



T.C.
DÜZCE ÜNİVERSİTESİ
TEKNOLOJİ FAKÜLTESİ
BİLGİSAYAR MÜHENDİSLİĞİ

AGİLE (ÇEVİK YAZILIM) YÖNTEMLERİ

162120012 ALİ SARI

II.ÖĞRETİM II.GRUP

DANISMAN

SERDAR BİROĞLU

DÜZCE - 2021

İÇİNDEKİLER

Sayfa No

ŞEKİL LİSTESİ.....	5
KISALTMALAR.....	6
1. Agile (Çevik Yazılım) Yöntemi	7
1.1 Agile nedir?	7
1.2 Agile metodunun tanımı.....	7
1.3 Agile (Çevik) Tarihçesi.....	7
1.4 Agile Proje Yönetimi'nin Tanımı	8
1.5 Agile neyi amaçlar?	8
1.6 Agile (Çevik) Manifestosunda 4 Temel Değeri	9
1.7 Agile Proje Yönetimi 12 İlkesi.....	9
1.8 Agile (Çevik) yöntemin avantajları.....	11
1.9 Agile (Çevik) yöntemin dezavantajları	12
2. Agile (Çevik) Gelişim Döngüsü	13
3. Agile (Çevik) ve Geleneksel (şelale(waterfall) veya spiral) yöntem arasındaki farklar	14
4. Agile (Çevik) yaklaşım çeşitleri.....	16
4.1 Extreme Programming.....	16
4.1.1 Extreme Programming Nedir?	16
4.1.2 Extreme Programing 4 Temel Noktası	16
4.1.2.1 İletişim	17
4.1.2.2 Basitlik.....	17
4.1.2.3 Geri Besleme	17
4.1.2.4 Cesaret.....	17
4.1.3 Extreme Programming Prensipleri	18
4.1.3.1 Rapid Feedback (Hızlı geri bildirim)	18
4.1.3.2 Assume Simplicity (Basitliği tercih etmek)	18
4.1.3.3 Incremental Change (İnkrementel değişiklik).....	18
4.1.3.4 Embracing Change (Değişimi istemek)	18
4.1.3.5 Quality Work (Kaliteli iş)	18
4.1.3.6 Teach Learning (Öğrenmeyi öğret)	19

4.1.3.7 Small Initial Investment (Az başlangıç yatırımı)	19
4.1.3.8 Play to win (Kazanmak için oyna)	19
4.1.3.9 Concrete Experiments (Somut denemeler)	19
4.1.3.10 Open, honest Communication (Açık ve samimi iletişim).....	19
4.1.3.11 Work with people's instincts, not against them (Takımın içgüdülerini kullan, onlara karşı koyma)	19
4.1.3.12 Accepted Responsibility (Sorumluluk üstlenmek)	20
4.1.3.13 Local Adaptations (Sürecin ortam şartlarına adapte edilmesi).....	20
4.1.3.14 Travel light (Az yükle yolculuk yapmak).....	20
4.1.3.15 Honest Measurement (Doğru ölçüm)	20
4.1.4 XP Teknikleri (XP Practices)	20
4.1.4.1 On-site Customer (Programcıya yakın müşteri)	20
4.1.4.2 Standup-Meeting (Ayakta toplantı).....	21
4.1.4.3 Planning Game (Planlama oyunu)	21
4.1.4.4 Short Releases (Kısa aralıklarla yeni sürüm)	21
4.1.4.5 Retrospective (Geriye bakış)	21
4.1.4.6 Metaphor (Mecaz)	21
4.1.4.7 Collective Ownership (Ortak sorumluluk)	22
4.1.4.8 Continuous Integration (Sürekli entegrasyon)	22
4.1.4.9 Coding Standards (Kod standartları)	22
4.1.4.10 Sustainable Pace (Kalıcı tempo).....	22
4.1.4.11 Testing (Test etmek).....	22
4.1.4.12 Simple Design (Sade tasarım).....	22
4.1.4.13 Refactoring (Yeniden düzenleme)	23
4.1.4.14 Pair Programming(Eşli çalışma).....	23
4.1.5 Extreme Programming Avantajları	23
4.1.6 XP Proje Safhaları	23
4.1.6.1 Keşif safhası (Exploration Phase)	24

4.1.6.2 Planlama Safhası (Planning Phase)	24
4.1.6.3 İterasyon ve Sürüm Safhası (Iterations to Release Phase)	24
4.1.6.4 Bakım Safhası (Maintenance Phase):	24
4.2 Feature Driven Development.....	24
4.3 Adaptive Software Development	26
4.4 Dynamic Systems Development Method	27
4.4.1 DSDM ayırt edici özellikleri :	27
4.4.2 Dinamik Sistem Geliştirme Proje Yapısı	28
4.4.3 Dinamik Sistem Geliştirme Süreci.....	28
4.5 Agile Unified Process.....	29
4.5.1 AUP Yaşam Döngüsü: 7 Disiplin.....	29
4.6 Test-Driven Development	30
4.7 Kristal (Crystal)	34
4.8 Açık Kaynak Geliştirimi (Open Source Development)	34
4.9 Rational Unified Process	35
4.10 Microsoft Solution Framework for Agile	36
4.11 Kanban	37
4.12 Yalın Yazılım Geliştirme.....	38
4.13 Scrum.....	39
4.13.1 Scrum Takımı:	39
4.13.2 Backlog.....	40
4.13.3 Sprint.....	40
4.13.4 User Story	41
4.13.5 Poker Kartları	42
4.13.6 Scrum XP karşılaştırılması	42
KAYNAKLAR.....	43

