ve iyileştirilmelidir.

Süreçlerin Ortak Özellikleri: Süreçlerin ortak özellikleri aşağıda sunulmuştur.

- Bir işi yapma yoludur.
- Alt süreç ve adımlardan oluşur.
- Yazılı ve grafiksel olarak belgelenmiştir.
- Tekrarlanmak üzere yapılmıştır.
- Girdileri ve çıktıları vardır.
- Genel amaç değişkenliği azaltmaktır.
- Ölçmeyi ve iyileşmeyi sağlar.

Süreç meta modelindeki kaynak; insan, para, yer, malzeme, araç, yazılım, bilgi ve zaman olarak değerlendirilebilir (İnce, 2004). Şekil 1-3'de süreç meta modeli gösterilmektedir.



Şekil 1-3 Süreç meta modeli (İnce, 2004)

Süreç Modelleri: Yazılım projelerini gerçek hayata geçirirken çeşitli süreç modelleri kullanılır. Bu süreç modelleri daha önceki bölümde gördüğümüz çağlayan modeli, evrimsel model, spiral süreç modelleri olabileceği gibi çevrim süresini azaltmayı hedefleyen hızlı uygulama geliştirmeyi sağlayan RAD (Rapid Application Development) modeli de olabilir. Bu konuda Çağlayan ve Bohem Spirali modellerinin artılarını kullanan yöntem olan kilometre taşı temelli geliştirme (Milestone-Based Development) modeli veya işlerin paralel olarak yapıldığı paralel geliştirme (Concurrent Development) süreç modelleri örnek olarak verilebilir (İnce, 2004). Bir

yazılım esnasında hangi yöntemin ne şekilde hangi projeler için kullanacağına yazılım firmasındaki, bu işler üzerine deneyimli olan kişilerin karar vermesi gerekir.

1.13.Yazılım Süreç İyileştirme Modelleri

Bu kısımda yazılımda süreç geliştirme ve iyileştirme modellerinin neler olduğu konularından bahsedilecektir. Bu modellerin birbirlerini nasıl etkiledikleri ve yazılım geliştirme işlerine nasıl bir bakış açısı ile yaklaştıkları üzerinde durulacaktır. Yazılım süreç geliştirme modelleri olarak zaman içerisinde, AQAP 150/160, TICKIT, TRILLIUM, ISO 12207, ISO 15504 (SPICE), BOOTSTRAP, CMM, CMMI gibi yazılım geliştirme modelleri oluşturulmuştur (İnce,2004). Aşağıda kullanılan süreç geliştirme ve iyileştirme modellerinden bazıları ile ilgili bilgiler verilmektedir.

AQAP: Bir NATO standardı olan AQAP (Allied Nato Quality Assurance Program), kalite güvence sistemi NATO'ya bağlı savunma sanayi kurumlarının kalite standartları için geliştirilmiştir. Aşağıda çeşitli AQAP standartları sunulmuştur:

- AQAP-2110
- AQAP-2120
- AQAP-2130
- AQAP-2131
- AQAP-2009
- AQAP-2000
- AQAP-150/160

AQAP standartları NATO'nun kalite ve güvence için geliştirdiği bir standarttır. AQAP-150/160 ise savunma sanayinin ihtiyaçları doğrultusunda yazılım geliştirme süreçleri için geliştirilmiştir. AQAP kullanarak geliştirilen ürünler, ileri teknolojinin kullanıldığı bilgi teknolojileri hizmetleri, yazılım tasarım ve üretimi, sistem tasarımı, sistem geliştirme, entegrasyon, savunma, iletişim yazılımları, sistem geliştirme, sistem bakım, onarım ve destek hizmet sunumu, simülatör geliştirilmesi gibi ürünlerdir. Bu ürünlere baktığımız zaman hepsinin son derece karmaşık yapılar olduğunu görürüz. AQAP kullanılarak gerçekleştirilen yazılımlar, isterler doğrultusunda tasarlanır ve buna göre geliştirilirler. NATO'ya bağlı ülkelerin askeri amaçlı olarak kullandığı, kalite ve güvence için geliştirilmiş bir standarttır (Software Develop Team, 2001).

IEEE/EIA 12207: ISO/IEC 12207 standardının Amerikalılar tarafından gözden geçirilip bazı ilaveler yapılmasından sonra oluşturulmuş bir standarttır. Bu standart, ISO/IEC 12207 standardını tamamıyla kapsamakta ve ilave olarak yeni kavramlar getirmektedir. Yapısı içerisinde proje bütçesi, yazılım maliyeti, müşteri, edinici, geliştirici, alt yüklenici, yazılım geliştirme kavramlarını detaylı olarak ele alır. Kendisinden önce geliştirilmiş projelerin IEEE/EIA 12207'ye geçirilmesi gerektiği durumlar da yazılımların kolaylıkla uyarlanabilmesini sağlar. IEEE/EIA 12207, üç cilt halinde yayınlanmıştır (Gray, 2008).

IEEE/EIA 12207.0–1996 ABD için açıklamalarıyla ISO/IEC 12207 ve ekler.

IEEE/EIA 12207.1–1997 Belgelendirme için yardımcı dokümanlar ve kılavuz.

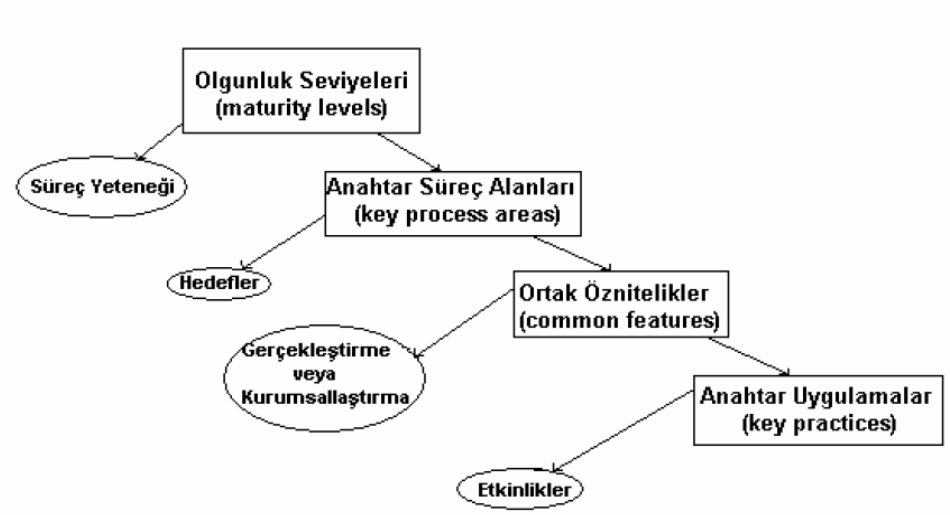
IEEE/EIA 12207.2–1997 ISO/IEC 12207'de farklı bakış açılarına sahip alternatif uygulama yaklaşımı.

IEEE/EIA 12207 standardı; hedefinin büyük ve karmaşık projeler olmasına rağmen, orta ve küçük ölçekli firmaların projelerinde de uygulanabilmektedir. Belgeleme şart koşulmakla birlikte, belgelerin içeriği veya biçimi tanımlanmamaktadır. Bu standardın etkili olabilmesi için üst yönetimle birlikte şirketin bütün çalışanlarının o işe gönülden bağlı olması, personel eğitimi ve şirket politikasının iyi anlatılmış olması gerekmektedir (Gray, 2008).

CMMI (Capability Maturity Model Integrated) Yetenek Olgunluk Modeli: CMM etkin bir yazılımın süreçlerini, anahtar elemanlarını tanımlayan bir çerçeve modeldir; olgun olmayan bir süreçten olgun ve disiplinli bir sürece giden evrimsel bir yol çizer. CMM, yazılım sürecinin olgunluğu üzerine hüküm vermek ve endüstrideki pratiklerle karşılaştırmak için bir ölçüm aracıdır (Shrum, 2007).

CMMI'ın basamaklı bir yapısı vardır. Yazılım süreç iyileştirmesinin, organizasyonun stratejik planları ve iş hedefleri, yapısı, kullandığı teknoloji ve sosyal kültürü ile bağlantılı olduğunu bilen CMMI buna göre yazılımı değerlendirir. Şekil 1-4'de CMMI'ın yapısı gösterilmektedir.

- Yazılım, tanımlı süreçlerde yürütülmüyorsa, başarı şansa ve koşullara kalmıştır.
- Sürekli başarı, tanımlı süreçlerle olur.
- CMMI, olgun olmayan bir kuruluşun, olgun bir kuruluşa giden adımlarını belirler.
- Yazılım süreç performansı, elde edilen sonuçlarla ortaya çıkar.
- Yetenek, olgunluğa bağlıdır, olgunlukla gelir (Shrum, 2007).



Şekil 1-4 CMMI'ın Yapısı (İnce, 2004)

CMMI (Süreç Olgunluk Çerçevesi): Organizasyon çapında yazılım sürecinin tanımlı olmadığı durumlarda, projelerde başarının tekrarı kişilere bağımlıdır. Bu durum, uzun süreli verimliliğe ve kaliteyi iyileştirmeye olanak vermez.

Sürekli iyileşme ancak, etkin yazılım mühendisliği ve yönetim pratiklerinden oluşan bir süreç altyapısı inşa edilerek mümkündür. Aynı zamanda CMMI olgun bir yazılım organizasyonuna giden aşamaları, öncelik sırasıyla veren bir modeldir (Shrum, 2007).

CMMI yapısı: CMMI'ın yapısı beş düzeyden oluşur. Her düzeyin kendi içerisinde tanımlanmış anahtar süreçleri mevcuttur.

CMMI Düzeyleri:

1. **İlk Düzey(Başlangıç Düzeyi):** Yazılım süreci, gelişi güzeldir. Bazen de kaotiktir. Başarı, kişisel çabalara bağlıdır. Süreçler tanımlanmışsa da uyulmaz. Kriz durumunda yapılan plan terk edilir. Her proje farklı yapılır. Lider önemli ve etkindir. Yine de iyi projeler çıkartabilir. En çok yapılan iş kodlama ve testtir.
2. **Tekrarlanabilir Düzey:** Temel proje usulleri konulmuştur. Zaman planı, maliyet, insan gücü, işlevsellik planlanır ve izlenir. Ürün dayanakları oluşturulmaktadır. Bu ürün dayanaklarına baseline adı verilir.
3. **Tanımlı Düzey:** Süreçler organizasyon çapında tanımlanmıştır ve belgelenmiştir. Eğitim verilmektedir. Yazılım geliştirme, yazılım yönetimi ve kurum yönetim süreçleri entegredir. Her projedeki standart süreçler o projeye uygulanır. SEPG (Sistem Mühendisi Süreç Grubu) grubu vardır.
4. **Yönetilen Düzey:** Yazılım sürecinin ve ürün kalitesinin ayrıntılı sayısal ölçümleri toplanır. Süreçler nicel olarak anlaşılmıştır.
5. **En iyileşen Düzey:** Süreç uygulamalarından gelen nicel bilgilerle, yeni teknoloji ve buluşlarla yapılan pilot uygulamalarda sürekli iyileşme vardır. Organizasyon sürekli iyileşmeye odaklıdır. Süreç değişikliği ve teknoloji değişikliği ile iyileşme sürekli olur (Shrum, 2007).

2. BÖLÜM

YAZILIMDA TEST YÖNTEMLERİ

Yazılımın test edilmesindeki temel amaç, belirlenen isterler ile gerçekleşen yazılım arasındaki farkları bulmaktır (IEEE Standard,1998). Diğer bir amaç da yazılımın çalışması süresince ortaya çıkabilecek hataların, mümkün olan en kısa sürede bulunmasıdır. Yazılım testi, bir program ya da sistemin hatalarını bulmak, özelliklerinin veya yeteneklerinin istenen sonuçlara uygunluğunu belirlemek amacıyla yapılır (Hetzel, 1998). Yazılımın içerisinde bulunan bütün hataları bulmak mümkün değildir (Springer, 2003). Yazılım içerisindeki hatalar çok farklı noktalardan meydana gelebilir. Yazılım hataları, yazılım geliştirmenin herhangi bir evresinde, isterlerin yanlış tanımlanması, iletişim eksiklikleri, süreçlerin eksik belirlenmesi, vb nedenlerden oluşabilir (McGregor, Sykes, 2001). Yazılım testi hata bulmaya odaklıdır, hataları düzeltmeyi içermez (Sommerville, 2006). Yazılımı test etmek ilk başta proje maliyetini arttırır gibi gözükse de, yazılımın test edilmesi, hataların en az seviyeye indirilmesi için çok önemlidir. Bu sayede sonradan olabilecek ve proje maliyetini arttıracak hatalar giderilmeye çalışılır (Burnstein,2003).

Yazılımı test etmek için kullanılan dört temel test vardır. Bu testler objeye yönelik yazılım geliştirme süreçlerinde kullanılabilir.

1. Birim/Sınıf (unit) Testi
2. Bütünleme (integration) Testi
3. Onaylama (validation) Testi
4. Metot(method) Testi

Temel olarak iki tür test yöntemi vardır:

1. White Box

   Test kodunun çalıştırılmasına dayanan bu teknikte yazılımın cümlelerinin, kodun izleyebileceği bütün yolların, koşul cümlelerinin ve veri akışının doğruluğunun onaylanması gerekmektedir. Yazılım geliştirmek için en çok kullanılan unit test yöntemidir. Geliştirilecek yazılıma ilişlin özelliklerin iyi tanımlandığı ve veri tanımlarının düzgün yapıldığı varsayımlarına dayanır. Aşağıda white box test teknikleri listelenmiştir.

   a. Cümle kapsama tekniği (Statement coverage)

   b. Dal (basit koşul) kapsama tekniği (Branch coverage)

   c. Birleşik koşul kapsama tekniği (United branch coverage)

   d. Yol kapsama tekniği (Path coverage)

   e. Döngü kapsama tekniği (Loop coverage)

2. Black Box

Yazılımın şartnamedeki özelliklere uygun olarak ilerleyip ilerlemediğinin test edilmesinde kullanılır. Her programın, kendi girdi kümesindeki değerleri kendi çıktı aralığındaki değerlere dönüştüren bir fonksiyon olduğu bakış açısına dayanır. Black Black box testte program şartnamesine göre ve hiçbir kod yazılmadan gerçekleştirilir.

2.1 Birim Testi

Bu yöntemde yazılımcı, yazılım kodunu oluşturan birimleri test ederek kullanıma hazır olup olmadığına bakar. Birim, yazılımda test edilebilecek en küçük bölüme denir. Yordamsal yazılımda bir birim özgün bir program, bir işlev veya prosedür olabilirken nesnel tabanlı programlamada bu bir süper, soyut ya da türemiş sınıfa âit bir yöntem de olabilir. Nesneye yönelik yazılımlarda kullanılan sınıf testi, geleneksel yöntemde birim testlerine karşılık gelir. Geleneksel yazılımlarda bir modülün algoritma detayları ve modül

ara yüzündeki veri akışları ile ilgilen birim testlerinin aksine, nesneye yönelik sistemlerde sınıf testleri, sınıf tarafından çevrelenmiş ve sınıfın yapısal davranışlarına göre belirlenir (Vardarlı, 2008). Daha temel bir ifade ile yazılımın içindeki her bir birimin farklı kriterlere göre çalıştırılıp test edilmesine dayanır. Çoğunlukla alfa testi öncesinde gerçekleştirilir.

2.2 Bütünleme Testi

Nesneye yönelik olarak geliştirilen yazılımlarda hiyerarşik bir kontrol yapısı yoktur. Bu nedenden dolayı yukarıdan-aşağı ve aşağıdan-yukarı bütünleme stratejileri kullanılmaz (Pressman, 2004). Nesneye yönelik sistemlerin bütünleştirilmesinde iki farklı yöntem uygulanmaktadır (Robert, 2000). Her bir iş parçacığı tek tek bütünleştirilir ve test edilir. Yan etkilerin olmaması için bağlanım (regression) testi uygulanır. İkinci bütünleme yaklaşımı olan kullanış tabanlı test sistemi kurmaya, servis sınıflarını kullanmayan veya çok az kullanan bağımsız sınıflar olarak adlandırılan sınıfların test edilmesi ile başlar. Bağımsız sınıflar test edildikten sonra bir sonraki aşama, bağımsız sınıfları kullanan bağımlı sınıfların test edilmesidir. Bu bir dizi bağımlı sınıf katmanlarının test edilmesi, bütün sistem kurulana kadar devam eder (Gregor**,** Sykes, 2001). Bu test çevrimsel artıklık denetimi (cyclic redundancy check) ve nesne-ilişki (entity relationship) modeli kullanılarak belirlenir. Burada kullanılan test programları sayesinde, ortaklık yapan bir grup sınıf ile, bu ortaklıkta oluşabilecek hatalar ortaya çıkarılmaya çalışılır.

2.3 Onaylama Testi

Onaylama testi, gerçekleştirilen yazılım sisteminin, ara yüzleri, kullanıcının görebildiği faaliyetler ve kullanıcının tanıyabildiği sistem çıktılarına bakılarak yapılır. Onaylama testinde, analiz modelinde kullanılan kullanıcı senaryolarından faydalanılır. Yazılım programlarının hazırlanmasına yardımcı olmak için, test uygulayıcısı, kullanım senaryosunun kullanıcı etkileşim isterlerindeki adımlarına bakarak yazılım onaylama testi yapar. Burada çıkan hataların birim maliyeti çoğunlukla birim test ve bütünleme testlerinden fazladır. Onaylama testlerinde Black Box test yöntemi de kullanılabilir.

Bununla beraber yazılımın onaylama testlerinde, yazılım ile ilgili yapılmış analizler ve akış şemalarından da yararlanılabilir (Vardarlı, 2008).