ŞEKİL LİSTESİ

Sayfa No

Şekil 1. AGİLE (Çevik Yazılım) kurucuları	8
Şekil 2. AGİLE (Çevik) Gelişim Döngüsü	13
Şekil 3. Waterfall Model	14
Şekil 4. XP 4 Temel Noktası	16
Şekil 5. XP Prensipleri	18
Şekil 6. XP Teknikleri	20
Şekil 7. XP Proje Safhaları	23
Şekil 8. DSDM	28
Şekil 9. Dinamik Sistem Geliştirme Süreci	28
Şekil 10. AUP Yaşam Döngüsü: 7 Disiplin	29
Şekil 11. Test-Driven Development	30
Şekil 12. Bug'ın maliyetinin SDLC'de hangi fazda ne kadar olduğu görülmekte.....	32
Şekil 13. TDD İyileştirme yapılması.....	33
Şekil 14. TDD uygulandığında ve uygulanmadığındaki karşılaştırması.....	33
Şekil 15. RUP'un Gelişimi	36
Şekil 16. Scrum Süreci	39
Şekil 17. Scrum Takım	39
Şekil 18. Backlog	40
Şekil 19. Sprint.....	40
Şekil 20. Sprint Gösterimi	40
Şekil 21. User Story	41
Şekil 22. Her bir User Story farklı bir boyutu	41
Şekil 23. Poker Kartları	42
Şekil 24. SCRUM Board	42

KISALTMALAR

XP	eXtreme Programming
FDD	Feature Driven Development
ASD	Adaptive Software Development
DSDM	Dynamic Systems Development Method
AUP	Agile Unified Process
TDD	Test-Driven Development
AKG	Açık Kaynak Geliştirimi
RUP	Rational Unified Process
MSF	Microsoft Solution Framework for Agile

1. Agile (Çevik Yazılım) Yöntemi

1.1 Agile nedir?

Agile modeli proje yönetimi, yazılım geliştirme sürecinde karşılaşılan problemleri çözmek üzere, tekrarlanan yazılım geliştirme modeli taban alınarak geliştirilmiş, sık aralıklarla parça parça yazılım teslimatını ve değişikliği teşvik eden bir yazılım geliştirme modeli. Genellikle sprint olarak bilinen artımlı, yinelemeli iş dizilerini kullanır.

1.2 Agile metodunun tanımı

Sprint, bir projenin belirli bir aşaması için ayrılan süredir. Sprintlerin süresi dolduğunda proje tamamlanmış sayılır. Ekip üyeleri arasında, gelişimin tatmin edici olup olmadığı konusunda anlaşmazlıklar olabilir; ancak, projenin belirli bir aşamasında bu konuda daha fazla çalışma yapılmayacaktır. Projenin kalan aşamaları, kendi zaman dilimleri içinde geliştirilmeye devam edecektir.

1.3 Agile (Çevik) Tarihçesi

Mevcut Agile proje yönetimi yöntemlerinin çoğunun kökleri yazılım geliştirmesine dayanır. 1990'lı yıllarda yazılım geliştirmesi yapan ekipler “ağır” geleneksel proje yöntemleri ile (Örneğin : waterfall) çalışmalarını ilerletemediklerini fark ettiler.

Bu eski yöntemlerin esneklik, uyumluluk ve özerklik gibi alanlardaki eksiklerini buldular. Eski proje yöntemlerinde proje içerisinde sapmalar olmamalıydı, bu sapmalar çok ciddi maliyetler doğurabiliyor idi.

Sonucun kesin ve net olduğu endüstrilerin (örneğin bir üretim hattını düşünebilirsiniz) aksine yazılım sektörü içerisinde değişim yazılımın temelinde olması gereken bir özelliktir.

Bu nedenle başlangıçta yapılan proje yönetim planına bağlı kalmak yerine, çevik proje yönetim yöntemleri; ekiplerin bu değişiklikleri mümkün olan en iyi ürünü yapmak için dikkate alabilecekleri anlamına geliyordu.

2001 yılında bir grup (17 kişi) yazılım geliştiricisi bir araya gelerek Agile proje yönetiminin temellerini atacak olarak kararları aldılar. Ve bu kararları Agile Yazılım Geliştirme Manifestosu olarak yayınladılar.



Şekil 1. AGİLE (Çevik Yazılım) kurucuları

1.4 Agile Proje Yönetimi'nin Tanımı

Bunların hepsi yazılım geliştirme odaklı görünüyorsa endişelenmeyin. Yazılım geliştirme süreçleri düşünülerek birçok çevik proje yönetimi metodolojisi geliştirilmiştir, ancak temel Agile değerleri ve ilkeleri, ürün ekiplerinden pazarlama ekiplerine kadar birçok farklı ekip için yararlıdır.

Agile Proje Yönetimi'nin tanımını aşağıdaki gibi yapabiliriz.

Agile Proje Yönetimi, sürekli test ve değişime açık, yinelemeli bir proje yönetimi yaklaşımıdır.

1.5 Agile neyi amaçlar?

- ❖ Kendini örgütleyebilen, çok fonksiyonlu takımlar oluşturabilmeyi
- ❖ Projeleri parçalar haline teslim edebilmeyi
- ❖ Projelerin çıkış hızını artırabilmeyi
- ❖ Projelerin kalitesini artırabilmeyi
- ❖ Proje ekibinin ve paydaşların değişime olan uyumlarını artırabilmeyi
- ❖ Sürekli öğrenen ve değişime açık bir kültürün oluşmasını ve gelişmesini

1.6 Agile (Çevik) Manifestosunda 4 Temel Değeri

- İş süreçleri ve araçlardan ziyade bireyler ve bireyler arasındaki etkileşim değerlidir.
- Kapsamlı bir dokümantasyon sürecinden ziyade, çalışan bir yazılım ortaya koymak daha önemlidir.
- Müşteri ile iş birliği yapmak, sözleşme görüşmelerinden daha önemlidir.
- Değişime cevap vermek, mevcut planı izlemekten daha önemlidir.

1.7 Agile Proje Yönetimi 12 İlkesi

Agile Manifesto'ya göre, bu proje yönetiminin 12 temel ilkesi vardır. Manifesto'nun kendi sözleriyle, bunlar aşağıdaki şekildedir.



1- En önemli öncelik değerli yazılımın erken ve devamlı teslimini sağlayarak müşterileri memnun etmektir.

Müşteriler, sürümler arasında uzun süre beklemek yerine düzenli aralıklarla çalışan yazımı aldıklarında daha mutlu olurlar.



2- Değişen gereksinimler yazılım sürecinin son aşamalarında bile kabul edilmelidir. Çevik süreçler değişimi müşterinin rekabet avantajı için kullanır.

Bir gereksinim veya özellik talebi değiştiğinde gecikmeleri önleyebilme yeteneği.



3- Çalışan yazılım, tercihen kısa zaman aralıkları belirlenerek birkaç haftada ya da birkaç ayda bir düzenli olarak müşteriye sunulmalıdır.

Takım, çalışan yazılımın düzenli olarak teslim edilmesini sağlayan yazılım Sprint'leriyle çalıştığı için Scrum bu prensibi yerine getirir.



4- İş süreçlerinin sahipleri ve yazılımcılar proje boyunca her gün birlikte çalışmalıdırlar.

İş sürecinin sahibi ve teknik ekip uyumlu hale geldiğinde daha iyi kararlar alınır.



5- Projelerin temelinde motive olmuş bireyler yer almalıdır. Onlara ihtiyaçları olan ortam ve destek sağlanmalı, işi başaracakları konusunda güven duyulmalıdır.

Mutsuz takımlara kıyasla, motive takımların en iyi işlerini teslim etme olasılığı çok daha fazladır.



6- Bir yazılım takımında bilgi alışverişinin en verimli ve etkin yöntemi yüzyüze iletişimidir.

Geliştirme takımı aynı yerde olduğunda iletişim çok daha başarılı olur.



7- Çalışan yazılım ilerlemenin birincil ölçüsüdür.

Müşteriye işlevsel bir yazılım sunmak ilerlemeyi ölçen en temel faktördür.



8- Çevik süreçler sürdürülebilir geliştirmeyi teşvik etmektedir. Sponsorlar, yazılımcılar ve kullanıcılar sabit çalışma temposunu sürekli devam ettirebilmelidir.

Takımlar, çalışan yazılımı teslim edebilecekleri tekrarlanabilir ve sürdürülebilir bir çalışma temposu oluştururlar ve bunu her sürümde tekrarlarlar.



9- Teknik mükemmeliyet ve iyi tasarım konusundaki sürekli özen çevikliği artırır.

Doğru beceriler ve iyi tasarım; ekibin hızı korumasını, ürünü sürekli iyileştirmesini ve değişimi sürdürmesini sağlar.



10- Sadelik, işin özü olmayan işlerin yapılmamasını en üst seviye tutmak elzemdir.

Şimdilik işi bitirecek kadar geliştirme.



11- En iyi mimariler, gereksinimler ve tasarımlar kendi kendini örgütleyen takımlardan ortaya çıkar.

Karar verme gücüne sahip, sahiplik alabilen, diğer takım üyeleriyle düzenli olarak iletişim kuran, yüksek kalitede ürünler sunan fikirlerini paylaşan, yetenekli ve motive takım üyeleri.



12- Takım, düzenli aralıklarla nasıl daha üretken ve verimli olabileceğini değerlendirir ve adapte olur.

Kişisel gelişim, süreç iyileştirme, gelişen beceriler ve teknikler ekip üyelerinin daha verimli çalışmasına yardımcı olur.

İster gerçek bir yazılımdan olsun, ister başka bir şeyden Agile Proje Yönetimi ile hızlı ve sık yenilemeler ile geliştirmeyi gerektirir.

Bu ilkeler bize öğrenmeyi, öğrenmeye göre kendimizi düzenlemeyi ve birlikte çalışmayı öğretir.






1.8 Agile (Çevik) yöntemin avantajları

Çevik, 1990'larda farklı hafif yazılım yaklaşımlarından yola çıkarak; katı, doğrusal Şelale (waterfall) metodolojisinden hoşlanmayan bazı proje yöneticilerinin bir yanıtıydı. Esneklik, sürekli iyileştirme ve hız odaklanır.

İşte Agile'nin en büyük avantajlarından bazıları:






Değişim benimsenir : Kısa planlama döngüleri ile, proje sırasında herhangi bir zamanda değişiklik yapmak ve kabul etmek kolaydır. Ekiplerin haftada bir kez projede değişiklik yapmasına izin vermek için, biriktirmeyi yeniden düzenleyip yeniden konumlandırmak için her zaman bir fırsat vardır.