2.4 Metot Testi

Nesneye yönelik programlamada metotların her biri bir alt yordama benzer. Metot tanımı, tek bir görevi yerine getirmek için, bir programcı tarafından oluşturulmuş kod bütünüdür. Metotların çoğunda bir metot kendine verilen işi gerçekleştirmek için başka bir metot kullanır. Bu nedenle metot testlerinde sadece o metot değil onunla bağlantılı olan diğer metotlarda birlikte test edilir. Kısaca bir metot bir birim olarak düşünülebilir (Everett, McLeod,2007).

Nesneye yönelik programlama içerisinde, sınıf kavramı da vardır. Sınıf tanımı kendi içerisinde bir nesnenin özelliklerini taşır. Nesnenin içerisinde kendi yapısı ile ilgili özellikleri ve davranışları vardır. Nesne, sınıfın her bir özelliğidir. Bazı durumlarda sınıf test işlemi için, sınıfın izole edilmesi gerekir. Ancak bu kalıtımın çok az kullanıldığı ve alt sınıfların izole bir biçimde test edildiği yerlerde uygulanabilir (Paul Jorgensen, 2006).

Sınıf bir birim gibi düşünülse bile sınıf içerisindeki metotların ayrı ayrı test edilmesi gerekmektedir. Bir başka ifade ile bir metot tek başına test edilse de metotlar sınıflardan ayrı var olamazlar. Sınıf testinde amaçlanan sonuç, tüm metotların iş birliği içinde çalıştıklarının ortaya konulmasıdır (Binder,2000).

Geleneksel beyaz kutu test yaklaşımı (white box test) kontrol edilecek verinin akışı yöntemine dayanır. Bu veri akış yöntemleri, test işleminin büyük bir kısmını üstlenirler. Ancak bu yöntemler sınıf testi için yetersizdir. Sınıf testinin daha kapsamlı ele alınması için kalıtımın var olduğu durumlar göz önüne alınmalıdır.

2.5. White Box Test

Beyaz kutu testi, yapısal test ve mantıksal test olarak da isimlendirilebilmektedir. Yazılımların test edilmesinde kullanılan önemli bir yöntemdir. Beyaz Kutu Testinin

amacı, yazılım kodları içerisinde, kodun izleyebileceği bütün yollara bakarak, tanımlanan koşulların ve algoritmanın doğruluğunun onaylanmasıdır. Yapısal testte yerine getirilmeyen işlevler belirlenemez. Diğer bir ifade ile Beyaz Kutu Testlerinde amaç, yazılımın kodlarına bakarak kodun bütün dallarının test edilmesini sağlamaktır. Test edilme işleri sayesinde program ile ilgili test verileri toplanır. Beyaz Kutu test tekniğinde bütün karar noktaları tanımlanır. Tanımlanan bu mantıksal noktalar için testler belirlenir. Sınır değerleri test edilir. Beyaz kutu testi yazılımın içyapısının testine dayanır. Yazılımın istenilen şekilde çalışıp çalışmadığını anlamamızı sağlayan bir test tekniğidir. Yazılım geliştirilmesi içerisinde belli dönemlerde kullanılması gerekmektedir (Gerald, 2007).

White box test tekniği en çok birim testlerinde kullanılır. Birim testlerinde elde edilen en büyük avantaj, white box test teknikleri ile elde edilir. Bu tekniklerin kullanımı sayesinde hata oranları azaltılabilir. Kod karmaşıklığı ile white box test tekniğinin sağladığı avantajlar ters orantılıdır. Kod karmaşıklığının az olduğu noktalarda white box test tekniklerinden alınacak fayda miktarı daha fazladır. Bu teknikleri kullanarak yazılım testini hazırlayacak kişi, test ile ilgili olarak veri akışlarını belirlemek ve tasarımını sağlamak zorundadır. Beyaz kutu testi belli test tekniklerinin birleşiminden oluşur. Kullanılan bütün yöntemler yazılım kodları ile ilgilidir (Gerald, 2007).

Cümle Kapsama Tekniği: Bu teknik beyaz kutu testi içerisinde en fazla kullanılan tekniktir. Bu test tekniğinin amacı, yazılım içerisindeki her bir kod parçasının çalıştırılmasıdır. Bu tekniğin etkin bir şekilde kullanımı için bu amaç doğrultusunda geliştirilen test araçları kullanılmalıdır. Bu sayede kodun kullanımı ile ilgili bilgiler elde edilebilir.

Kullanılacak bu test araçları sayesinde kodun kullanım bilgisi elde edilebilir. Cümle kapsama tekniği etkili bir teknik olmasına rağmen tüm olası sonuçların denenmesi mümkün olmadığı için tek başına yeterli bir teknik değildir (Schach S R, 2007). Aşağıda cümle kapsama tekniği ile ilgili bir örnek verilmiştir. Bu örnekte cümle kapsama tekniği ile bulunamayacak bir hata gösterilmeye çalışılmıştır. Sonuç

değişkenine doğru değer atanması engellenilmiştir. Bu hatanın ortaya çıkarılması için farklı değerler ile test edilmesi gerekmektedir (Gerald, 2007).

Aşağıdaki kodun testi için verilen test değerleri; txtbasla.text= 6 ve txtbitis.text =8 dir. Verilen bu değerler ile test yapılmak istendiğinde txtbasla.text >6 AND txtbitis.text =8 koşulu sağlamadığından kod çalıştırılamaz. Kod çalıştırılamadığından test işlemi gerçekleştirilemez ve hata yakalanamaz. Bu hatanın yakalanabilmesi için txtbasla.text ve txtbitis.text değişkene farklı değerler verilerek test edilmesi gerekmektedir.

```
IF ( txtbasla.text >6 AND txtbitis.text =8)

    txtsonuc.text=9;

End IF
```

White Box tekniği 5 alt teknikten oluşur:

1. Dal (basit koşul) kapsama tekniği
2. Birleşik koşul kapsama tekniği
3. Yol kapsama tekniği
4. Data flow tekniği
5. Döngü kapsama tekniği

Dal Kapsama Tekniği: Yazılım içerisindeki kod bloklarından alınan kodların her bir dalının test edilmesine dayanır. Yazılım içerisindeki mantıksal karar noktaları için en az bir kere alınan kod çalıştırılır, bu sayede oluşturulan yapı içersinden seçilen kod bütünü ile test edilmiş olur. Dal kaplama tekniğinin zor yanı, doğru ve yanlış sonuçlara erişebilmek için ayrı bir test programı gerekmektedir (Ilene,2003). Program içerisinde if ya da case gibi program akışını etkileyen durumlar komutlar vardır. Farklı değişkenler ile algoritma içerisindeki akışı değiştirmek mümkündür. Aşağıda verilen program örneğinde if komutu kullanılmış ve program içerisinde temp değişkenine ve list dizisi içerisinde atama işlemleri yapılmıştır. Test işlemi için belirlenen tek bir değer ve algoritma içerisinde seçilen tek bir yol ile test işlemi yapılmaktadır. Bir sonraki bölümde anlatılan birleşik yol kaplama tekniğinde ise farklı değerler ile birden fazla kez test işlemi gerçekleştirilmektedir. Aşağıda sunulan örnek bir dizi içerisindeki sayıların

büyükten küçüğe doğru sıralaması için verilmiştir. Dal kaplama tekniğinde test için N, last ve temp değişkenlerinin her birine tek bir değer ataması yapılarak list[] dizisi oluşturulmuştur. Yapılan bu atamalar ile algoritma içerisinde seçilen tek bir yol test edilmiştir. Bir sonraki kısımda anlatılan birleşik koşul kapsama tekniğinde ise değişkenlere farklı değerler verilerek, algoritma içerisindeki farklı yolların test edilmesi sağlanmıştır.

```
for (j=1; j<N; j++)  {
      last = N - j + 1;
     for (k=1; k<last; k++)  {
         if (list[k] > list[k+1])  {
               temp = list[k];
               list[k] = list[k+1];
               list[k+1] = temp;
         }
    }
}
print("Done\n");
```

Birleşik Koşul Kapsama Tekniği: Temel bir ifade ile dal kapsama tekniğinin birleştirilmiş değerler için uygulanmasıdır. Dal kapsama tekniğinden türetilmiştir. Yazılım oluşumu için kullanılan iç içe koşul cümlelerinin test edilmesine dayanır. Mantıksal operatörleri de bu yapı içerisinde kullanılmaktadır. Koşul cümleleri ve mantıksal operatörlerin artması ile karmaşıklığı doğru orantılı olarak artmaktadır. Aşağıdaki örnekte temel anlamda bir kapsama tekniği ile ilgili değer tablosu verilmiştir. Çizelge 2-1'de iki koşulla ilişkili bir kapsama tekniği doğruluk tablosu verilmiştir (Ilene,2003).

Çizelge 2-1 Doğruluk Çizelgesi

Ders Kodu	Sezon	Intibak
5000	1	**0**
25000	2	**0**
50000	1	**0**
75000	1	**0**
100000	2	**0**
125000	1	**1**

Aşağıdaki örnek kodun ele alındığını var sayalım

```
IF  ( DersKodu >= 100000 AND Sezon= 2 )
        Intibak = 1
ELSE
        Intibak = 0
END IF
```

İlgili kod parçasında iki farklı koşul tanımlanmıştır. Burada DersKodu değişkeninin 100.000'den büyük olması ve Sezon alanının 2 olması koşulunun sağlandığı durumlarda Intibak değişkenine 1 değeri atanmaktadır. Örneğin DersKodu değişkenine 125.000 değeri verilip Sezon değişkenine 1 değeri verildiğinde Intibak değeri 0 olmaktadır. Ancak Çizelge 2-1'de gösterildiği üzere DersKodu değişkenine 125.000 değeri verilip Sezon değişkenine 1 değeri verildiğinde Intibak değişkeninin 1 olması gerekmektedir. Kullanılan doğruluk çizelgesi sayesinde yukarıda verilen kod parçasın hatalı olduğu anlaşılmıştır.

Bir önceki konuda bahsedilen dal kapsama tekniği sadece 2 farklı değerin testi için kullanılır. Birleşik koşul kapsama tekniğinde ise iki farklı değişken ile ilgili, alt, eşitlik ve üst koşullar değerlendirmeye katılır. Yapılan değerlendirme ile sonuç değerleri karşılaştırılması sonucunda, istenilen sonuçlara ulaşılıp ulaşılmadığı anlaşılmaya çalışılır. Doğruluk testlerindeki amaç, programın çalışması sırasında oluşabilecek mantıksal hatalarının önüne geçmektir.

Yol Kapsama Tekniği: Yukarıda bahsedilen test teknikleri arasında en güçlüsü yol kapsama tekniğidir. Yol kapsama tekniği, yazılım içerisinde kullanılan kodların bir bölümünün alınması ile gerçekleştirilir (Torkar,R,2006). Alınan kodlardaki bütün mevcut yolların test edilmesine dayanır. Bu test sayesinde kod içerisindeki bütün satırların kullanılması sağlanır. Bu yöntemin en büyük zorluğu, büyük yazılımlarda yazılıma ait her bir satırın yol kapsama tekniği ile test edilmesinin getireceği maliyettir. Yazılıma ait en önemli kısımların testi için kullanılabilecek bir test tekniğidir (Richard, P, 2004).

Bu gün itibari ile kullanılan yazılımlar on binlerce hatta milyonlarca satırdan oluşmaktadır. Bu nedenle her bir satırın tek tek bütün koşullar ile test edilmesi imkânsızdır. Bu nedenle bazı kısıtlamalara gidilmesi çoğu durumda gerekebilmektedir. (Schach S. R, 2007). Bu kısıtlar aşağıdaki listede özetlenmiştir:

1. Lineer kod dizileri (linear code sequences)
2. Tüm tanım kullanım yollarının kapsanması (all-def-use-path coverage)
3. Döngüsel karmaşıklık ölçütü (cyclomatic complexity metric) (Richard, P, 2004).

Data Flow Tekniği: Yazılım içerisinde tanımlanan bir değişkenin hesaplama sonuçlarına göre testini içerir. Bu yöntemde değişkene farklı değerler atanıp daha önceden hesaplanan değerler ile tutarlı olup olmadığına bakılmaya çalışılır. Datanın; algoritma içerisinde ne şekilde değiştiğini anlamak için yapılan testtir. Algoritma üzerinde yeteri kadar yol (path) belirlenir. Seçilen yollara öncelik atanır. Testi gerçekleştirmek için yol üzerinde belirlenmiş bütün objeler en az bir kere çalıştırılır. Aşağıdaki Og_Ele_Ders_Yuk değişkenin farklı değerlerle test edilmesi amaçlanmıştır.

Test işlemi için ilk aşamada Og_Ele_Ders_Yuk değişkenine 0 değeri verilmiş, Sezon değişkeni ile çarpma işlemi yapıldıktan sonra Og_Ele_Ders_Yuk değişkeni hesap fonksiyonuna gönderilmiştir. Bu işlem farklı değerler ile test edilip elde edilen değerlerin doğru olup olmadıklarına Çizelge 2-2'deki doğruluk çizelgesinden bakılmıştır.

```
Og_Ele_Ders_Yuk   = 0;
Og_Ele_Ders_Yuk =  Og_Ele_Ders_Yuk  * Season;
Hesap (Og_Ele_Ders_Yuk)
```

Çizelge 2-2 Doğruluk Çizelgesi

Og_Ele_Ders_Yuk	Sezon	Hesap (Og_Ele_Ders_Yuk)
0	2	0
5	1	12
10	2	18
20	1	28

Yukarıda Og_Ele_Ders_Yuk ve Season değerlerine farklı değerler verilerek farklı Og_Ele_Ders_Yuk değerleri elde edilmeye çalışılır. Elde edilen değerler önceden belirlenen doğruluk tabloları ile karşılaştırılarak test yapılır.

Bu yöntemde hesaplama için seçilen değişkene farklı değerler gönderilerek, ortaya çıkan sonuçlar teker teker analiz edilir (Beizer, 1990).

Döngü Kapsama Tekniği: Yazılımlar içerisinde sıklıkla kullanılan yapılardan olan döngülerinde test edilmesi gerekmektedir. Yazılım içerisinde döngülerin doğru şekilde tanımlanmaması nedeniyle pek çok hata oluşabilmektedir (Beizer, 1990).

2.6 Black Box Test

Şartnameye yönelik yazılım test tekniğidir. Yazılımın şartnamedeki özelliklere uygun olarak ilerleyip ilerlemediğinin test edilmesi ile ilgili olarak kullanılır. Her programın, kendi girdi kümesindeki değerleri kendi çıktı aralığındaki değerlere dönüştüren bir fonksiyon olduğu bakış açısına dayanır. Programın şartnamesine göre ve hiçbir kod yazılmadan tasarlanabilir. Programın şartnamede belirtilen tüm işlevlere sahip olup olmadığı kontrol edilebilir ancak programın istenmeyen bazı davranışları içerip içermediği kontrol edilemez. Şekil 2-1'de kara kutu testi gösterilmektedir (Glenford, 2004).

Şekil 2-1 Kara Kutu Testi(Glenford, 2004)

Temel yazılım testleri, bütün yazılım yaşam döngüsü için bir doğrulama ve onaylama mekanizmalarıdır. Yazılım metotları kendi içerisinde Black Box ve White Box olarak ikiye ayrılır. Black Box ve White Box test teknikleri toplam 6 farklı test tekniğinden oluşur.Test case yazılımı ile amaçlanan, yazılım içerisinde olası durumlara bakılarak hata tespitlerinin yapılmasını sağlamaktır. Tanımlanan test durumları ile tekrarlanabilir bir şablon çıkartılır (Glenford J,2004).

Test'e başlarken yazılım testi, yazılımın süreçleri ve her bir süreç içerisindeki özelliklere bakarak isterlerle yapılanlar arasında bir farklılık var mı diye araştırır. Aynı zamanda yazılım testi ile meydana getirilen yazılımın özelikleri de belirlenmeye çalışılır (Pettichord, 2002).

3. BÖLÜM

AGILE PROGRAMLAMA

Yazılım mühendisliği, müşterinin ihtiyaçlarını karşılayan, öngörülmüş bütçe ile zamanında teslim edilen ve hatasız bir yazılım geliştirmek için teknik yöntemler öngören bir mühendislik disiplinidir (Schach, 2007). Bu bakımdan yazılım mühendisliği bir yöntemler kümesi, teknikler kümesi ya da araçlar kümesi olarak değerlendirilebilir. Yazılım mühendisliğinde, yazılım yaşam döngüsü içerisinde yer alan safhaların gerçekleştirilmesi yazılım üretimi için çok önemlidir (Fowler M, 1999).

Yazılım üretimindeki sorun, geliştirilen yazılımlardaki müşteri gereksinimlerinin artmasına paralel olarak yazılım üretimindeki karmaşıklığın artmasıdır. Proje içersinde meydana gelen değişiklikler, zaman ve bütçenin aşılmasına neden olmaktadır. Karşılaşılan sorunlar neticesinde, yazılımların büyük bir kısmının etkin bir şekilde kullanılmadığı görülmektedir (Keren, 2005).

Bu sorunların aşılması yönünde yapılan çalışmalar sonucunda çevik (agile) olarak adlandırılan ve yazılım yaşam döngüsünü baştan aşağı değiştiren yöntemler geliştirilmiştir. Bu yöntemler arasından en yaygın olarak kullanılanları, Uç Programlama (Extreme Programming – XP) ve Scrum'dır (Borandağ, 2006).

Extreme Programming ve Scrum dışında, Crystal Family, Feature Driven Development, Rational Unified Process for Agile, Dynamic System Development, Adaptive System Development, Open Source Software Development Lean Development ve Microsoft Solution Framework (MSF) gibi çevik yöntemler bulunmaktadır (Highsmith, 2002).

Çağlayan modeline karşı oluşan eleştiriler temel alınarak geliştirilen XP, yazılım geliştirmeye farklı bir bakış açısı getirmiştir. Çağlayan modeline karşı yapılan eleştiriler izleyen kısımda verilmiştir;

1. Sürüm yani çalışan program ortaya çok geç çıkmaktadır.
2. Sürüm çok geç çıktığı için, hatalar çok geç fark edilmektedir.
3. Süreçler verimli değildir.