-  **Bitiş hedefi bilinmeyen projeler :** Çevik, nihai hedefin açıkça tanımlanmadığı projeler için çok faydalıdır. Proje ilerledikçe hedefler ortaya çıkacak ve gelişim bu gelişen gereksinimlere kolayca adapte olabilecektir.
-  **Daha hızlı, yüksek kaliteli çıktılar :** Projeyi tekrar tekrar devreye almak (yönetilebilir birimler), ekibin yüksek kaliteli geliştirme, test etme ve işbirliğine odaklanmasını sağlar. Her yineleme (sprint) sırasında test yapmak, hataların daha hızlı tanımlanmasını ve çözülmesini sağlar. Ve bu yüksek kaliteli yazılım tutarlı, art arda yinelenen adımlarla daha hızlı teslim edilebilir.
-  **Güçlü takım etkileşimi :** Çevik, sık iletişimin ve yüz yüze etkileşimlerin önemini vurgulamaktadır. Takımlar birlikte çalışır ve insanlar projelerin sorumluluğunu ve kendi parçalarına düşeni üstlerine alabilirler.
-  **Müşteriler dinlenir:** Müşterilerin teslim edilen işi görmeleri, girdilerini paylaşmaları ve son ürün üzerinde gerçek bir etkisi olması için birçok fırsatı vardır. Proje ekibiyle çok yakın çalışarak sahiplik duygusu kazanabilirler.
-  **Sürekli iyileştirme :** Çevik projeler, tüm proje boyunca kullanıcılardan ve ekip üyelerinden geri bildirim almayı teşvik eder, böylece öğrenilen dersler, gelecekteki iterasyonları iyileştirmek için kullanılır.

1.9 Agile (Çevik) yöntemin dezavantajları

Çevik'teki esneklik seviyesi genellikle pozitif olsa da, aynı zamanda bazı sorunlara da yol açabilir. Katı bir teslimat tarihi belirlemek zor olabilir, dokümantasyon ihmal edilebilir veya nihai ürün orjinal olarak tasarlanandan çok farklı olabilir.

İşte Agile'nin dezavantajlarından bazıları:

-  **Planlama daha az sabit olabilir:** Bazen vazgeçilemez bir teslimat tarihini sabitlemek zor olabilir. Agile belirli süreli yinelemeli çıktılara dayalı olduğundan ve proje yöneticileri genellikle görevleri tekrarlamakta olduğundan, başlangıçta teslimat için planlanan bazı öğelerin zamanında tamamlanamaması mümkündür. Ayrıca, projenin herhangi bir anında yeni bir sprint eklenebilir ve bu da nihai hedef süreyi uzatacaktır.
-  **Takım bilgili olmalıdır:** Çevik takımlar genellikle küçüktür, 8-10 kişi civarında olması tercih edilir. Bu yüzden takım üyeleri çeşitli alanlarda çok yetenekli olmalıdırlar. Ayrıca seçilen Agile metodolojiyi anlamalı ve rahat hissetmelidirler.
-  **Yazılım geliştiricilerden zaman taahhüdü:** Çevik, geliştirme ekibi tamamen projeye adanmış olduğunda en başarılı olanıdır. Geleneksel bir yaklaşımdan daha fazla zaman alan Çevik

süreç boyunca aktif katılım ve işbirliği gereklidir. Ayrıca, bu geliştiricilerin projenin tüm süresine bağlı kalmaları gerektiği anlamına gelir.



Belgeler ihmal edilebilir: Agile Manifesto, kapsamlı bir dokümantasyon yerine çalışma yazılımını tercih eder, bu nedenle bazı ekip üyeleri, dokümantasyona odaklanmayı daha az önemli gibi hissedebilirler. Kapsamlı dokümantasyon, proje başarısına yol açmamakla birlikte, Çevik takımlar dokümantasyon ve tartışma arasındaki doğru dengeyi bulmalıdır.



Nihai ürün çok farklı olabilir: Çevik projeler ilk başlarda kesin bir plana sahip olmayabilir, bu nedenle nihai ürün başlangıçta tasarlanandan çok farklı görünebilir. Agile çok esnek olduğu için, değişen müşteri geri bildirimlerine dayanarak yeni iterasyonlar eklenebilir, bu da çok farklı bir hedefe gitmeye yol açabilir.

2. Agile (Çevik) Gelişim Döngüsü



Şekil 2. AGİLE (Çevik) Gelişim Döngüsü

Agile geliştirme döngüsündeki aşamalar burada. Bu aşamaların art arda gerçekleşmemesi gerektiğini belirtmek önemlidir; esnektir ve her zaman gelişmektedir. Bu fazların çoğu paralel olarak gerçekleşir.



Gereksinim analizi: Bu aşama yöneticileri, paydaşları ve kullanıcılarla iş gereksinimlerini tanımlamak için birçok toplantıyı içerir. Ekip, ürünü kimin kullanacağı ve nasıl kullanacağı gibi bilgileri toplamalıdır. Bu gereksinimler ölçülebilir, ilgili ve ayrıntılı olmalıdır.



Planlama: Bir fikrin uygun ve uygulanabilir olduğu düşünüldüğünde, proje ekibi bir araya gelir ve istenilen özellikleri tanımlamaya çalışır. Bu aşamanın amacı, düşüncüyü daha küçük parçalara (özellikler) ayırmak ve sonra her bir özelliği önceliklendirmek ve bir iterasyona atamaktır.

Tasarım: Sistem ve yazılım tasarımı, önceki aşamada belirtilen gereksinimlerden hazırlanmıştır. Ekip, ürün veya çözümün neye benzeyeceğini düşünmelidir. Test ekibi ayrıca bir test stratejisi veya devam etmek için bir plan ile gelir.

Uygulama, kodlama veya geliştirme: Bu aşama, özelliklerin oluşturulması ve test edilmesiyle ve tekrarlama için zamanlamaların planlanmasıyla ilgilidir. Geliştirme aşaması, teslim edilen hiçbir özellik olmadığından sprint 0 ile başlar. Bu iterasyon, sözleşmelerin sonuçlandırılması, ortamların hazırlanması ve finanse edilmesi gibi görevlerle birlikte harekete geçmek için temel oluşturuyor.

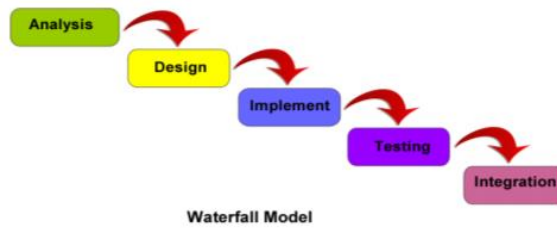
Test: Kod geliştirildikten sonra, ürünün müşteri ihtiyaçlarını ve eşleşen kullanıcı hikayelerini gerçekten çözdüğünden emin olmak için gereksinimlere karşı test edilir. Bu aşamada, birim test, entegrasyon testi, sistem testi ve kabul testi yapılır.

Çıktıyı Canlıya Alma: Test ettikten sonra, ürün kullanım için müşterilere teslim edilir. Ancak, dağıtım projenin sonu değildir. Müşteriler ürünü kullanmaya başladığında, proje ekibinin ele alması gereken yeni sorunlarla karşılaşabilirler.

İzleme ve takip etme: Hem çıktının düzgün çalışıp çalışmadığı izlenir ve gerekirse müdahale edilir hem de kullanıcılardan geri bildirim toplanır ve yeniden gereksinim analizi ve planlamalara girdi oluşturulur.

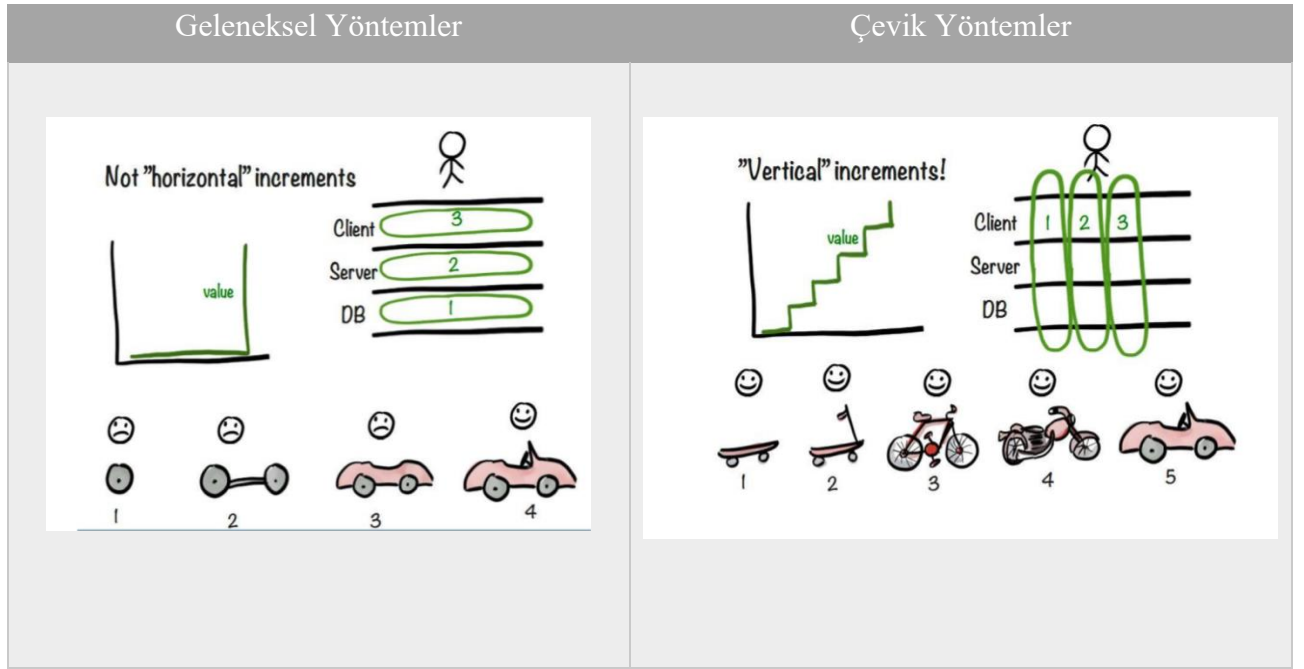
3. Agile (Çevik) ve Geleneksel (şelale(waterfall) veya spiral) yöntem arasındaki farklar

Çağlayan modeli 2008 yılında dahi geçerliliğini koruyan bir modeldir ve çevik modellemeden farklılık gösterir. Bu model yazılım projesini baştan sona planlar. Gelişim, sunulabilir işler açısından ölçülür: talep açıklamaları, tasarım dokümanları, test planları, kod incelemeleri vb. Bu durum belli aralıklara bölünmeye uygun değildir ve ilerideki değişikliklere uyum gösterilemez.



Şekil 3. Waterfall Model

Geleneksel Yöntemler	Çevik Yöntemler
Müşteriler ne istediğini iyi bilir.	Müşteriler ne istediğini keşfeder
Geliştiriciler neyi, ne şekilde üreteceklerini iyi bilir.	Geliştiriciler neyi nasıl üreteceğini keşfeder.
Bu yol boyunca hiç birşey değişmeyecektir.	Bu yol boyunca bir çok değişiklik yapılabilir.