4. Yeni gelen gereksinimler doğrultusunda kendi yapısını değiştiremediği için, yeni gereksinimleri karşılamada esnek değildir.
5. Değişiklikler geç ve zor yapılmaktadır (Acar, 2009).

Günümüzde pek çok şirket tarafından kullanılan XP (Bayerische Landesbank, Credit Swiss Life, DaimlerChrysler, First Union National Bank, Ford Motor Company and UBS.) projelerin daha hızlı bir şekilde gerçekleştirilmesini sağlamak amacı ile kullanılmaktadır. XP'nin, yazılım geliştirme sırasında bütün takımın aynı amaç doğrultusunda bir arada olması gibi kendisine özgü yöntemleri vardır (Dymond, 2006).

3.1 Extreme Programlama - XP

1990'lı yılların ortasında Kent Beck tarafından geliştirilen Extreme Programming -XP ile çevik yazılım üretim yöntemlerine farklı bir bakış açısı getirilmiştir. XP içeriği olarak dinamik ve değişken projelerde uygulanabilen bir yapıya sahiptir. En önemli avantajı, müşterilerle kurduğu iletişimdir. XP ile yazılım mühendisliği esaslarına uygun olarak, müşterinin değişen isteklerine hızla yanıt veren, hatasız ve en kısa sürede bir yazılım ürününün müşteri hizmetine sunulması amaçlamaktadır.

Ayrıca XP yöntemi, grup içi iletişime çok önem veren bir yazılım geliştirme yöntemidir (Highsmith,2002). Bir yazılım geliştirme disiplini olarak ortaya çıkan XP'nin amacı, kolay, iletişimin daha çok kullanıldığı, müşteri geri dönüş isterlerinin daha fazla olmasına imkân sağlayan bir yazılım geliştirme yöntemi sunmaktır (Torteli, 2005).

Çizelge 3-1'de de Agile yöntemlerden biri olan Extreme Programming metodu gelişim öncesine ilişkin tarihlere yer verilmiştir.

Çizelge 3-1 Extreme Programlama Gelişim Tarihleri (Goyal, 2004)

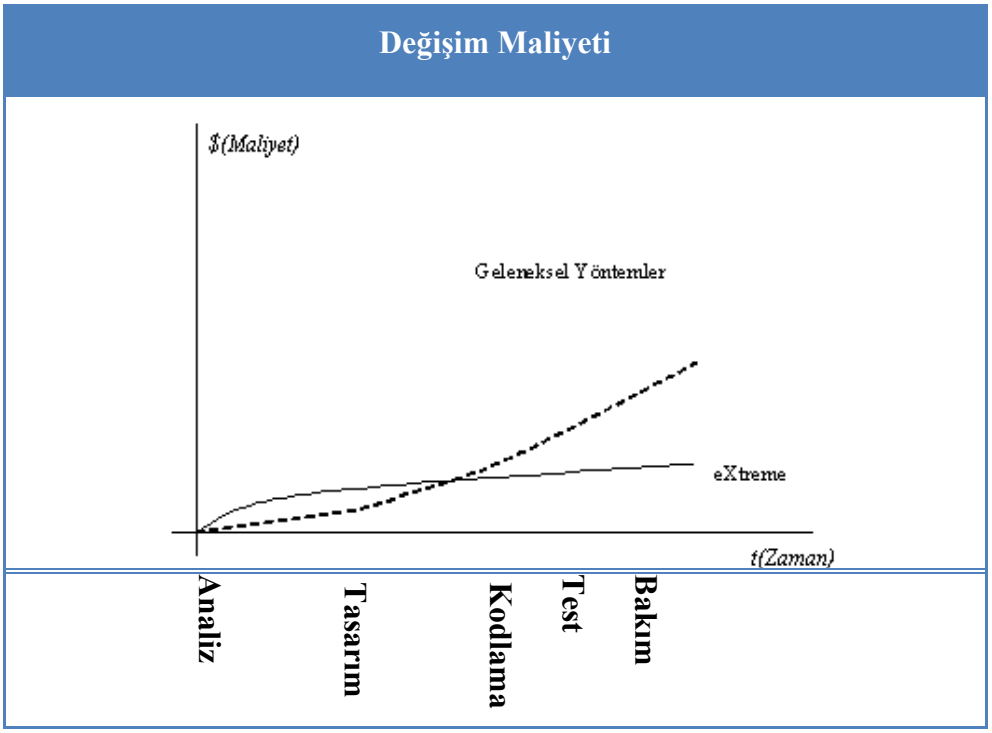
Yıl	Gelişim Tarihleri
1970	Bireysel XP Uygulamaları
1979	Karar verme (Alexander)
1986	Hızlı gelişme (Nonaka, Takeuchi and Cunningham)
1988	Evrimsel Gelişme (Gilb)
1988	Spiral Model (Bohem)
1994	Proje Özellikleri Planlama (Jacobsen)
1999	Programcılar için fiziksel ortam (De morca, Coplien)

3.2 XP'nin Bazı Avantajları ve XP Manifestosu

XP yazılımın erken safhalarında somut gelişmeler sağlayabilir. Yazılım içerisinde oluşan hataların erken safhalarda farkına varılabilir. Küçük yaşam döngüleri oluşturulur. Yazılım içerisinde geri dönüşler yapılarak hatalar telafi edilir (Hendrickson, 2000).

Artırımsal planlama yaklaşımı, yazılımın genel planının oluşturulmasını sağlar. Artırımsal yaklaşımla birlikte projenin gelişip, hayata geçmesi süreci tahmin edilmeye çalışılır. Esnek iş yürütme ve fonksiyonellik, işletmelerin değişen ihtiyaçlarına cevap vermeye çalışır. Program geliştirme süresince monitörlerden programcılar ve müşteriler ortak bir şekilde test yaparak, programın güvenliğini ve istenilenlere uygun olup olmadığını anlamaya çalışır. Şekil 3-1'de yazılım isterlerindeki farklılaşmanın getirdiği

değişim maliyeti geleneksel yaklaşımda artarken Extreme Programming'de o oranda bir artış olmadığı görülmektedir.



Şekil 3-1 Değişim Maliyeti ile XP in İlişkisi

Extreme Programlama manifestosu aşağıdaki maddelerden oluşmaktadır:

1. Gerçekleştirilecek yazılımı kullanacak kişiler müşterilerdir. Yazılım geliştirilirken müşteriden gelen öneriler çok önemlidir. Bu yüzden müşteri ile yazılımı gerçekleştirecek kurumlar arası ilişkiler son derece yakın olmalıdır.
2. Çalışan bir yazılımın kapsamlı belgelerden daha çok müşterilerin ihtiyaçlarını karşılaması önemlidir. Yüz yüze görüşme alternatif dokümantasyondan daha önemlidir.
3. Yazılımı gerçekleştiren kişilerle müşteriler arasındaki ilişkiler daha sıkı olmalıdır.
4. Yapılacak iş planı değişkendir. Plan izlemektense değişikliklere bakarak yol belirlenir.
5. Müşteriye erken ürün vererek müşteriden gelişen yazılımla ilgili sürekli bilgi alınmalıdır.
6. Değişen isterlere anında cevap vermek çok önemlidir.
7. İşin büyüklüğüne göre sürüm ortaya çıkarılmalıdır.

8. İş yapmak için motive olmuş insanlara gerek vardır. İş, motive olmuş insanlara verilmelidir.
9. Gelişmenin en iyi ölçüsü, gelişen yazılımdır.
10. Basitlik, en temel ilkedir.
11. Yazılımı gerçekleştirecek organizasyonlarda en iyi organizasyon 10-15 kişi arasıdır.
12. Belli aralıklarla durum değerlendirmesi yapılmalıdır (Beck, 2002).

3.3XP'nin Çalışma Mantığı

XP'nin çalışma mantığı, bazı basit uygulamalara(practices) bağlıdır. Başlangıçta bu uygulamalar çok basit ve gereksiz gelse de bir süre geçtikten sonra alınan başarılar ile kendi gücünü göstermektedir (Beck, 2002).

Geliştirilecek yazılımın kullanıcısı ile birlikte yazılım geliştiricilerin aktif olarak proje içerisinde bulunması, projenin uzmanlaşmış bir seviyeye ulaşmasını sağlamaktadır. XP, yaşam döngüsü içerisinde değişikliklerin yapılması konusunda yazılımcıya güven vermektedir (Acar, 2009).

XP içerisinde yer alan ekip üyeleri, müşteri, yazılım geliştirici, test sorumlusu, ölçüm sorumlusu, ekip lideri, danışman ve yöneticidir (Antón, 2004). Küçük ölçekli projelerde bu rollerden bazılarını tek bir kişi üstlenebilmektedir.

Bu yöntemde takım çalışması vurgulanmaktadır. Müşteriler ile birlikte alınan kararlar doğrultusunda işler yürütülmektedir. Yazılım geliştirmek için en uygun yol belirlenmektedir. Ayrıca uygulama sahasına yönelik, müşterinin iş süreçlerini ele alan kullanıcı hikâyeleri (user stories) yazılmaktadır. Kullanıcı hikâyelerini yazmaktaki temel amaç müşteri gereksinimlerini ortaya çıkarmaktır. XP, dört temel değer üzerine kuruludur. Bunlar iletişim, basitlik, geri besleme ve cesarettir. XP'nin temeli bu dört değer içinde yatmaktadır. Bu değerler birlikte ele alındığında XP öğrenimi ve kullanımı kolaylaşmaktadır (Beck, 2002).

İletişim: Proje paydaşları arasındaki iletişimi düzenleyerek, projenin devamını sağlayan XP'nin en temel yapı taşlarından biridir (Beck, 2002).

Basitlik: Basitliği sağlamak için, esnek ve sade bir sistem gerçekleştirmeye çalışır. Böylelikle, projelerin hızlı ve düşük maliyetli gerçekleşmesi sağlanır (Beck, 2002).

Geri Besleme: Programcılar yazdıkları testlerden geri dönüş alarak yazılımın kaliteli olmasını sağlarlar. Bu sayede müşteri ile yazılım ekibi arasında sonradan doğabilecek büyük anlaşmazlıklar daha önceden giderilerek müşteri ile yazılım ekibinin ortak bir noktada buluşması sağlanmış olunur (Beck, 2002).

Cesaret: XP, başarısızlıktan korkmayı değil en kısa zamanda başarısız olmayı ve bu başarısızlığı telafi etmeyi önerir. Bu sayede proje maliyeti düşer ve yazılım gelişim süresi kısalır (Beck, 2002).

Şekil 3-2 XP Pratikleri ve Anahtar Uygulamaları (Beck, 2002)

Şekil 3-2'de gösterilen XP pratikleri ve anahtar uygulamaları 3 kısma ayrılmıştır. Dış kısımda müşteriler ve teknik ekibin bir arada eş güdümlü çalışması sağlanır. Orta kısım ise iş sorunlarına ilişkindir; iş sorunlarına odaklanılmasını sağlar. Teknik sorunların ortadan kaldırılması ile ilgilidir. İç kısımda, yazılımın kalitesini sağlamak amaçlı olarak oluşturulmuştur. Yazılım üretme aşamasında kalitenin düşmemesini sağlar; üretim kısmı ile ilgilidir.

XP'de, projeye küçük ya da büyük bir katkı sağlayan herkes proje içersinde ayrılmaz bir bütün olarak görülür. Müşteri de bu takım çalışmasının bir parçasıdır. Ürün geliştirilme sırasında müşteriler de çalışan ekibin bir parçasıdır. Üretilecek yazılımın ekibi içerisinde bulunurlar (Maurer. F, 2005). Yaşam döngüsünde XP'nin ana ilkesi, yazılımı süreçlere ayırarak işleri alt süreçler ile gerçekleştirmektir. Projenin ilerleme hızının arttırılması amaçlanmaktadır (Beck K, 2002).

3.4 XP Uygulamaları

Çizelge 3-2XP Programlama Uygulamaları (Sarıdoğan, 2004).

XP Uygulamaları
Planlama Oyunu
Ekipte Müşteri (Müşteri Testi)
Önce Test
Basit Tasarım
Eşli Programlama
Sürekli Tümleştirme
Kısa Sürümler (Küçük Sürümler)
Yeniden Yapılandırma
Ortak Sahiplenme(Bütün Takım)
Benzetme
Kodlama Standardı
Haftada 40 Saat Çalışma

Planlama Oyunu: Projenin ön safhasında proje ile ilgili işler çok detaylı olarak tanımlanmaz. Sadece belirlenen sorunlar çözülmeye çalışılır. Bu safhadan sonra kısa süreçler için ayrıntılı proje planı yapılır. Planlama oyunu içerisinde, müşteri tarafından müşteri kartları tanımlanır. Tanımlanan müşteri kartlarına neyin yapılacağı yazılır (Müller, 2004). Ekip lideri tarafından müşteri kartları temel alınarak teknik kartlar oluşturulur. Teknik kartlarda yapılacakların listesi maddeler halinde yazılır ve bu sayede yazılım geliştirimi için ne kadar süreye ihtiyaç olduğu teknik kartlar sayesinde belirlenmiş olur (Beck, 2002).

Ekipte Müşteri: Müşteri temsilcisi yazılım ekibi ile birlikte yazılımı gerçekleştirme sürecinde aynı ortamda bulunur. Bu sayede yazılım gerçekleştirilirken, yazılım ekibinin ihtiyaç duyduğu bilgilere çok kısa sürede erişmesi sağlanmış olur (Acar, 2009).

Önce Test: Kod yazılmadan önce test programı yazılır. Bu sayede mevcut sorunların daha erkenden ortaya çıkması sağlanarak, daha güvenli bir yazılım gerçekleştirilmeye çalışılır (Chan, 2004).

En Basit Tasarım: Müşterilerden alınan gereksinimleri en basit şekilde sağlayan bir tasarım ortaya konur. Müşterinin ileride ihtiyacı olabilecek gereksinimlere fazla dikkat edilmeden, sadece o süre içinde oluşmuş gereksinimlere göre bir tasarım yapılır (Newkirk, 2002).

Eşli Programlama: XP projelerinde iki programcının aynı bilgisayarı kullanarak birlikte kod geliştirmesidir. Bir kişi kodu yazarken, diğer kişi ise kodu izler. Kişiler belli aralıklarla yer değiştirir. Bu pratik ile farklı bilgilere sahip olan programcılar kendilerini geliştirme olanağı bulur (Williams, 2002).

Sürekli Entegrasyon: Geliştirilen her bir modül diğer modüllerle birleştirilir. Modüller bir araya getirilerek modül uyuşmamasından dolayı olabilecek entegrasyon

hataları giderilir. Haftada bir ya da iki kez modül entegrasyonu işi gerçekleştirilir (Bossi, 2003).

Kısa Sürümler: Müşterilere kısa aralıklarla programın sürümü verilir. Bu sayede müşteriye yazılımın ne durumda olduğu gösterilir. XP'nin avantajı diğer metotlara göre daha erken bir süre içerisinde kullanıcıya çalışabilir bir sürüm vermesidir. Bu sayede kullanıcı programın tamamlanan kısımlarına daha erken safhada bakma imkânı kazanır (Beck, 2002).

Yeniden Yapılandırma: Tasarım hataları yazılım sisteminin daha ileride düzeltilemeyecek bir hale dönüşmesine sebep verebilir. Bu yüzden kod ve tasarım sürekli gözden geçirilmelidir (Beck, 2002).

Ortak Sahiplenme: Geliştirilen yazılım kodu, bütün ekip üyelerinin ortak malıdır. Ortak kod sahiplenme uygulamasının temeli bir ekip üyesinin belirli kurallar çerçevesinde diğer ekip üyelerinin ürettiği kodlara erişmesidir. Bu sayede mevcut yazılımı daha iyiye götürme adına değişiklikler yapabilme imkânına sahip olmaları amaçlanmıştır (Bures, 2006).

Benzetme: Gerçekleştirilecek yazılımda sistemler birbirlerine benzetilerek yazılım yapılmaya çalışılır. Örneğin gerçekleştirilecek yazılım parçalara ayrılarak her bir parçası bir başka sistemle benzeştirilir (Kalaycı, 2005).

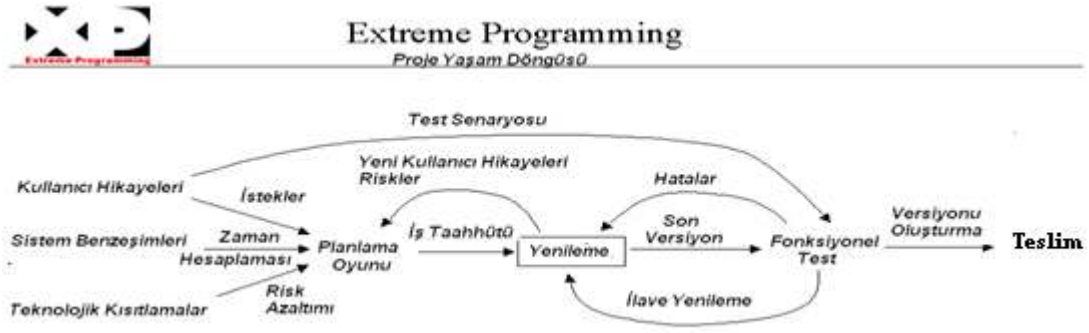
Kodlama Standardı: Farklı kodlama standartları kullanmaktansa tek bir kodlama standardının kullanılmasında yarar vardır. Program içerisindeki her bir nesnenin adı, her bir formun adı, kullanılan değişken isimleri, fonksiyonlarının tutulacakları yerler vb. konuların tek bir standartta olması kodlamanın anlaşılırlığı ve sadeliği açısından önemlidir. Bu sayede kodlama sırasında oluşabilecek karmaşıklıklar da engellenmiş olur (Beck, 2002) .