	Geleneksel	Agile (Çevik)
Temel Varsayımlar	Sistemler tamamen tanımlanabilir, tahmin edilebilir, titiz ve kapsamlı bir planlama ile inşa edilebilir.	Hızlı geri bildirim sağlanarak değişime dayalı sürekli tasarım geliştirilir. Test etme ilkeleri kullanılarak küçük ekipler tarafından yüksek kaliteli, uyarlanabilir yazılım geliştirilebilir.
Kontrol	Süreç merkezli	İnsan merkezli
Yönetim stili	Komuta Kontrol	Liderlik ve İşbirliği
Bilgi yönetimi	Açık	Üstü kapalı
Rol ataması	Birey - uzmanlaşmayı tercih eder	Kendini organize eden ekipler - rol değişimini teşvik eder

İletişim	Resmi	Resmi değil
Müşterinin rolü	Önemli	Kritik
Proje döngüsü	Görevler veya faaliyetler rehberliğindedir.	Ürün özelliklerine göre yönlendirilir.
Geliştirme modeli	Yaşam döngüsü modeli (Şelale, Spiral veya bazı varyasyonlar)	Evrimsel teslim modeli
İstenilen organizasyonel form / yapı	Mekanik	Organik
Teknoloji	Kısıtlama yok	Nesneye yönelik teknolojiyi destekler

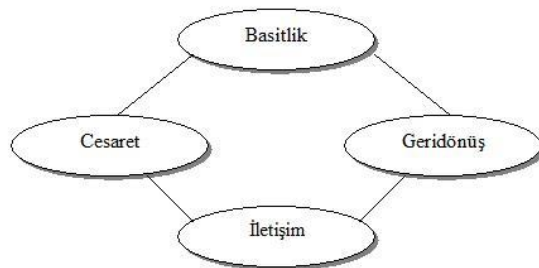
4. Agile (Çevik) yaklaşım çeşitleri

4.1 Extreme Programming

4.1.1 Extreme Programming Nedir?

En popüler çevik süreçlerden (Agile Process) birisi olarak bilinir. XP ile oluşturulan çevik süreçte müşteri ve gereksinimleri merkezi bir rol oynamaktadır. Yazılım esnasında XP ile tam belirli olmayan ve çabuk değişikliğe uğrayan müşteri gereksinimlerine ayak uydurulabilir. Bu konvansiyonel yazılım metotlarda mümkün değildir, çünkü proje öncesi müşteri gereksinimleri en son detayına kadar kağıda dökülmüştür. Oluşan bir dokümantasyon baz alınarak, yazılım gerçekleştirilir. Proje ilerledikçe müşteri tarafından yapılması istenen değişikliklerin maliyeti çok yüksek olacaktır, çünkü mevcut yapı (tasarım - design) istenilen değişikliklerin yapılmasını engelleyebilir ya da yeni bir yapılanmaya gidilmesi gerekebilir. XP, kullanıldığı projelerde formalite ve bürokrasinin mümkün en az seviyeye çekilmesine önem verir. Çevik olabilmek için az yükü yola çıkılması gerekmektedir. Bu yüzden proje öncesi geniş çapta tasarım ve dokümantasyon oluşturulmasına izin verilmez.

4.1.2 Extreme Programming 4 Temel Noktası



Şekil 4. XP 4 Temel Noktası

XP'nin özü bu dört değer içinde yatmaktadır. Bu değerler yaşandığı taktirde XP öğrenimi ve kullanımı kolaylaşır. Bu değerlerin geçerlilik bulmadığı ortamlarda XP'nin uygulandığı söylenemez. XP ile verimli bir çevik süreç oluşturabilmek için bu değerlerin hepsinin kabul görmesi ve uygulanması gerekmektedir.

Bu değerlerin ne anlama geldiğini yakından inceliyelim.

4.1.2.1 İletişim

XP in ilk temel taşı iletişimdir. Projelere baktığımızda ortaya çıkan önemli problemlerin insanların birbirleriyle tam olarak anlaşamaması nedeniyle olduğunu görünür. Bazen programcılar sormaları gereken doğru soruyu soramazlar. Bazen de projenin yapısıyla ilgili önemli bir gelişmeyi söylemezler. Bu yüzden projelerde çeşitli tıkanma noktaları olabilir. Bazen de proje yöneticileri programcılara belirtmeleri gerekenleri tam olarak belirtemezler. Bu da projenin gelişme süreçlerini olumsuz etkiler. XP de iletişim yüz yüze olmalıdır. İletişim sürekliliği olmalıdır. Yazılım ekibi ile yazılımı kullanacaklar arasında sıkı bir iletişim bağı olması esastır. Bu sayede sorunlar erken fark edilir.

4.1.2.2 Basitlik

XP in ikinci temel taşı basitliktir. Aslında basitlik sağlanması zor olan bir konudur. Basitlik, zorunlu işlerin yapılmasıdır. Gerekli olmayan bir şey varsa, kesinlikle bu konu XP basitlik ilkesi içerisinde olmamalıdır. Dünyadaki en zor şey gelecekte ne gibi ihtiyaçlarla karşılaşacağımızı bilmemektir. Bu nedenden dolayı oluşturulacak yapı, gelecekte ne gibi isterlerin ortaya çıkacağını tam olarak bilmeden oluşur. Buna göre esnek zaman ve maliyeti göz önüne alınmış bir yapı oluşturmak gerekmektedir. XP en iyi şekilde bu basitliği sağlamak için, bu günün ihtiyaçlarını hedef alarak esnek ve basit bir sistem gerçekleştirmeye çalışır.

4.1.2.3 Geri Besleme

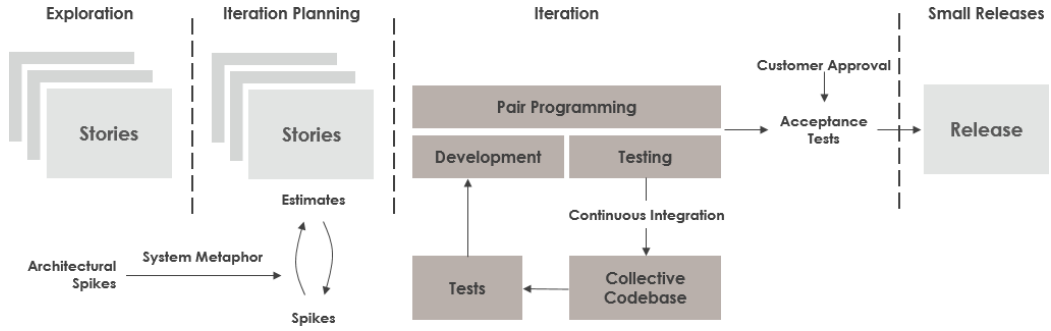
Sistemlerin geri dönüşümü olması sistemler için paha biçilemez bir fırsat yaratır. Geri dönüşüm sayesinde optimizasyonun oluşmasına engel olan tehlikeler ortadan kaldırılır. Öncelikle programcılar bütün sistemin mantıksal yapısını içeren bölüm testleri oluşturur. Programcılar sistem hakkında somut bilgiler elde ederler. Müşteriler yeni bir “stories” hikâye ile geldikleri zaman, programcılar hemen bu yeni gelen hikâyenin gerçekleşmesi ile ilgili çalışma yapıp bu “stories” e uygun bir çalışma gerçekleştirirler.

4.1.2.4 Cesaret

XP in dört temel noktasından en zoru cesarettir. Projelerin üzerine yılmadan gidilmesi projelerin geliştirilmesi açısından son derece önemlidir. Cesaretin olmadığı projelerde korku gelişir ve gelişen

bu korku projeyi başarısızlığa iter. Yazılım işlerindeki başarısızlık ise; yazılımın çöpe gitmesidir ve maalesef genel duruma da baktığımızda yazılımların büyük bir kısmının çöpe gittiğini görüyoruz. Başarısızlık, genel tabiat içerisinde olan bir durumdur. Başarısızlıktan korkmak yerine başarısızlığı oluşturan nedenler üzerine gitmek yerinde olacaktır. Başarısızlıkla mümkün olunan en kısa sürede karşılaşmak, daha sonra telafi etme şansını artırır.

4.1.3 Extreme Programming Prensipleri



Şekil 5. XP Prensipleri

4.1.3.1 Rapid Feedback (Hızlı geri bildirim)

Sık ve hızlı geri bildirim edinmek, projenin gidişatını olumlu etkiler. Geri bildirim sayesinde yanlış anlaşılımlar ve hatalar ortadan kaldırılır.

4.1.3.2 Assume Simplicity (Basitliği tercih etmek)

Basit çözümler kolay implemente edilir ve kısa zamanda oluşturulur. Bu geri bildirimin de hızlı bir şekilde gerçekleşmesini sağlar. Basit çözümlerin kavranması ve anlatılması daha kolaydır.

4.1.3.3 Incremental Change (İnkrementel değişiklik)

Basit çözümler uygulasa bile, yazılım sistemleri zaman içinde karmaşık bir yapıya dönüşebilir. Yapılan en ufak bir değişiklik bile sistemin düşünmediğimiz bir bölümü üzerinde hata oluşmasına sebep verebilir. Oluşabilecek bu hataları kontrol altında tutabilmek için değişikliklerin ufak çapta olması gerekmektedir. Büyük değişiklikler beraberinde büyük sorunları getirebilir. Bu sebepten dolayı değişikliklerin ufak çapta ve sıklıkla yapılması gerekmektedir.

4.1.3.4 Embracing Change (Değişimi istemek)

İlerleyebilmek için kendimize bir yön tayin etmemiz ve yeniliklere açık olmamız gerekiyor.

4.1.3.5 Quality Work (Kaliteli iş)

XP projelerinde kaliteli işin ortaya konabileceği bir ortamın oluşturulması gerekmektedir. Hiçbir programcı hatalı program yazmak istemez. Çalışma ortamının etkisiyle yüksek kalitede yazılım

yapmak hem programcının öz güvenini artırır hem de müşteriye tatmin edici ürünlerin ortaya konmasını sağlar.

4.1.3.6 Teach Learning (Öğrenmeyi öğret)

XP programcı takımlarında tertipçilik ve kıdem farkı yoktur. Tecrübeli programcılar bilgilerini daha az tecrübeli programcılarla paylaşarak, hem bilginin çoğalmasını sağlarlar hem de takım arkadaşları ile teknik olarak aynı seviyeye gelirler.

4.1.3.7 Small Initial Investment (Az başlangıç yatırımı)

XP en modern ve pahalı araç gereçlerle projeye başlanmasını beklemez. Başlangıç giderleri ne kadar düşük tutulabilirse, projenin iptali durumunda kayıplar o oranda az olacaktır. Başlangıçta tüm takımın dar bir finansman korsesi giymesi sağlanarak, proje için daha önemli görevlere odaklanmaları sağlanır.