Haftada Kırk Saat Çalışma: Çalışma verimliliği açısından haftada 40 saatlik bir çalışma süresi ayrılır. Hafta içi her gün 8 saatlik bir çalışma süresi ile işler gerçekleştirilir. Haftada 40 saat uygulamasında fazla mesai kavramı yoktur. Sadece haftada bir kez fazla mesai yapılabilir. Bu pratiğe göre 40 saatten fazla çalışılsa bile verimde yeteri kadar artış olmaz. Hatta yapılan hata sayısında artış meydana gelme riski de artar. Eğer 40 saatlik çalışma süresi yetmiyorsa o zaman sistem iyi bir şekilde düzenlenmiyor demektir. Sistemi yeniden ele alıp, iş akışlarına tekrardan bakılması gerekmektedir (Beck, 2002).

3.5 XP ile Proje Başlangıcı

XP'de projeler kullanıcı senaryolarıyla başlar. Kullanıcı senaryoları ele alınarak parçalara ayrılır. Yapılacak işte kullanıcının hangi bilgileri sisteme gireceği ve hangi bilgilere ulaşacağı tespit edilir. Her bir modül ile ilgili bilgi müşteri kartlarına aktarılır. Müşteri kartlarının asıl amacı geliştirilecek yazılım ile ilgili isterlerin düzenli bir şekilde tutulmasını sağlamaktır. Müşteri kartlarındaki isterler ekip yöneticisi tarafından, daha detaylı bilgileri içeren teknik kartlara dönüştürülür. Bu sayede yazılımcılar üzerlerindeki iş yüklerini ve geliştirilecek modülleri de takip etme imkanına sahip olurlar. Her bir kart basit olmalıdır. Müşterinin isterleri sade bir dil ile anlatılmalıdır. Teknik karttaki ihtiyaçlar karşılanır hale gelince, karttaki işler onaylanır (Beck, 2002).

Proje Çevrimi: Proje yaşam döngüsü, kullanıcı hikayeleri, sistem benzeşmeleri ve teknolojik kısıtlamaların belirlenmesinden sonra başlar.Bu bilgiler ışığında zaman hesaplamaları yapılır. Gelen isterler ve oluşabilecek riskler planlama oyunu içerisinde değerlendirilir. Proje ile ilgili taahhütler belirlenir. Tasarımlar müşteri kartları ve teknik kartlar ile belirlenmeye çalışıldıktan sonra, test ve yazılım kısmına geçilir. Şekil 3-3'de görüldüğü gibi, kod sürekli olarak gözden geçirilip yeni istekler doğrultusunda yeni ilaveler yapılır. İsterlerin karşılanması aşamasından sonra geliştirilen kod fonksiyonel test işlemi aşaması ile test edilir. Bu aşama başarılı bir şekilde geçildikten sonra versiyon oluşturulup geliştirilen yazılım müşteriye teslim edilir (Beck, 2002).



Şekil 3-3 Proje Çevrimi (Beck, 2002)

3.6 XP ile Testlere Öncelik Vermek

Klasik yazılım geliştirme sistemlerinde analiz dokümanları hazırlanır, dokümanların hazırlanma aşaması bittikten sonra kodlamaya geçilir (Meyers, 1992). Kodlama tamamlandıktan sonra da klasik testler yapılır, dönen hatalar veya eksiklikler sebebiyle sürekli testler yapılarak müşterinin istekleri karşılanmaya çalışılır (Poppendieck, 2003).

XP'de ise klasik yöntemlerden farklı olarak bir test kodu yazılmaktadır. Test Kodu, bir modülün doğru çalıştırıldığından emin olmak için programcılar tarafından oluşturulan bir koddur (Felsing, 2002). Test kodunun amacı, çalışan programlar üzerinde yapılan geliştirme çabalarının programın genel yapısına bir yanlışlığa sebep verip vermediğini anlamak için oluşturulur. Bu sayede verilen girdilere uygun çıktı alınıp alınmayacağı da anlaşılmış olur (McGregor,2001). Burada kullanıcılar ana programı kullanırlar. Test programı ise programcılar tarafından kullanılır. Bu kullanım ya belli zamanlarda bu test kodu çalıştırılarak ya da otomatik olarak kodun kendi kendini çağırmasıyla sağlanabilir. Testler iki farklı şekilde olabilir. Sonuca odaklı testler ya da program içerisinde iç bilgilerin üretildiği kısımda oluşturulan testler (Beedle, 2001).

Test programı yazılması için öncelikle ana programa girecek girdiler ve çıktılar belirlenir. Her test kısmı için bir kontrol listesi hazırlanır. Burada ana programa girilen girdilerle test programının girdileri tam olarak aynı girdiler değildir. Ana programa

yapılan girdiler klavyeden yapılan girdiler iken test programına yapılan girdiler, test programı oluşturulmadan ortaya konulan her türlü test senaryosuna uygun verilerdir. Burada test senaryosuna ait verilerin girileceği bir veritabanı kullanmak çok yararlı olacaktır. Aynı şekilde test programına ait çıktılarında bir veritabanında tutulması çok yararlı olacaktır (Felsing, 2002).

Burada çıktılar programın kullanıcıdan beklediği tüm verilerdir. Test programı tarafından oluşturulan test verileri ile ana programdan gelen çıktıların karşılaştırılmasıyla oluşan verilerde test programının kontrol listesidir. Bu listede hatalı olan bilgiler ile hatalı olmayan bilgiler listelenmiştir. Bu sayede hataya sebebiyet verebilecek kod blokları da kontrol edilmiş olunur. Burada en önemli konulardan bir diğeri de girdilerin programa nasıl aktarılacağıdır. Burada programcıların hazırladıkları test senaryoları ile yapılacak girişlerle veriler oluşturulmakta ve oluşturulan çıktılar test için yaratılan yeni bir veritabanına kaydedilmektedir. Test programının sonunda verilerde tutarsızlık veya hata olduğunda ise kontrol listesinin hata kodu olan değer yazılmaktadır. Bu verilere bakarak programcı hangi noktada hata olduğunu anlayabilir. Test programı yazmanın bir diğer avantajı da, hata kodlarını çeşitli kategorilere ayırarak yazılım karmaşıklığı azaltmakta yardımcı olmasıdır. Böylelikle programcı, test programının çalışması sonucunda aldığı verilerde hatalar varsa bu hataların nerelerden kaynaklandığını öğrenebilir. Bu öğrenme işi kategorilere ayrılan hata kodlarından alınan verilerle sağlanır (Jorgensen, 2006). Karmaşıklığı azaltmak için izlenecek bir diğer yol ise ayrı bilgi girişleri için ayrı test kodları yazılmasıdır. Eğer böyle bir yöntem izlenirse karmaşıklık oranı biraz daha azaltılmış olacaktır. Tabi ki böyle bir tercih yapıldığında her bir bölüm için farklı veri grupları veya test gruplarına ihtiyaç vardır. Yardım için hazırlanan fonksiyonlar ve ekrana bilgi listeleme fonksiyonları içinde test veri tabanı hazırlanmış olması gerekir. Aynı şekilde boş bir veri tabanının da çıktılar için oluşturulması gerektir (Chan,2004).

Test veritabanlarında en çok karşılaşılan sorun ise liste alma sırasında görülmüştür. Buna, liste işlemlerinde oluşan detaylarla doğru orantılı olarak karmaşıklığın da artması neden olmaktadır. Bu karmaşıklaşmadan kurtulmak için aynen veri girişlerinde yapıldığı gibi bu işlerin belli kategorilere ayrılarak yapılması çok daha yerinde olacaktır. Bu işler yapıldığında testler sırasında oluşan hatalara ulaşmak daha

kolay olduğu gibi hataların nerelerden kaynaklandığını bulmak da çok daha kolay olmaktadır (Chan,2004).

Test Kodu ve Ana Yazılım: Yapılan ana programda hataların nerelerden kaynaklandığını bulmak uzun bir vakit gerektirir. Bu nedenden dolayı test kodunu doğru bir şekilde yazmak çok önemlidir. Ama test kodunun ana veritabanında kullanmasından ziyade kendi veritabanını kullanması gerekir. Bunun asıl amacı test süresini kısaltmaktır. Eğer yeni veriler üzerinde bir işlem yapılmak isteniyorsa ve bizim test veri tabanımız eski ise bu verilerin test için oluşturulan özel veritabanına girilmesi yeterli olacaktır (Chan,2004).

Program üzerinde bir değişiklik yapıldığında ise programın çalışmasının ardından test programı çalıştırılarak girilen veriler üzerinde yeni programın bir hataya sebep olup olmadığı kontrol edilecektir. Kontrol için kullanılan listeler incelendikten sonra bir hata bulunamadıysa bu yeni değişiklik onaylanır. Bu değişikliği ilgilendiren bir başka veri değişikliği varsa o veri değişikliği veritabanında yapılır. Aynı şekilde test veritabanında bu bilgi değişikliği yapılır. Eğer ihtiyaç duyulursa test programına ilişkin güncellemeler de yapılır. Burada unutulmaması gereken test programının hiçbir zaman statik olmadığıdır. Yeni gelişen durumlar varsa kendi yapısını değiştirmesi gereken yerlerde değiştirir (Andres,2004).

Test Kodu Oluşturulurken Dikkat Edilmesi Gereken Kurallar: Test için gereken verilerin gelişi güzel seçilmesinden ziyade veri seçimi işine dikkat edilip veriler ona göre seçilmelidir. Test için gereken veriler gelişi güzel seçilmemeli, bu işe önem verilip veriler dikkatle seçilmelidir. Test için kullanılacak veriler, başlangıçtaki girdi ve çıktıların neler olduğu, hangi bilgi blokları içinde tutulacağı ve bunların nasıl test edileceğine dikkat edilmelidir. Asıl program değişiklikleri ile paralel olarak test programı değişiklikleri gözden geçirilmelidir (Chan,2004).

Test Kodu Yazmanın Avantajları: Test kodu yazmanın avantajları aşağıda listelenmiştir (Chan,2004).

1. Farklı veriler kullanarak değişik koşulların testi yapılabilmektedir

2. Test kodu yazılırken değişik test senaryoları kullanarak yazılacak ana programın nasıl şekilleneceği ile ilgili bazı bilgiler edinilebilir.
3. Ana programda ortaya çıkacak hataların sayısı azaltılmış olacaktır.
4. Testler istenildiği zaman otomatik yapılarak süreçlerin doğru yürütülüp yürütülmediği anlaşılabilir.
5. Kişisel test senaryolarıyla kaybedilen zaman ve gözden kaçan test senaryoları çok daha iyi bir şekilde uygulanabilir.
6. Ana programdan önce test programına başlandığı için ana programla ilgili eksik kalmış yerler daha önceden fark edilebilir.
7. Yapılan değişikliklerin veriler üzerinde herhangi bir etkisi olup olmadığı saptanabilir.

3.7. XP ile İlgili Yapısal Bilgiler

Yazılım projeleri geliştirilirken çeşitli sorunlarla karşılaşılması kaçınılamazdır. Proje içerisinde karşılaşılabilecek önemli sorunlar ise şunlardır: (Kalaycı, 2005)

- Gecikmeler
- Projenin iptali
- Yama tutmayan sistemler
- Hata oranı
- Yanlış anlamalar
- İş dünyasındaki birçok değişiklikler
- İhtiyaç duyulmayan birçok özellik
- İşten çıkmalar

XP bu sorunlarla mücadele etmek için kendi yaşam döngüsünü ve pratiklerini kullanmaktadır.

XP'nin her bir soruna karşı kullandığı pratikler aşağıdaki gibidir:

- Gecikmeler= Kısa sürüm zamanları.
- Projenin iptal edilmesi = Planlama oyunu ile iş açısından en önemli süreçlere önem verilmesi.

- Yama tutmayan sistemler = Anlaşılır test senaryoları (otomatik test)
- Hata oranları = Testler
- Yanlış anlamalar = Müşterinin takımın ayrılmaz bir üyesi olması sayesinde, yanlış anlamaların büyük bir kısmı önlenir.
- İş dünyasındaki değişiklikler = Kısa sürüm zamanları
- İhtiyaç duyulmayan birçok özellik = Planlama oyunu sayesinde, iş açısından en öncelikli süreçlere önem verilmesi.
- İşten çıkmalar = Çift programcı (Kalaycı, 2005).

XP'deki Roller: XP' de ekip içersinde farklı roller vardır. Kullanılan bu roller ile iş atamaları gerçekleştirilir (Kalaycı, 2005).

- Müşteri
- Yazılım geliştirici
- Test sorumlusu
- Ölçüm sorumlusu
- Koç
- Danışman
- Yönetim

Müşteri: İş tanımını yapar. Neyin yapılacağına ve yapılacak işin ne kadar önemli olduğuna karar verir. Sistemin oluşabilmesi için gerekli olan kabul testlerini tanımlar.

Yazılım geliştirici: İletişim ile görevlidir. Olayların nasıl daha basit bir şekilde gerçekleştirileceği konusu ile ilgilenir. Kodlama, test, sistem entegrasyonu ve müşteri senaryolarının özelliklerinin ve zorluklarının tespiti ile ilgilenir.

Test Sorumlusu: Ekip büyüklüğüne göre tanımlanan bir roldür. Eğer yeterli ekip yoksa ekip içerisinden eşli programlama (pair programming) ile ilgilenen bir kişi bu işi gerçekleştirebilir. Müşterilere test yazma konusunda yardımcı olur. Test araçlarının fonksiyonel olarak doğru çalışıp çalışmadığına bakar. Bütün testlerin düzgün çalışmaları ile sorumludur. Geçen ve geçmeyen testlerin raporlarını da tutar.

Ölçüm Sorumlusu: Bulunan değerlerin bir araya getirilmesinden sorumludur. Gerçekleşen değerler ile önceden tespit edilen değerlerin karşılaştırmasını yapar. Gidişatın tespit edilmesini sağlar.

Koç: Süreç uzmanıdır. Proje içerisinde çıkabilecek sorunların önceden tespit edilmesini sağlamaya çalışır. Sorunlara müdahale zamanlarını belirler.

Danışman: Özel teknik sorunları çözmeye çalışır. Çözümü takım üyeleri ile birlikte bulmaya çalışır.

Yönetici: Müşteri tarafından ya da programcı tarafından verilmesi gereken kararlara karışmaz. Müşteri ile programcıların yazılım içerisinde bir arada uyum içerisinde çalışmalarını sağlayacak şartların oluşmasını sağlar. Toplanacak ölçütlere karar verir. Ödüllendirmeleri yapar. Gereksinim ihtiyaçlarını saptar. Müşteri, proje ile ilgili olarak, genel planı ve maliyetleri bilmelidir. İşlerin ne zaman biteceğini ve bu işlerin sonucunda çıkacak maliyeti ile ilgili müşteri bilgilendirilmelidir. Müşteri ayrıca, yapılacak işleri yeniden tarif edebilme, yaşanan gecikmeler var ise bunlar hakkında mümkün olan en kısa süre içerisinde bilgi sahibi olma, otomatik test programlarının sonuçlarını görme ve çalışan programın bütününü görme hakkına sahiptir. Yazılımcının hakları ise, gereksinimleri bilmek, hangi işlerin önce yapılacağını bilmek ve yardım alabileceği bir ortamda çalışmaktır (Kalaycı, 2005).

XP ne zaman uygulanmaz: XP, her proje ya da her şirket için uygulanabilen bir yazılım geliştirme yöntemi değildir. Şirket yapısı, müşteri kültürü, yönetimsel yapı, çalışma ortamı gibi faktörler XP uygulaması için çok önemlidir. Müşteri yapısı esnek olmayan durumlarda XP uygulanmamalıdır. Proje ekibi büyüklüğünün 10 kişiden fazla olduğu durumlarda ve proje süresi 6 aydan uzun sürecek olan projelerde XP kullanımı uygun değildir (Kalaycı, 2005).

4. BÖLÜM

Yeterince test edilmeyen ve sistem tasarımı yapılmayan yazılımların müşteriye verilmesi sonucu büyük maddi kayıplar ve kaynak israfları yaşanabilmektedir. Bu kapsamda tezin amacına uygun olarak, yazılımın doğasında olan hata unsurunu azaltmayı sağlayabilecek bir hibrit test aracı ve test metodu geliştirilmiştir. Bu bölüm içerisinde, küçük ve orta ölçekli yazılım organizasyonları tarafından çoğunlukla göz ardı edilen yazılım testinin, yazılım yaşam döngüsünün tüm safhalarında nasıl uygulanacağı ele alınmıştır. Çoğunlukla kodlama ve sonrasına bırakılan yazılım testi, test öncelikli yazılım geliştirme yönteminin kullanımı ile XP'ye uygun olarak analiz, tasarım ve gerçekleştirme safhaları üzerinde de uygulanabilmektedir. Özellikle küçük ve orta ölçekli yazılım firmaları, yazılım geliştirme sürecinde, sıklıkla karşılaştıkları yazılım hataları nedeniyle, yazılım projelerinde büyük sorunlarla karşılaşmaktadırlar. Hibrit yazılım geliştirme metodu ile küçük ve orta ölçekli yazılım firmalarının yazılım üretimi ve yazılım geliştirimi süresince karşılaştıkları hatalar en aza indirilmeye çalışılmıştır. Hataların azaltılması için iki farklı test yöntemi olan white box ve black box test yöntemleri yazılım geliştirme süreci içerisinde XP ile birlikte kullanılmıştır. XP'nin yaşam çevrimi ve uygulamaları yazılım projelerinin özellikleri ve büyüklüklerine uygun olarak kullanılmaya çalışılmıştır. Bölüm 4.1'de test öncelikli hibrit yazılım geliştirme metodundan bahsedilecektir. Bölüm 4.2'de ise test öncelikli hibrit yazılım geliştirme metodun etkin bir şekilde kullanılması için geliştirilen yazılım anlatılmaktadır. Bölüm 4.3'de hibrit test aracı ve test metodu kullanarak 3 ayrı yazılım projesi gerçekleştirilmiştir. Bu çalışmada hibrit yazılım geliştirme metodu yardımıyla üç farklı yazılım projesinin geliştirilme süreçlerine ve elde edilen bilgilere yer verilmektedir.

4.1 Test Öncelikli Hibrit Yazılım Geliştirme Metodu

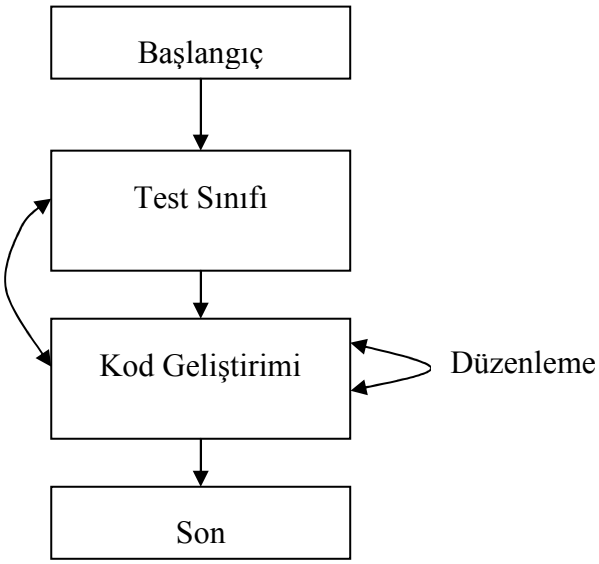
Yazılım projelerinde temel amaç, üretilecek olan yazılımın isterlerinin eksiksiz bir biçimde karşılanmasıdır. Bu amaca ulaşmaktaki en büyük güçlük ise yazılım geliştirme sırasında ve geliştirme sonrasında karşılaşılan hatalardır. Bu kapsamda yazılımın doğasında olan hata unsurunu azaltmayı sağlayabilecek XP metodolojisi kullanılarak geliştirilen yazılımlara uygulanabilecek bir hibrit test aracı geliştirilmiştir.

Küçük ve orta ölçekli yazılım organizasyonları tarafından çoğunlukla göz ardı edilen yazılım testinin, yazılım yaşam döngüsünün tüm safhalarında nasıl uygulanacağı bu başlık altında ele alınmıştır. Çoğunlukla kodlama ve sonrasına bırakılan yazılım testi, test öncelikli yazılım geliştirme yönteminin kullanımı ile XP'ye uygun olarak analiz, tasarım ve gerçekleştirme safhaları üzerinde de uygulanabilmektedir. Bu sayede test işleminin daha aktif kullanılması amaçlanmıştır.

Test öncelikli yazılım geliştirme yönteminin safhaları:

- **Analiz Safhası:** Projelerin analiz safhasında, kara-kutu birim test tekniği (black-box unit testing) kullanılmıştır. Bu test tekniği kullanılarak, şartnamedeki isterlerin program içerisinde ne şekilde karşılandığı tespit edilmiş ve hata maliyetleri azaltılmaya çalışılmıştır. Geliştirilecek programların sınırları ve kısıtları belirlenmiştir. Ayrıca, teknolojik alt yapının ne şekilde oluşturulacağı da tespit edilmiştir. XP'nin planlama oyunu uygulaması kullanılarak, proje ekibi ile müşterinin geliştirilecek olan proje üzerinde bir arada çalışması sağlanmıştır. Müşteri kartlarından elde edilen bilgiler, daha önceden geliştirilen hibrit test temelli programa aktarılmıştır. Elde edilen analiz bilgileri sonucunda adam/saat hesaplaması yapılmıştır.
- **Tasarım Safhası:** Bu safhada, planlama oyunu sonucunda oluşan müşteri kartlarındaki bilgiler detaylandırılarak incelenmiştir. Veritabanı yapısı ve geliştirilecek projenin ana ve yan süreçleri tanımlanmıştır. Bu tanımlamalardan sonra elde edilen bilgiler, hibrit test temelli program içersinde yer alan teknik kartlar kısmına aktarılmıştır. Programa girilen bu bilgiler ile proje bitiş tarihi revize edilmiştir.
- **Kodlama Safhası**: Bu safha boyunca, beyaz-kutu birim test (white-box unit testing) tekniği kullanılmıştır. XP'nin önce test uygulamasına uygun olarak, öncelikle temel test kodu, daha sonra ise işlevsel kod yazılmıştır. Test başarılı olduktan sonra yeni bir test yazılarak işlevsel koda uygulanır. İşlevsel kod, bu testlerden başarılı olacak şekilde yeniden düzenlenir. Bu adımlar yazılımcı tarafından belirlenen bütün testler başarılı oluncaya kadar devam eder. Bu sayede, öncelikle test kodları yazılmaya başlandığı için, geliştirilecek yazılımın ne şekilde olması gerektiği büyük oranda belirlenmiş olur. Test kodu yazmanın

en büyük avantajı, yazılım gelişimi ile birlikte onu test edecek bir kodun da hazır olmasıdır. Şekil 4-1'te kodlama işinin test yöntemi ile nasıl gerçekleştirildiği anlatılmaktadır.



Şekil 4-1 Test Öncellikli Yazılım Akış Şeması

- **Test Safhası:** Bu safha, proje içerisinde gerçekleştirilmesi gereken doğrulama ve kabul testlerinin yapılmasını içerir. Kabul testinin ardından, bakım safhasına geçilir.
- **Bakım Safhası:** Bu aşamadan sonra müşteri tarafından yeni gelen istekler analiz, tasarım, kodlama ve test aşamalarından geçecek şekilde tekrar değerlendirilir.

4.2. Hibrit Test Aracı

Geliştirilen uygulama yazılımı, XP'ye ve kullanılan test temelli yapıya uygun olarak geliştirilmiştir. Modüllerde kullanılan parametreler ve yönetim ekranlarından oluşan uygulamaya ait bilgiler aşağıda açıklanmıştır. Uygulamanın temel amacı geliştirilen diğer programlardaki isterler ve testlerin takibini sağlamaktır. Bu yazılım oluşturulan hibrit test yönteminin aktif olarak projelerde kullanılması için

geliştirilmiştir. Geliştirilen üç farklı yazılım projesinin test temelli olarak yönetilmesi amacı ile geliştirilmiştir.

Uygulama temel modülleri şunlardır:

- Sisteme Giriş Modülü,
- Proje Yönetim Modülü
- Müşteri Kartları Yönetim Modülü
- Teknik Kartlar Yönetim Modülü
- Raporlama Modülü'nden oluşmaktadır.

Yazılım Geliştirme Araçları: Yazılım geliştirme faaliyetleri sırasında Çizelge 4-1'de gösterilen yazılım geliştirme araçları kullanılmıştır.

Çizelge 4-1 Yazılım Geliştirme Araçları

Araç	İşlevi
Visual Studio (C#.net)	Uygulama geliştirmede programlama aracı olarak Visual Studio içerisinde yer alan C# programlama dili kullanılmıştır.
MS SQL Server	Veri tabanı olarak Microsoft SQL Server kullanılmıştır.
PhotoShop CS3	Ekran ve banner'ların tasarımına görsellik kazandırmak amacıyla mevcut araca ek olarak bu yazılım aracından istifade edilmiştir.

Geliştirilen hibrit test yazılımı; test öncelikli yazılım geliştirme metodun etkin bir şekilde kullanılmasına yardımcı olması için geliştirilmiştir. Aşağıda geliştirilen bu yazılımın modülleri ile ilgili bilgiler verilmektedir.

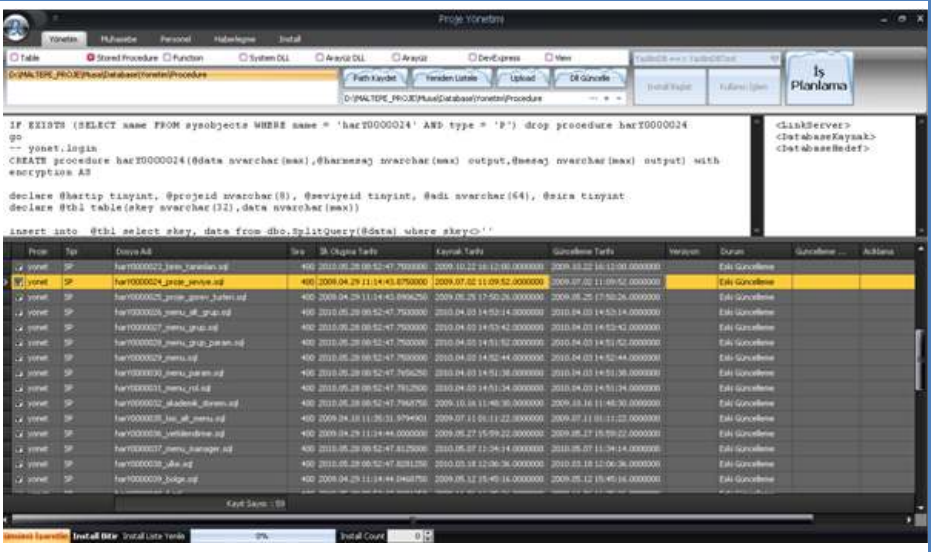
Sisteme Giriş Modülü: Bu modül, yazılımı kullanacak tüm kullanıcıların yetkileri doğrultusunda sisteme girişini ve ilgili ekrana yönlendirilmesini sağlar. Şekil 4-2'de gerçekleştirilen hibrit test uygulama yazılımın giriş ekranı sunulmuştur. Uygulamaya giriş yapmak isteyen kullanıcı, 'Kullanıcı Adı' ve 'Şifre' alanlarını doldurduktan sonra 'Giriş' düğmesine basar. Program girilen 'kullanıcı adı' ve şifre bilgisini veri tabanı üzerinden kontrol eder. Doğrulama sağlanıyorsa kullanıcının profil bilgisine göre erişim izni verir; aksi durumda uygulama girişine izin verilmez. Kullanıcı 'Sisteme giriş izniniz yok!' şeklinde bir mesajla uyarılır.

	Sisteme Giriş Ekranı
Kullanıcı	Tüm kullanıcılar
Ekran Görüntüsü	
Kullanılan Nesneler	Panel, Table, Label, Textbox, Button
VT Tabloları	Users

Şekil 4-2 Uygulama Yazılımı Sisteme Giriş Modülü Ekran Görüntüsü

Projeler Yönetim Modülü: Şekil 4-3'da gerçekleştirilen hibrit test uygulama yazılımının proje yönetim modülüne ait ekran görüntüsü gözükmektedir. Kullanıcı profil ve parametreleri bu modül üzerinden tanımlanır. Ayrıca, bu modül ile uygulamaya ilişkin temel tanımlamalar da yapılabilir. Sadece "yönetici" yetkisine sahip kullanıcı tarafından bu modül kullanılabilir. Geliştirilecek projelerle ilgili temel

tanımlamaların yapıldığı modüldür. Bu modülde proje seçimi yapıldıktan sonra, seçilen proje ile ilgili test ekranları listelenebilir.

Projeler Yönetim Modülü	
Kullanıcı	Admin
Ekran Görüntüsü	
Kullanılan Nesneler	Panel, Table, Label, Textbox, Button
VT Tabloları	Projeler

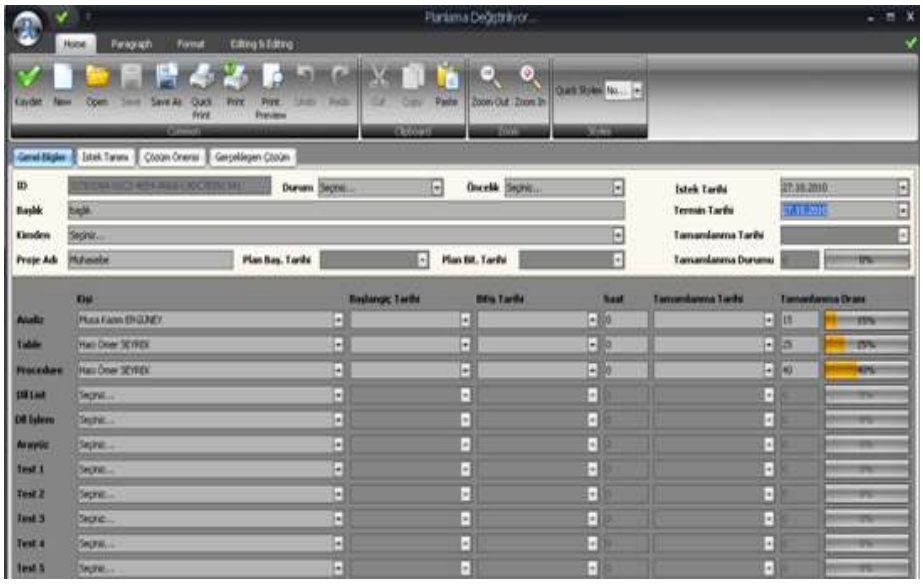
Şekil 4-3 Uygulama Yazılımı Projeler Yönetim Modülü Ekran Görüntüsü

Müşteri Kartları Yönetim Modülü: Şekil 4-4'da gerçekleştirilen hibrit test uygulama yazılımının müşteri kartları yönetim modülü ekran görüntüsü gözükmektedir. Müşterilerden alınan bilgilerin işlendiği müşteri kartları yönetim modülü yardımıyla geliştirilecek projeye ait istek ve talepler takip edilebilmektedir.

Müşteri Kartları Yönetim Modülü	
Kullanıcı	Admin, Proje Yöneticisi, Müşteri
Ekran Görüntüsü	
Kullanılan Nesneler	Panel, Table, Label, Textbox, Button
VT Tabloları	TeknikKartlar, BlackBox, MusteriKartlari, Projeler

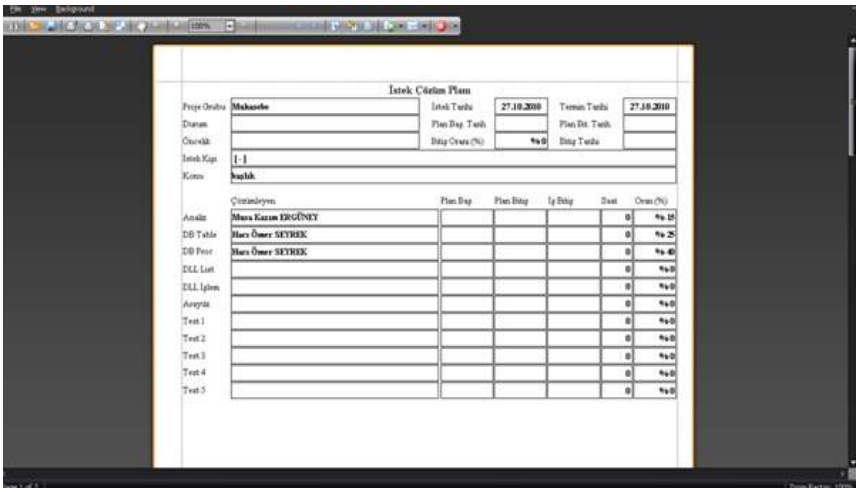
Şekil 4-4 Uygulama Yazılımı Müşteri Kartları Yönetim Modülü Ekran Görüntüsü

Teknik Kartlar Yönetim Modülü: Şekil 4-5'de gerçekleştirilen hibrit test uygulama yazılımının teknik kartlar yönetim modülün ekran görüntüsü sunulmuştur. Yazılımcı, gerçekleştireceği yazılım parçası ile ilgili bilgileri bu modül üzerinde görmektedir. Yazılımcılar üzerlerindeki iş yüküne ilişkin bilgileri de bu modül üzerinden takip edebilirler. Ayrıca, gerçekleştirilen testlere ait bilgilere de bu modül üzerinden ulaşılabilmektedir.

	Teknik Kartlar Yönetim Modülü
Kullanıcı	Admin, ProjeYöneticisi, Yazılımcı
Ekran Görüntüsü	
Kullanılan Nesneler	Panel, Table, Label, Textbox, Button
VT Tabloları	TeknikKartlar, BlackBox, MusteriKartlari, Projeler, WhiteBox

Şekil 4-5 Uygulama Yazılımı Teknik Kartları Yönetim Modülü Ekran Görüntüsü

Raporlama: Şekil 4-6'da gerçekleştirilen hibrit test uygulama yazılımının, raporlama modülüne ait ekran görüntüsü sunulmuştur. Bu modül, müşteri kartları ve teknik kartlara ilişkin girilen bilgilerin raporlanmasını sağlamaktadır.

Raporlama	
Kullanıcı	Admin, ProjeYöneticisi, Müşteri, Yazılımcı
Ekran Görüntüsü	
Kullanılan Nesneler	Panel, Table, Label, Textbox, Button
VT Tabloları	BlackBox, TeknikKartlar, MusteriKartlari, Projeler

Şekil 4-6 Uygulama Yazılımı Raporlama Modülü Ekran Görüntüsü

4.3. Geliştirilen Yazılım Projeleri

XP metodolojisinin ve önerilen test metodunun uygulandığı 3 adet kapsamlı yazılım projesi gerçekleştirilmiştir. Geliştirilen projelerin tümü Eylül 2010 itibari ile faal olarak kullanılmaktadır. Bu yazılım projeleri Extreme Programming'in yaşam döngüsüne ve çalışma ortamına büyük oranda bağlı kalınarak geliştirilmiştir. Ayrıca bu yazılım projelerinin test kısmı için geliştirilen uygun hibrit test metodu ve test aracı kullanılmıştır. Projeler ile ilgili detaylar aşağıda verilmiştir.

Yazılım projeleri ile ilgili genel tanımlar: Geliştirilen yazılımlar ile ilgili olarak aşağıdaki başlıklar ile yazılımlar hakkında bilgi verilmiştir.

- **Yazılım Projesi**
  - **Yazılımın Adı:** Gerçekleştirilen yazılım projesinin adının yer aldığı kısım.

- **Amaç:** Yazılımın temel amacının tanımlandığı kısım.

- **Proje Geliştirme Süreci:** Yazılım projesinin hangi aşamalardan geçerek oluşturulduğunun anlatıldığı kısım.
  - **Proje Planlama:** Yazılım Planlama safhalarının anlatıldığı kısım.
  - **Ana ve Yan Süreçlerin Tespit Edilmesi:** Proje İle ilgili temel ve temel süreçlere bağlantılı kısımların ortaya çıkarıldığı kısım.
  - **Yazılımın Kodlanması:** Yazılımın kodlanması aşamasında dikkat edilecek kısımların anlatıldığı kısım.
  - **Yazılımın Testi:** Yazılım projesi içerisinde testin nasıl gerçekleştirildiğini anlatılan kısım.

- **Bilgilendirme Tablosu:** Yazılım ile ilgili özelliklerin tanımlandığı tablo. Yazılım ekibi büyüklüğü, tecrübe, eğitim seviyesi, kullanılan programlama dili, kullanılan araçlar, yazılım ile ilgili ölçümler.

- **Yazılım ölçümleri**: Bu kısımda yazılım projelerinden elde edilen yazılım ölçümlerine yer verilmiştir. Müşteri kartları ve teknik kart sayısı, satır sayısı, yazılım projesi içerisinde çalışan personelin sayısı, kodun açıklama oranı(comment), karmaşıklık oranları
  - Müşteri Kartları: Yapılan analizden elde edilen bilgilerin listelendiği kartlardır.
  - Teknik Kartlar: Planlama oyunu sonucunda oluşan müşteri kartlarındaki bilgilerin, ekip lideri tarafından detaylandırılarak listelendiği kartlardır.
  - Açıklama Oranı: Kod açıklama oranı.
  - Karmaşıklık Oranı: Kodun kalitesini ölçmekte kullanılan en önemli metriktir. Kodun ne kadar kompleks olduğunu hakkında fikir veren en önemli ölçümlerden biridir. Karmaşıklık oranını hesap etmek için; akış diyagramlarına bakar, program içerisindeki

noktaları, yolları ve yol ayrımlarını hesap eder. 1-9+ arasında değer alır. 5 ve üzerindeki değerler yazılımın yüksek karmaşıklık değerine sahip olduğunu göstermektedir.

Kullanılan Ölçüm Programı: Geliştirilen yazılım projelerinin ölçümü için açık kaynak kodlu Source Monitor isimli program kullanılmıştır. Yapılan bütün hesaplamalarda bu program kullanılmıştır. Gerçekleştirilen projeler ile ilgili detaylar aşağıda verilmiştir. Projelerin ana modüllerinden ve geliştirilme süreçlerinden bahsedilmiş, projeler ile ilgili detaylı bilgilendirme tablosu ve geliştirilen projelerinin ölçüm bilgilerine yer verilmiştir.

Yazılım projesi-1 personel otomasyon sitemi: Maltepe Üniversite personel otomasyon sisteminin, personel ve raporlama modüllerinin geliştirilmesi. Yeni üniversite personel otomasyon sisteminin, personel modülü, demirbaş takip modülü, raporlama modülü, sistem uyarıları modülü, genel ayarlar modülü ve bu modüllere bağlı alt modüllerden oluşması. Bu amaç doğrultusunda projenin bitiş süresi göz önünde bulundurularak, 3 kişilik bir yazılım ekibi oluşturulmuştur. Projenin gereksinimleri göz önünde bulundurularak 6 aylık bir sürede geliştirmek üzere, XP uygulamalarında deneyimli bir ekip lideri ve bu konularda deneyimi olmayan iki yazılımcı ile proje gelişim süreci başlatılmıştır. Proje hibrit test metodu ve yazılımı ile geliştirilmiştir.Yazılım projesi içinde yer alan ana modüller ile ilgili bilgiler aşağıda verilmiştir.

- **Personel Modülü:** Bu modül personel kaydı, personel arama ve personel bilgilerini görüntüleme/güncelleme işlevlerini yerine getirmektedir.
- **Demirbaş Takip Modülü:** Bu modül Maltepe üniversitesi içerisinde personel üzerine zimmet edilen demirbaşların takibi ile ilgili işlevleri yerine getirmektedir.
- **Raporlama Modülü:** Bu modül sistemde tutulan personel verilerine göre ihtiyaç duyulan raporlama işlevlerini yerine getirmektedir. Bu modül, yapısı gereği esnek bir raporlama olanağı sunmaktadır.
- **Sistem Uyarıları Modülü:** Bu modül Maltepe üniversitenin idari ve akademik işleyişi ile ilgili planlanan işlerin personel tarafından takibini sağlamaktadır.

- **Genel Ayarlar Modülü:** Bu modül sistemin temelinde ihtiyaç duyulan ve zamanla değişebilecek ayarların kullanıcı tarafından kolaylıkla yapılmasını sağlamaktadır.

Yazılım projesi geliştirme süreci: Bu bölümde proje planlama, mevcut gereksinimlerin ortaya çıkartılması, ana ve yan süreçlerin tespit edilmesi, yazılımın kodlanması ve yazılımın testi olmak üzere toplamda beş adımda gerçekleşen proje geliştirme süreci anlatılmaktadır.

Proje Planlama: XP'nin yapısına uygun olarak, proje geliştirme süreci öncelikle planlama oyunuyla başlamıştır. Planlama oyunu ile birlikte projenin ayrıntılarından uzak temel bir plan yapılmıştır. Planlama oyunu sayesinde Maltepe üniversite personel otomasyon sisteminin gelişiminde kullanılacak süreçler belirlenmiştir. Sonrasında projenin temelini oluşturan beş ana modül ve bu modüllere bağlı alt modülleri ele alan toplam 62 adet müşteri kartı tanımlanmıştır. Planlama oyunu uygulamasın yapısına aykırı olarak müşteri kartları, müşteriden alınan bilgiler doğrultusunda ekip lideri tarafından Microsoft Excel üzerinde oluşturulmuştur. Sonrasında, ekip lideri tarafından bu müşteri kartlarında yer alan bilgiler teknik kartlara aktarılmıştır. Maltepe üniversitesi personel otomasyon sisteminin geliştirilmesi süresince, benzetme uygulaması kullanılmamıştır.

Ana ve Yan Süreçlerin Tespit Edilmesi: Proje içerisinde öncelikli olan yazılım süreçlerine bakılarak, teknik kartların hangi sırada kodlanacağı belirlenmiştir. XP'nin yaşam döngüsü kullanılarak, proje geliştirilirken ortaya çıkan yeni gereksinimler karşılanmaya çalışılmıştır.

Yazılımın Kodlanması: Bu süreçte, proje planı içerisinde belirlenen ve daha önceki proje deneyimlerinden yararlanılarak oluşturulan kodlama standartlarına uygun olarak, teknik kartlardaki gereksinimler program kodlarına dönüştürülmüştür. Basit tasarım uygulaması kullanılarak, kısa sürede anlaşılması, yönetilmesi ve değiştirilebilmesi kolay bir yazılım geliştirilmesi amaçlanmıştır. Müşterinin değişen gereksinimleri göz önünde bulundurularak, ekip içinde mümkün olduğunca basit tasarıma önem verilmiştir. Ayrıca, kodlama sırasında önce test, haftada 40 saat ve eşli programlama uygulamaları kullanılmıştır. Eşli programlama uygulaması sayesinde,

programcılar kendilerini geliştirme imkânı bulmuştur. Kodlama işlemi sürecinde, yapılan sistem değişiklikleri ve yeni bileşenler hemen sisteme entegre edilerek sürekli günlük entegrasyonlarla üç haftada bir yeni sürümlerin oluşturulması sağlanmıştır. Ekipte müşteri uygulaması, müşterinin personel yetersizliğinden dolayı bu süreç içersinde tam olarak uygulanamamıştır.

Yazılımın Testi: Otomasyon sisteminin testi üç aşamada gerçekleştirilmiştir. Birinci aşamada ortak sahiplenme uygulamasına göre geliştirilen kodlara programcılar ve ekip lideri tarafından kara-kutu (black-box unit testing) birim test tekniği, ikinci aşamada ise, beyaz-kutu birim testleri (white-box unit testing) uygulanmıştır. Ayrıca; müşterinin iş süreçlerinin belirlenmesi aşamasında oluşturulan kullanıcı hikâyelerine uygun olarak, yazılım müşteriye test ettirilmiştir. Bu test işlemi sonunda müşteriden gelen geri bildirimlere göre yazılım ekibi tarafından yazılım içerisindeki hatalar giderilmiş ve yeni sürümler ortaya çıkarılmıştır. Testin son aşaması kabul testleridir. Kabul testleri, yazılımı kullanacak personel tarafından gerçek veriler kullanılarak yapılmıştır. Kabul testlerinin müşterinin gereksinimlerini karşılaması ile birlikte yazılımın teslim süreci tamamlanmıştır. Şekil 4-7'de gerçekleştirilen personel otomasyon sistemine ait ekran görüntüsü görülmektedir.

Şekil 4-7 Personel Otomasyon Sistemi Ekran Görüntüsü

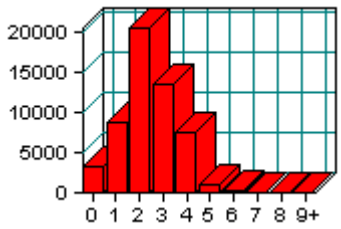
Bilgilendirme Tablosu: Çizelge 4-2'de gerçekleştirilen personel otomasyon sistemine ait bilgilendirme tablosu verilmektedir. Geliştirilen yazılım bu gün itibari ile Maltepe üniversitenin kadrolu (akademik, idari) tüm personeli ve görevlendirme statüsü ile gelip Maltepe üniversitesi içerisinde hizmet veren tüm personelin bilgilerini içermektedir. Toplamda 1000'in üzerinde aktif personelin bilgisini içeren bu yazılım aynı zamanda sistem üzerinden gerekli raporlama işlemlerini de gerçekleştirilebilmektedir.

Çizelge 4-2 Personel Programı Bilgilendirme Tablosu

	Personel Yazılım Projesi
Yazılım Ekibi Büyüklüğü:	3 Yazılımcı + 1 Müşteri
Eğitim Seviyesi:	2 Yüksek Lisans 2 Lisans Mezunu
Yazılım Tecrübe Bilgileri:	< 5 yıl: 2 kişi 5-10 yıl: 2 kişi
Uzmanlık Bilgileri:	Yüksek
Programlama Dili Bilgisi:	Yüksek
Oluşturulan Müşteri Kart Sayısı:	62
Domain:	Windows Application

Ölçümler: Gerçekleştirilen personel otomasyon sisteminin ölçümü için Source Monitor isimli program kullanılmıştır. Çizelge 4-3'de görüntülenen tüm hesaplamalar bu program tarafından elde edilmiştir. Projenin lines (satır) büyüklüğü, komut sayısı, kodun açıklama oranı (comment), ortalama karmaşıklık oranları, programın içinde bulunan dosya sayısı, kodların karmaşıklık oranlarına ilişkin bilgiler Çizelge 4-3'de verilmiştir.

Çizelge 4-3 Personel Otomasyon Sistemi-Ölçümler

Personel Otomasyon Sistemi	
Satır Sayısı	62.815
Toplam Komut Sayısı	54.307
Açıklama Oranı	%10.6
Ortalama Karmaşıklık	2.31
Dosya Sayısı	131
Veri Oluşturma Tarihi	26 Ekim 2010
Karmaşıklık Oranları	**Komut Sayısı**
0	3,257
1	8,755
2	20,370
3	13,312
4	7,392
5	871
6	186
7	114
8	36
9+	14
Karmaşıklık Oranları Grafiği	

26 Ekim 2010 tarihinde personel otomasyon sistemine ilişkin Satır sayısı. toplam komut sayısı, açıklama sayısı ve toplam karmaşıklık sayısı değerlerin elde edilmesinde Source monitör programı kullanılmıştır. Elde edilen sonuçlar Çizelge 4-3'de sunulmuştur. Yazılım için 54.307 satırlık bir komut sayısının elde edildiği görülmüştür. Yazılımın başlanıldığı andan 26 Ekim 2010 tarihi itibariyle toplamda 817 hata ortaya çıkarılmış ve bulunan bu hatalar hibrit test yazılımı içerisindeki ilişkisel veri tabanında tutulmuştur. Gerçekleştirilen kodların ortalama karmaşıklık oranları 2.31 olarak bulunmuştur. Gerçekleştirilen kodların sadece % 2,24'lik bir kısmının 5 ve üzeri karmaşıklık değerine sahip olduğu görülmektedir. Bu elde edilen oranlar yazılımın yönetilebilirlik açısından iyi bir konumda olduğunu göstermektedir. Yazılım karmaşıklığı yazılım kalitesini

etkileyen bir unsurdur. Bu nedenle bulunan oranlarda 5 ve daha üstü karmaşıklığın az olması yazılımların yönetimi ve yazılım kalitesi açısından da önemlidir. Yazılımda karmaşıklık genelde, sistemi oluşturan bileşenlerin birbirleri ile ilişkileri ya da bileşenlerin kendi içerisinde çağırdıkları diğer alt bileşenlerin sayısı ile hesaplanır. Proje sonucunda elde edilen 2.31 oranı yazılım karmaşıklığına neden olan gereksinimleri, kullanılan teknolojileri ve müşteri ilişkilerin etkin bir şekilde kullanıldığının da önemli bir göstergesidir. Proje içerisinde kullanılan hibrit test tekniği ile yazılım geliştirimi süresince karşılaşılan hatalar en aza indirilmeye çalışılmış hataların azaltılması için iki farklı test yöntemi olan white box ve black box test yöntemleri yazılım geliştirme süreci içerisinde XP ile birlikte kullanılmıştır. İzleyen bölümde gerçekleştirilen bir başka proje olan optik okuyucu ve değerlendirme yazılımı açıklanmaktadır. Projelerin ana modüllerinden ve geliştirilme süreçlerinden bahsedilmiştir. Ayrıca projeler ile ilgili detaylı bilgilendirme tablosu ve geliştirilen projelerinin ölçüm bilgilerine de yer verilmiştir.

Yazılım projesi-2 optik okuyucu ve değerlendirme: Yazılım projesinin amacı; Maltepe üniversitesinde çoktan seçmeli soru sistemlerinde kullanılan optik okuyucu ve değerlendirme yazılımının oluşturulmasıdır. Bu kapsamda üniversitede içinde gerçekleştirilen Bilgisayara Giriş I-II muafiyet sınavları ve sene içi ara sınav ve final sınavları, uzaktan eğitim final sınavları, E-MYO final sınavları, yabancı dil sınavları, vb sınavların değerlendirilmesinde kullanılacak optik okuyucu ve değerlendirme yazılımın gerçekleştirmiştir. Proje geliştirme yöntemi olarak hibrit test metodu ve yazılımı kullanılmıştır. Yazılım projesinin geliştirme aşamaları aşağıda verilmiştir.

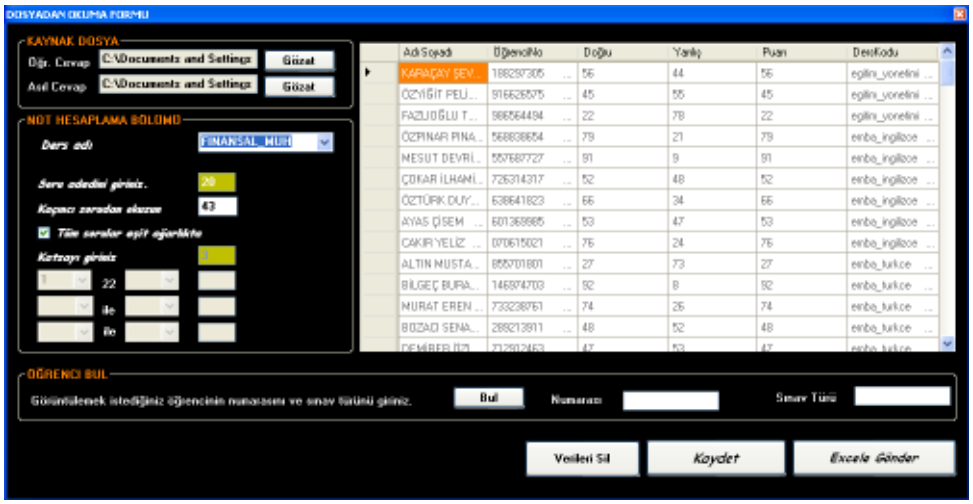
Proje Planlama: Geliştirilecek yazılımın kapsamı belirlendikten sonra, sistem içerisinde olması gereken işlevsel gereksinimler tespit edilmeye çalışılmıştır. Bu aşamada, çoktan seçmeli sorulara ilişkin soru ağırlıkları ve adetleri ile kitapçık türleri belirlenmiştir. Optik okuyucunun algılamasına yönelik form tasarımı hazırlanmıştır. Bunun yanında, optik okuyucunun bağlantı portları, veri tipleri ve veritabanı şeması tanımlanmıştır. Proje ile ilgili kapsam belirleme çalışması, XP'nin planlama oyunu uyuglamasına uygun olarak gerçekleştirilmiştir. Bununla birlikte yazılımın geliştirilmesi aşamasında isterler belirlenmeye çalışılmıştır. Ortaya çıkan isterler, ekip lideri tarafından incelenerek, müşteri kartlarına ve teknik kartlara aktarılmıştır.

Ana ve Yan Süreçlerin Tespit Edilmesi: Optik okuyucu değerlendirme yazılımı için ihtiyaç duyulan 8 temel süreç ve bu süreçlerle bağlantılı 24 alt süreç tespit edilmiştir. Süreçler ile alt süreçler arasındaki bağlantılar ve oluşturulma zamanları tanımlanmıştır. Tanımlanan bu bilgiler, kullanılan hibrit test aracına aktarılmıştır.

Yazılımın Kodlanması: Bu süreçte, Maltepe üniversitede kullanılan kodlama standartlarına uygun olarak, yazılım geliştirme çalışmalarına başlanmıştır. Müşteri kartlarındaki veriler dikkate alınarak, oluşturulan teknik kartlardaki gereksinimler, yazılımcılar tarafından program kodlarına dönüştürülmüştür. Yazılım projesinde, XP'nin basit tasarım uygulaması kullanılmıştır. Bu sayede projenin kısa sürede anlaşılması, yönetilmesi, değiştirilebilmesi ve kolay bir yazılım geliştirilmesi amaçlanmıştır. Müşterinin değişen gereksinimleri temel alınarak, ekip içinde mümkün olduğunca basit tasarıma ve geri dönüşlere önem verilmiştir. Ayrıca, projenin her aşamasında test öncelikli yazılım geliştirme yöntemi kullanılarak kod geliştirilmiştir. Kodlama işlemi sürecinde, yapılan sistem değişiklikleri ve yeni bileşenler hemen sisteme entegre edilmiştir. Böylece ortaya çıkabilecek hataların erken safhalarda tespit edilmesi sağlanmıştır. XP'nin sürekli tümleştirme uygulaması kullanılarak, üç hafta da bir yeni sürümler oluşturulmuştur.

Yazılımın Testi: Geliştirilen yazılımın testi, analiz aşamasından başlanılarak her bir safhada ayrı ayrı ele alınmıştır. Yazılımın testi üç aşamada gerçekleştirilmiştir. Projenin ilk aşamasında kara-kutu birim test tekniği (black-box unit testing) kullanılmıştır. İkinci aşamada XP'nin ortak sahiplenme uygulamasına uygun olarak, programcılar ve ekip lideri tarafından geliştirilen kodlar beyaz-kutu birim testlerine (white-box unit testing) tabi tutulmuştur. Ayrıca, XP'nin ekipte müşteri uygulamasına uygun olarak yazılım müşteriye test ettirilmiştir. Müşteriden gelen geri bildirimlere göre, yazılım ekibi tarafından hatalar giderilmiş ve yeni sürümler ortaya çıkarılmıştır. Testin son aşaması ise doğrulama ve kabul testleridir. Kabul testleri, müşteri tarafından gerçek veriler kullanılarak yapılmıştır. Müşteri onayının alınması ile birlikte yazılımın teslim süreci tamamlanmıştır. Test aşamasında yapılan işlemlerin takibi için, daha önceden geliştirilen hibrit test yazılımı kullanılmıştır. Yukarıda anlatılan test yöntemlerinin haricinde, test aşamasında adım adım izleme (walkthrough) yöntemi de

kullanılmıştır. Bu sayede kodu geliştiren yazılımcının gözünden kaçan hataların diğer yazılımcılar tarafından kolaylıkla tespit edilmesi ve düzeltilmesi sağlanmıştır. Şekil 4-8'de gerçekleştirilen optik okuyucu değerlendirme sistemine ait bir ekran görüntüsü görülmektedir.



Şekil 4-8 Optik Okuyucu Değerlendirme Sistemi Ekran Görüntüsü

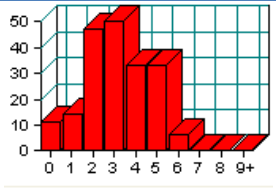
Bilgilendirme Tablosu: Çizelge 4-4'de gerçekleştirilen optik okuyucu programı bilgilendirme tablosu verilmektedir. Geliştirilen yazılım Ekim 2010 itibari ile Maltepe üniversitede çoktan seçmeli soru sistemine göre gerçekleştirilen tüm yıl içi sınavlarda kullanılmaktadır. Bu gün itibari ile 50000'in üzerinde optik kâğıt sistem üzerinden okunarak gerekli notlandırma işlemleri gerçekleştirilmiştir.

Çizelge 4-4 Optik Okuyucu Yazılım Projesi Bilgilendirme Tablosu

Optik Okuyucu Yazılım Projesi	
Yazılım Ekibi Büyüklüğü:	2 Yazılımcı
Eğitim Seviyesi:	1 Yüksek Lisans 1 Ön Lisans
Yazılım Tecrübe Bilgileri:	5-10 yıl: 1 kişi 0-5 yıl:1 kişi
Uzmanlık Bilgileri:	Yüksek
Programlama Dili Bilgisi:	Yüksek
Oluşturulan Müşteri Kart Sayısı:	2
Domain:	Windows Application, C#, SQL Server

Ölçümler: Gerçekleştirilen optik okuyucu sistemin ölçümü için Source Monitor isimli program kullanılmıştır. Çizelge 4-5'de görüntülenen tüm hesaplamalar bu program tarafından elde edilmiştir. Projenin satır (lines) büyüklüğü, komut sayısı, comment oranları, ortalama karmaşıklık oranları, programın içerisinde bulunan dosya sayısı ve subroutines sayıları ve oluşturulan kodların karmaşıklık oranları Çizelge 4-5'de verilmiştir.

Çizelge 4-5 Optik Okuyucu Sistemi Ölçümler

Optik Okuyucu Yazılım Projesi	
Satır Sayısı	334
Toplam Komut Sayısı	194
Açıklama Oranı	%9
Ortalama Karmaşıklık	3.05
Dosya Sayısı	6
Veri Oluşturma Tarihi	26 Ekim 2010
Karmaşıklık Oranları	**Komut Sayısı**
0	11
1	14
2	47
3	50
4	33
5	33
6	6
7	0
8	0
9+	0
Karmaşıklık Oranları Grafiği:	

26 Ekim 2010 tarihinde optik okuyucu sistemine ilişkin satır sayısı. toplam komut sayısı, açıklama sayısı ve toplam karmaşıklık sayısı değerlerin elde edilmesinde Source monitör programı kullanılmıştır. Elde edilen sonuçlar Çizelge 4-5'de sunulmuştur. Çizelge 4-5'de görüldüğü üzere karmaşıklık oranları 7,8 ve 9+ olan değerler bulunmamaktadır. Yazılımın başlanıldığı andan 26 Ekim 2010 tarihi itibariyle toplamda 3 hata ortaya çıkarılmış ve bulunan bu hatalar hibrit test yazılımı içerisindeki ilişkisel veri tabanında tutulmuştur. Gerçekleştirilen yazılımın karmaşıklık oranlarının düşük olması yazılımların yönetilebilirlikleri açısından önemlidir. Yazılım geliştirimi süresince karşılaşılan hatalar en aza indirilmeye çalışılmış hataların azaltılması için iki farklı test yöntemi olan white box ve black box test yöntemleri yazılım geliştirme süreci içerisinde XP ile birlikte kullanılmıştır. XP'nin yaşam çevrimi ve uygulamaları yazılım projelerinin özellikleri ve büyüklüklerine uygun olarak kullanılmaya çalışılmıştır. Yazılımın başlanıldığı andan 26 Ekim 2010 tarihi itibari ile toplamda 3 hata ortaya çıkarılmış ve bulunan bu hatalar hibrit test yazılımı içerisindeki ilişkisel veri tabanında tutulmuştur. Yazılım karmaşıklığında 7,8,9+ gibi değerlerin olmayışı yazılımın özenli bir şekilde ve hata toleransı az olarak yazıldığını da göstermektedir. İzleyen kısımda bir başka proje olan muhasebe yazılımının geliştirme süreçleri anlatılmaktadır. Ayrıca proje ile ilgili detaylı bilgilendirme tablosu ve geliştirilen projelerin ölçüm bilgilerine de yer verilmiştir.

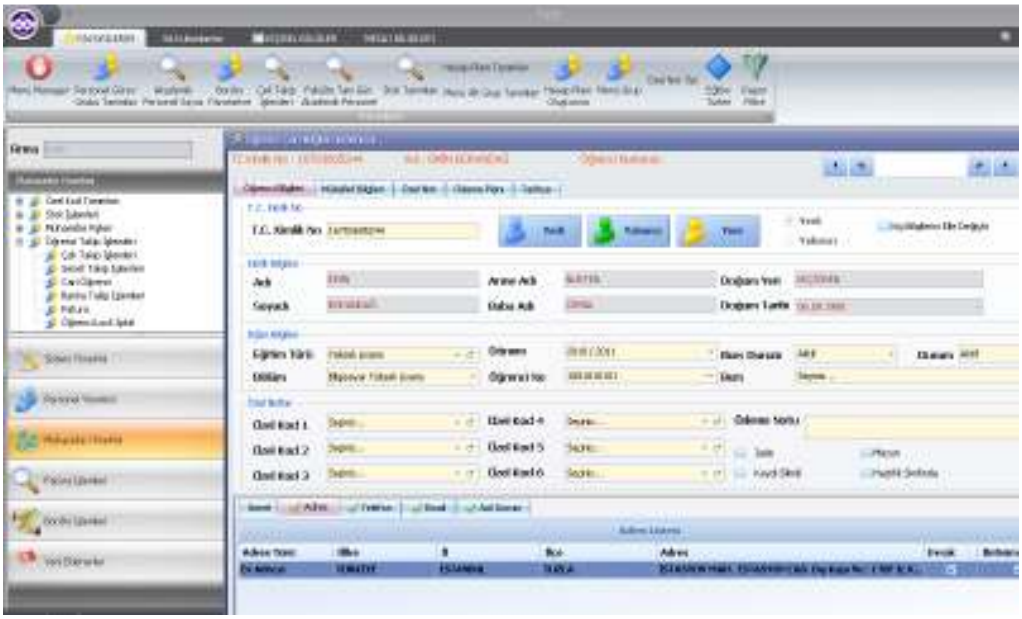
Yazılım projesi -3 muhasebe bilgi sistemi: Maltepe üniversite ve kolej'de kullanılmak üzere tasarlanmış bir muhasebe bilgi sisteminin oluşturulması. Üniversitede geliştirilen diğer yazılım projeleri ile eş güdümlü çalışacak muhasebe otomasyon sisteminin geliştirilmesi. Proje geliştirme yöntemi olarak hibrit test metodu ve yazılımı kullanılmıştır. Yazılım projesinin geliştirme aşamaları aşağıda verilmiştir.

Ana ve Yan Süreçlerin Tespit Edilmesi: Bu sürece Maltepe üniversitesi ve Marmara koleji tarafında kullanılan eski yazılım incelenerek başlanılmıştır. Maltepe üniversite idari işler personelinin ihtiyaçlarını tam olarak karşılamayan ve her yıl için bakım anlaşması isteyen programın incelenmesi ile ana ve yan süreçlerin detaylı tanımlama işlemlerine başlanmıştır. Bu aşamada tanımlanılan bilgilerin hibrit test yazılımı içerisinde kayıtları yapılmıştır. Sonraki aşamada hangi yazılımcının hangi süreçler ile ilgilenecekleri kısmı tanımlanarak yazılımın kodlanması kısmına geçilmiştir.

Yazılımın Kodlanması: Bu süreçte, Maltepe üniversitesinde kullanılan kodlama standartlarına uygun olarak, yazılım geliştirme çalışmalarına başlanmıştır. Müşteri kartlarındaki veriler dikkate alınarak oluşturulan gereksinimler, ekip lideri tarafından teknik kartlara dönüştürülmüştür. Diğer geliştirilen yazılım projelerinde olduğu gibi, basit tasarım uygulaması kullanılmıştır. Bu sayede kısa sürede anlaşılması, yönetilmesi ve değiştirilebilmesi kolay bir yazılım geliştirilmesi amaçlanmıştır. Müşterinin değişen gereksinimleri temel alınmıştır. Ekip içinde mümkün olduğunca basit tasarıma ve geri dönüşlere önem verilerek program geliştirilmeye çalışılmıştır. Ayrıca, projenin her aşamasında test önde tutulmuş; önce test yöntemi kullanılarak kod geliştirilmiştir. Kodlama işlemi sürecinde, yapılan sistem değişiklikleri ve yeni bileşenler hemen sisteme entegre edilerek ortaya çıkabilecek hataların erken safhalarda tespit edilmesi sağlanmıştır. Proje içerisinde sürekli günlük entegrasyonlarla üç haftada bir yeni sürümler oluşturulmuştur.

Yazılımın Testi: Geliştirilen yazılımın testi, tasarım aşamasından başlanılarak her bir süreç içerinde ayrı ayrı ele alınmıştır. Otomasyon sisteminin testi üç aşamada gerçekleştirilmiştir: Bu aşamada öncelikli olarak hangi kısımların ne şekilde test edileceği belirlenmiştir. Birinci aşamada ortak sahiplenme uygulamasına uygun olarak Black-Box unit testleri uygulanmıştır. İkinci aşamada ise, beyaz-kutu birim test (white-box unit testing) tekniği XP'nin önce test uygulamasına uygun olarak kullanılmıştır. Kodlandıktan sonra yazılım proje liderinin denetiminde diğer yazılımcılara test ettirilmiştir. Ayrıca kullanıcı hikâyelerine uygun olarak yazılım müşteriye test ettirilmiştir. Bu test işlemi sonunda müşteriden gelen geri bildirimlere göre yazılım ekibi tarafından yazılım içerisindeki hatalar giderilmiş ve yeni sürümler ortaya çıkarılmıştır. Testin son aşaması kabul testleridir. Bu kabul testleri, yazılımı kullanacak personel tarafından gerçek veriler kullanılarak yapılmıştır. Kabul testlerinin müşterinin gereksinimlerini karşılaması ile birlikte yazılımın teslim süreci tamamlanmıştır. Proje geliştirilmesi aşamasında yapılan işlemlerin takibi için daha önceden geliştirilen hibrit test programı kullanılmıştır. Yukarıda anlatılan test yöntemlerinin haricinde, adım adım inceleme(walkthrough) yöntemi de kullanılmıştır. Bu sayede kodu geliştiren yazılımcının gözünden kaçan hataların diğer yazılımcılar tarafından kolaylıkla tespit edilmesi ve düzeltilmesi sağlanmıştır. Son olarak proje lideri tarafından test edilen yazılım uygulama öncesi müşteri testine tabi tutulmuştur. Müşteri testini takiben

istenilen özelliğe ilişkin kod sisteme dahil edilmiştir. Şekil 4-9'de gerçekleştirilen muhasebe otomasyon sistemine ait ekran görüntüsü verilmiştir.



Şekil 4-9 Muhasebe Otomasyon Sistemi Ekran Görüntüsü

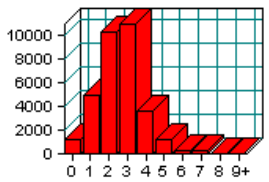
Bilgilendirme Tablosu: Çizelge 4-6'da gerçekleştirilen muhasebe otomasyon sistemine ait bilgilendirme tablosu verilmektedir. Gerçekleştirilen muhasebe sistemi; Maltepe üniversitesi kullanımının yanı sıra vakıf bünyesinde kurulan birçok kurumda kullanılmaktadır. Maltepe üniversitesi ve üniversiteye bağlı 3 şirketle, Bodrum'da faaliyet gösteren 2 otelde, İstanbul'da bulunan Marmara kolej ve koleje bağlı bulunan 12 şirket ile, Bodrum Marmara koleji ve koleje bağlı bulunan 3 ayrı şirketin bütün muhasebe işlemleri bu sistem üzerinden gerçekleştirilmektedir.

Çizelge 4-6 Muhasebe Otomasyon Programı Bilgilendirme Tablosu

Muhasebe Otomasyon Programı	
Yazılım Ekibi Büyüklüğü:	3 Yazılımcı 1 Test Sorumlusu
Eğitim Seviyesi:	1Yüksek Lisans 1 Lisans 2 Ön Lisans
Yazılım Tecrübe Bilgileri:	> 10yıl: 1 kişi 5-10 yıl: 3 kişi
Uzmanlık Bilgileri:	Yüksek
Programlama Dili Bilgisi:	Yüksek
Oluşturulan Müşteri Kart Sayısı:	89
Domain:	Windows Application, C#, SQL Server

Ölçümler: Gerçekleştirilen muhasebe otomasyon sistemi ölçümü için Source Monitor isimli program kullanılmıştır. Çizelge 4-7'da görüntülenen verilerin tümü bu program tarafından elde edilmiştir. Projenin satır büyüklüğü, komut sayısı, açıklama oranları, ortalama karmaşıklık oranları, programın içerisinde bulunan dosya (Örneğin frm,ocx,dll,vb) sayıları ve oluşturulan kodların karmaşıklık oranları Çizelge 4-7'de verilmiştir.

Çizelge 4-7 Muhasebe Otomasyon Programı Ölçümleri

Muhasebe Otomasyon Programı	
Satır Sayısı	53.556
Toplam Komut Sayısı	32.236
Açıklama Oranı	%2.5
Ortalama Karmaşıklık	2.52
Dosya Sayısı	80
Veri Oluşturma Tarihi	27 Ekim 2010
Karmaşıklık Oranları	**Komut Sayısı**
0	1206
1	4851
2	10124
3	10799
4	3530
5	1124
6	280
7	265
8	43
9+	14
Karmaşıklık Oranları Grafiği:	

26 Ekim 2010 tarihinde muhasebe otomasyon programına ilişkin satır sayısı. toplam komut sayısı, açıklama sayısı ve toplam karmaşıklık sayısı değerlerin elde edilmesinde Source monitör programı kullanılmıştır. Elde edilen sonuçlar Çizelge 4-7'de sunulmuştur. Çizelge 4-7 'de görüldüğü üzere ilgili yazılım 32.236 satırlık komuttan oluşmaktadır. Yazılıma başlanıldığı andan 26 Ekim 2010 tarihi itibariyle toplamda 557 hata ortaya çıkarılmış ve bulunan bu hatalar hibrit test yazılımı içerisindeki ilişkisel veri tabanında tutulmuştur. Gerçekleştirilen kodların ortalama karmaşıklık oranları 2.52 olarak bulunmuştur. Gerçekleştirilen kodların sadece %

5,35'lik bir kısmının 5 ve üzeri karmaşıklık değerine sahip olduğu görülmektedir. Bulunan oranlarda 5 ve daha üstü karmaşıklığın az olması yazılımların yönetimi ve yazılım kalitesi açısından önemlidir. Ayrıca proje sonucunda elde edilen 2.52 oranı yazılım karmaşıklığına neden olan gereksinimleri, kullanılan teknolojileri ve müşteri ilişkilerinin etkin bir şekilde kullanıldığının, yazılımın yönetilebilir olduğunun ve yazılım kullanılmaya başlandığında ortaya çıkacak hataların az olacağının da önemli bir göstergesidir.

Test öncelikli hibrit yazılım geliştirme metodu, hibrit test aracı ve yazılım projeleri ile ilgili değerlendirmeler: Test öncelikli hibrit yazılım geliştirme metodunda yazılım geliştirme aşamaları olarak ana ve yan süreçlerin tespit edilmesi, tasarım, yazılımın kodlanması, bakım ve yeni isterler süreçleri olarak belirlenmiştir. Her bir süreç içerisinde test işine öncelik verilerek XP'nin uygulamaları ve yaşam çevrim özellikleri kullanılmıştır. White box ve black box test yöntemleri de hibrit yazılım geliştirme metodu içerisinde etkin olarak kullanılmıştır.

Önce test yöntemine uygun olarak kodlama safhasına geçilmeden önce test durumları (text-cases) tanımlanmıştır. Hibrit test aracı; müşterilerden gelen istekleri, yazılım geliştirmek için gerekli olan teknik detayları, müşteri ve yazılımcı testleri ile ilgili gerekli bilgileri ilişkisel veritabanına kaydederek yazılım projelerinin etkin ve düzenli bir şekilde geliştirilmesini sağlamıştır. Tanımlanan bu yöntem ve araç sayesinde 3 ayrı yazılım projesi geliştirilmiştir. 5 kişiden oluşan bir yazılım ekibi projelerin geliştirilmesinde görev almıştır. Geliştirilen yazılım projelerinde test öncelikli hibrit yazılım geliştirme metodu ve hibrit test aracı etkin bir şekilde kullanılmıştır. XP'nin planlama oyunu uygulaması ile gerçekleştirilen yazılım proje büyüklükleri tahminine uygun olarak projeler belirlenen zaman içerisinde tamamlanmış bakım ve yeni isterler aşamasına geçilmiştir. Projeler için toplamda 100.000 satırın üzerinde kod oluşturulmuştur. Test öncellikli olarak geliştirilen bu 3 farklı yazılım projesi Maltepe üniversite içerisinde 1-2 yılı aşkın bir süredir kullanılmaktadırlar. Müşteriden gelen hatalar XP temelli yazılım çevrimi içerisinde test öncelikli hibrit yazılım geliştirme metodu ve hibrit test aracı kullanılarak çözülmektedir. Yazılım kalitesi için kullanılan en temel yöntem olan karmaşıklık oranlarına bakıldığında oluşturulan projelerin karmaşıklık değerlerinin sadece %1,86'sının 5 ve daha üzeri karmaşıklık değerine sahip

olduğu görülmektedir. Ayrıca müşteri tarafından iletilen ve hibrit test aracı içinde tutulan hata sayılar da bu bilgiyi doğrulamaktadır. Her üç projede elde edilen hata sayılarını; personel otomasyon sitemi için 817, çok daha az bir kod sayısına sahip olan optik otomasyon sistemi için 3 ve muhasebe yazılımı için ise 557 olarak bulunmuştur. Bulunan bu hataların çoğu yazılım geliştirimi sırasında kullanılan hibrit test yöntemi sayesinde tespit edilip, düzeltilmiştir.

Günümüzde karmaşık ve test işine yeteri kadar önem vermeyen yöntemlerin kullanılması nedeniyle yazılım projelerinde istenilen başarı oranlarına erişilememektedir. Büyük yazılım projeler için geliştirilen yazılım geliştirme yöntemleri ise kullanım ve yönetim açsından büyük zorluklar oluşturmaktadır. Ancak geliştirilen test öncelikli hibrit yazılım geliştirme metodu ve hibrit test aracı ile yazılım projeleri etkin bir şekilde yönetilmiştir. Kullanılan yöntem sayesinde hatalar daha erken safhalarda ortaya çıkartılmış ve düzeltilmeye çalışılmıştır. XP'nin yöntem içerisinde kullanımı ile yazılım firmaları açısından kullanım kolaylığı sağlanmıştır. White box ve black box test yöntemleri ile de test işi ön planda tutulmaya çalışılmıştır. Önerilen test öncelikli yazılım geliştirme yöntemi, küçük ve orta ölçekli farklı yazılım projeleri üzerinde uygulanabilir durumdadır. Geliştirilen hibrit test aracı sayesinde, projelerin gelişme seviyeleri ve anlık hata durumları da gözlenebilmektedir.

5. BÖLÜM

SONUÇ VE DEĞERLENDİRME

Son yirmi yılda yazılım geliştirmede kullanılan yazılım sistemlerini ve süreçlerini değerlendirmek, yazılımda kalite sertifikasyonunu sağlamak, süreçleri iyileştirmek ve yazılımların yeteneklerini belirlemek için çeşitli modeller geliştirilmiştir. Her ne kadar farklı yetenek ve özelliğe sahip modeller geliştirilse de, yazılım hataları hep önemli bir problem olarak gündemde kalmıştır. Şirketlerin en önemli varlıkları arasında gösterilen bilginin oluşturulmasında kullanılan yazılımların, eksiksiz ve hatasız çalışmaları çok büyük bir önem taşımaktadır.

Buradaki en önemli sorun, müşteri gereksinimlerinin artmasına paralel olarak karmaşıklığın artmasıdır. Hataların en büyük nedenlerinden biri analiz ve tasarım sürecinin öngörülen oranda yapılmamasıdır. Bir diğeri ise geliştirilen yazılım ürünlerinin yeteri kadar test edilmemesidir. Özellikle küçük ve orta ölçekli yazılım organizasyonlarında yazılım testi göz ardı edilmektedir. Bu nedenlerden dolayı müşterilere teslim edilen yazılımlarda büyük maddi kayıplar ve kaynak israfları yaşanmaktadır. Karşılaşılan sorunlar neticesinde, yazılımların büyük bir kısmının etkin bir şekilde kullanılmadığı görülmektedir.

Bu çalışmada ilişkisel veri tabanı kullanan, küçük ve orta ölçekli yazılım firmaları için hibrit test yöntemleri içeren Extreme Programming temelli bir hibrit test metodu ve aracı geliştirilmiştir.

Bu çalışmanın birinci bölümünde yazılım süreç, kalite ve metodolojileri hakkında genel bir değerlendirme yapılmış, yazılım ile ilgili temel tanım ve kavramlarla yer verilmiş, yazılım geliştirmede etkin olan kriterler, standartlar, yaygın kullanılan metodolojilere ilişkin bilgiler sunulmuştur. İkinci bölümde yazılım test teknikleri ve test yöntemleri ile ilgili konular hakkında açıklamalar yapılmıştır.

Çalışmanın üçüncü bölümünde Extreme Programming –XP başlığı ile ele alınarak yazılım geliştirme yöntemi hakkında bilgiler verilmiş ve Extreme Programlama modeline ilişkin kapsamlı bilgilendirme yapılmıştır.

Dördüncü bölümde önerilen test öncelikli hibrit yazılım test modeli anlatılmış; test temelli modelin gerçekleştirilmesinde araç olan uygulama yazılımına ve geliştirilen uygulama yazılımının ara yüz örnek ve özelliklerine ilişkin bilgilere, araştırma sonuçları ve önerilere ilişkin bilgilendirmelere yer verilmiştir.

Extreme Programming temelli bu hibrit test modelinin ele alındığı bu çalışmada toplamda 116.705 satır (lines) kod oluşturulmuş, 5 kişiden oluşan bir yazılım ekibi projelerin geliştirilmesinde görev almıştır. Oluşturulan projelerin karmaşıklık değerleri göz önüne alındığında sadece %1,86'sının 5 ve daha üzeri karmaşıklık değerine sahip olduğu görülmektedir. Bulunan oranlarda 5 ve daha üstü karmaşıklığın az olması yazılımların yönetimi ve yazılım kalitesi açısından da önemlidir. Bu yazılım projeleri Extreme Programming'in yaşam döngüsüne ve çalışma ortamına büyük oranda bağlı kalınarak geliştirilmiştir. Ayrıca bu yazılım projelerinin test kısmı için geliştirilen uygun hibrit test metodu ve test aracı kullanılmıştır.

Bu çalışmada aşağıdaki bulgular elde edilmiştir:

- Test öncelikli yazılım geliştirme yöntemi ve hibrit test aracının kullanımı ile yazılım projeleri etkin bir şekilde yönetilebilir.
- Geliştirilen hibrit test yöntemi 3 farklı yazılım projesi üzerinde uygulanmış ve yazılımların test temelli olarak geliştirilmesi sağlanmıştır.
- Kullanılan yöntem sayesinde hatalar daha erken safhalarda saptamış ve gerekli düzelmelerin daha önceden düzeltilmesine olanak sağlanmıştır.
- XP metodolojisinin ve önerilen test metodunun uygulandığı 3 adet kapsamlı yazılım projesi gerçekleştirilmiştir.
- Önerilen test öncelikli yazılım geliştirme yönteminin, küçük ve orta ölçekli farklı yazılım projeleri üzerinde uygulanabilirliği görülmüştür.
- Geliştirilen hibrit test aracı sayesinde, projelerin seviyeleri ve anlık hata durumları gözlenebilir.

KAYNAKLAR

1. **Acar, Ö. (2009).** *Extreme Programming.* Istanbul: Pusula Yayıncılık.
2. **Alkan, M .(2004).** *YazılımSınıfları.* http://www.csharpnedir.com/makalegoster.asp?MId=230.
3. **Andres, K. B. (2004).** *Extreme Programming Explained: Embrace Change* (Cilt 2nd edition). Addison-Wesley.
4. **Antón, L. W. (2004).** Toward a framework for evaluating Extreme Programming. (s. 11-20). In 8th International Conference on Empirical Assessment in Software Engineering (EASE '04).
5. **Beck, K. (2002).** *eXtreme Programing.* Addison-Wesley.
6. **Beedle, K. S. (2001).***Agile Software Development with Scrum.* . Alan R. Apt.
7. **Beizer, B. (1990).***Software Testing Techniques 2nd edition.* New York: Van Nostrand Reinhold.
8. **Borandağ, E. (2006).***Basamaklı CMMI Modeli ile Extreme Programming Metodunun Değerlendirilmesi.* İstanbul: Maltepe Üniversitesi, Fen Bilimleri Enstitüsü, Yüksek Lisans Tezi.
9. **Bossi, P. (2003).** Extreme Programming applied: a case in the private banking domain. *In Proceedings of OOP, Munich* .
10. ***Brian, R*. (1968).** *NATO Reports* http://homepages.cs.ncl.ac.uk/brian.randell/NATO/NATOReports/index.html adresinden alınmıştır
11. **Bures, L. L. (2006).** Essential communication practices for Extreme Programming in a global software development team. *Journal of Information and Software Technology Elsevier* .
12. **Burton, G.(2010).** *Software History* http://www.softwarehistory.org/history/Default.htm adresinden alınmıştır

13. **Chan, K. M. (2004).** Test driven development and software process improvement in china. *In Proceedings of the 5th International Conference on Extreme Programming and Agile Processes in Software Engineering (XP 2004), volume 3092 of Lecture Note.*

14. **Cockburn, A. (2004).** Crystal Clear: A Human-Powered Methodology for Small Teams. *Addison-Wesley Professional* .

15. **Deursen, A. V. (2009).** Delft University of Technology,Delft. *A Research Agenda.* The Netherlands Model-Driven Software Evolution.

16. **Dymond, D. H. (2006).** Appropriate agile measurements: Using metrics and diagnostics to deliver business value. *In Agile 2006 Conference*, (s. 126-131).

17. **Felsing, S. P. (2002).** *A Practical Guide to Feature Driven Development.* Prentice Hall.

18. **Gerald D. E. , Raymond M. Jr. (2007).** *Software testing across the entire software Development Life Cycle.*

19. **Gerald D. E, Raymond M. Jr. (2007).** *Software Testing.* John Wiley & Sons Inc.

20. **Glenford, J. ,Myers T. B**. **, Todd M. T. (2004).** The Art of Software Testing. *John Wiley & Sons, Inc New York NY USA*.

21. **Goyal, S. K. (2004).** *CMMI Implementation.* Tata McGraw-Hill Publishing Company; 1st

22. **Gray, L. (2008).** Standard for Information Technology. Software Life Cycle Processes Guidebook to IEEE/EIA 12207, Abelia.

23. **Group, B. W. (1998).** A Spiral Model of Software. *IEEE Computer Society* .

24. **Hendrickson, R. E. (2000).** *Extreme Programming Installed.* Addison-Wesley.

25. **Hetzel, W. C. (1998).** *The Complete Guide to Software Testing Wellesley, Mass. 2nd ,QED Information Sciences*

26. **Highsmith, J, (2002).** *Agile Software Development Ecosystems, Addison Wesley*

27. **Ian, S (2006.).** *Software Engineering* (Cilt 8th edition). Addison Wesley.

28. **IEEE Standard (1998).** 829-1998, IEEE Standard for Software Test Documentation.

29. **Ilene, B (2003).** *Practical Software Testing: A Process-Oriented Approach 1st edition.* Springer.

30. **İnce, F. (2004).** Yazılım Kalitesi ve Modelleri. Ders Notları, Maltepe Üniversitesi, Fen Bilimleri Enstitüsü.

31. **Jones, C. (2000).** *Software Assessments Benchmarks and Best Practices.* Addison Wesley.

32. **Kalaycı, O. (2005).** CMMI Süreçleri ve XP http://www.nitelik.net/sayfa-yayinlarimiz.asp alınmıştır.

33. **Kan, S. H. (2002).** Metrics and Models in Software Quality Engineering, 2nd Ed., Addison-Wesley Professional.

34. **Kemerer, S. C. (1994.).** A metrics suite for object oriented design. *IEEE Transactions on Software Engineering, 20(6)*.

35. **Keren, Y. D. (2005).** Agile metrics at the israeli air force. *In Agile 2005 Conference* ,

36. **Kuitunap, H. (2009).** Software_classification_explanation. *Software Business Lab,Helsinki University of Teknology* http://www.swbusiness.fi/attachments/software_classification_explanation.pdf.

37. **Martin, F. K. B. (1999).** Refactoring: Improving the Design of Existing Code. *Addison-Wesley*.

38. **McGregor**, **D. A. S. (2001).** *A Practical Guide to Testing Object-Oriented Software.* Addison-Wesley.

39. **Meyers, L. P. (1992).** *Measures For Excellence: Reliable Software On Time Within Budget.* Yourdon Press Computing Series.

40. **Müller, R. A. (2004).** Extreme programming in a university project. *In Proceedings of the 5th International Conference on Extreme Programming and Agile Processes in Software Engineering (XP 2004), volume 3092 of Lecture Notes on Computer Science.*

41. **Nancy, M., Lawrence, T, Suzanne C. (1996).** *Best Training Practices Within the Software Engineering Industry* Technical Report SEI

42. **Newkirk, J. (2002).** Introduction to Agile Processes and Extreme Programming Thought Works. *Proceedings of the 24th International Conference on Software Engineering* , 695-696.

43. **Vardarlı, O. (2008).** *Saydam kutu test yönteminin nesne tabanlı bir sisteme uygulanışı.* İstanbul: Maltepe Üniversitesi, Fen Bilimleri Enstitüsü.

44. **Jorgensen, P. (2006).** *Software Testing: A Craftsman's Approach.* Auerbach Pub.

45. **Pettichord, C. K. (2002).** *Lessons Learned in Software Testing.* John Wiley & Sons.

46. **Poppendieck, M. P. (2003).** *Lean Software Development: An Agile Toolkit for Software Development Managers.* Addison-Wesley.

47. **Torkar**, **R. (2006).** Towards Automated Software Testing Techniques Classifications and Frameworks. page 6.

48. **Richard, P. (2004).** *Soware Engineering : A Practitioner's Approach.* McGraw Hill Higher Education.

49. **Robert, V. B**. **(2000).** *Testing Object-Oriented Systems: Models, Patterns and Tools.* Addison Wesley.

50. **Pressman, R. S. (2004).** Software Engineering: A Practitioner's Approach. *McGraw Hill Higher Education* , pages 444-637.

51. **Sallie, H. W. (1993).** *Object oriented metrics that predict maintainability.* J. Systems and Software, 23:.

52. **Sarıdoğan, E. (2004).** *Yazılım Mühendisliği.* Papatya Yayıncılık.

53. **Schach, S. R. (2007).** *Object-Oriented & Classical Software Engineering* (Cilt 7th Ed). McGraw Hill.

54. **Shrum, M. B. (2007).** *CMMI(R): Guidelines for Process Integration and Product Improvement.* Addison-Wesley Professional .

55. **Software Develop Team (2001).** Yazılım İçin Ömür Devri Boyunca Birleştirilmiş NATO Kalite Gereksinimleri AQAP-160 .

56. **Springer, B. I. (2003).** *Practical Software Testing: A Process-Oriented Approach 1st edition.*

57. **Torteli, A. F. (2005).** XP south of the equator: An experience implementing XP in Brazil. *In Proceedings of the 6th International Conference on Extreme Programming and Agile Processes in Software Engineering*, (s. 10-18).

58. **Turner, B. B. (2003).** Using risk to balance agile and plan-driven methods. *In IEEE Computer, volume 36*, pages 57-66.

59. **Turner, B. B. (2003).** *Balancing Agility and Discipline: A Guide for the Perplexed.* Addison-Wesley.

60. **William, A, Richard, K. S. (1996)**. *History of Software Engineering,.*

61. **Williams, L. (2002).** *Pair Programming Illuminated.* Addison-Wesley Professional.

ÖZGEÇMİŞ

1980 yılında Ankara İli Keçiören ilçesinde doğdu. İlköğretim ve liseyi Amasya'da tamamladı. Maltepe Üniversitesi Mühendislik-Mimarlık Fakültesi, Bilgisayar Mühendisliği Bölümü'nü bitirdi. Yüksek lisans eğitimini yine aynı üniversitede Fen Bilimleri Enstitüsü'nde Bilgisayar Mühendisliği alanında yaptı. Doktora eğitimine Trakya Üniversitesi Bilgisayar Mühendisliği Bölümünde devam etti. 2004 yılından beri Maltepe Üniversitesinde Bilişim Bölümünde çalışmaktadır.