4.1.3.8 Play to win (Kazanmak için oyna)

XP takımları kazanmak için oynar. Her zaman gözlerinin önünde nihai sonuç vardır: programı tamamlamak ve müşteriye teslim etmek.

4.1.3.9 Concrete Experiments (Somut denemeler)

Verdiğimiz kararların sonuçlarını kontrol edebilmek için denemeler yaparız, çünkü alınan kararlar her zaman doğru olmayabilir. Bir kontrol mekanizmasına ihtiyacımız olduğu belli. Bu da somut denemeler aracılığıyla nerede olduğumuzu tespit etmekten geçer. Bu somut denemeler yazılım sistemleri içinde geçerlidir. Örneğin testler hazırlayarak, oluşturduğumuz mimari ve tasarımı kontrol ederiz.

4.1.3.10 Open, honest Communication (Açık ve samimi iletişim)

Projenin başarılı olabilmesi için bireyler arasında açık ve samimi türde bir iletişim olması gerekmektedir. Birçok projede bu böyle değildir. Çoğu zaman bireylerin korkuları, deneyimsiz olmaları ya da kendilerini çok beğenmeleri ve diğerlerini kendilerinden alt safhada görmeleri, açık ve samimi bir iletişim ortamının oluşmasını engeller.

4.1.3.11 Work with people's instincts, not against them (Takımın içgüdülerini kullan, onlara karşı koyma)

Bireysel içgüdü yanı sıra, bireylerin oluşturduğu takımların da içgüdüğü vardır. Eğer takım bir şeylerin doğru gitmediği hissine sahipse ve bunu dile getiriyorsa, o zaman bir şeyler yolunda gitmiyor demektir.

4.1.3.12 Accepted Responsibility (Sorumluluk üstlenmek)

Sorumluluk birilerine verilmemeli, bireyler kendileri sorumluluk üstlenmelidir.

4.1.3.13 Local Adaptations (Sürecin ortam şartlarına adapte edilmesi)

Büyük bir ihtimalle her takımın XP yi Kent Beck'in anlaştığı tarzda harfiyen uygulaması mümkün değildir. Atalarımızın da dediği gibi her yiğidin yoğurt yiyiş tarzı farklıdır. Amaç XP yi harfiyen uygulamak değildir, amaç kısa bir zamanda projeyi başarılı bir sonuca ulaştırmaktır.

4.1.3.14 Travel light (Az yükle yolculuk yapmak)

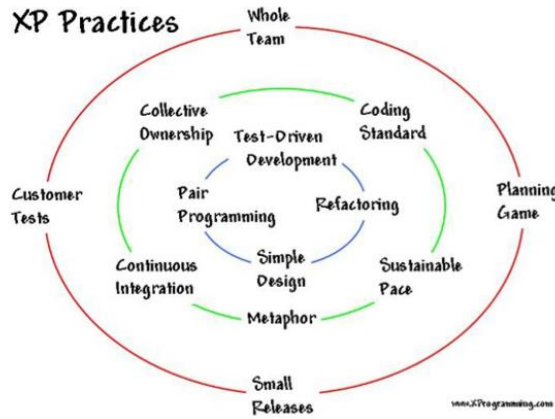
Projede hızlı ilerleyebilmek için fazla bir yükle yola çıkılmaması gerekmektedir. Beraber çalışmayı kolaylaştırmak için kullanımı kolay araç ve gereçler seçilmelidir.

4.1.3.15 Honest Measurement (Doğru ölçüm)

Proje gidişatını kontrol edebilmek için değişik türde ölçümlerin yapılması gerekmektedir. Örneğin hazırlanan birim testleri ile sınıfların işlevleri kontrol edilir.

4.1.4 XP Teknikleri (XP Practices)

Dört XP değer ve on beş XP prensibi on dört XP tekniği ile desteklenmektedir. XP teknikleri programcıların XP değer ve prensiplerini uygulamada yardımcı olur. Kent Beck tarafından hazırlanan ilk XP versiyonunda on iki teknik yer almaktaydı. Diğer çevik süreçlerin de etkisiyle Standup-Meeting ler ve retrospektif toplantılar XP teknikleri arasına katılmıştır.



Şekil 6. XP Teknikleri

4.1.4.1 On-site Customer (Programcıya yakın müşteri)

XP projeleri müşteri gereksinimlerine odaklı ilerler. Bu yüzden müşteri ve sistem kullanıcılarının projeye dahil edilmeleri gerekmektedir. Müşteri gereksinimlerini ekibe bildirir. Programcıların implementasyonu gerçekleştirebilmesi için müşteri tarafından dile getirilen gereksinimleri anlamaları gerekmektedir. Yanlış anlaşılmalrı ve hataları gidermek için programcıların müşteri ve sistem

kullanıcıları ile diyalog halinde olabilmesi gerekmektedir. Bu sebepten dolayı müşteri veya sistem kullanıcılarının programcıların erişebileceği bir uzaklıkta olmaları gerekir. Tipik XP projelerinde müşteri ve programcılar aynı odada beraber çalışabilirler.

4.1.4.2 Standup-Meeting (Ayakta toplantı)

Proje çalışanları her gün 15 dakikayı aşmayan ve ayakta yapılan toplantılarda bir araya gelirler. Bu toplantının amacı, projenin gidişatı hakkında bilgi alışverişinde bulunmaktır.

4.1.4.3 Planning Game (Planlama oynu)

XP projeleri iteratif ve inkrementel yol alır. Bir sonraki iterasyonda yapılması gereken işleri planlama oyununda görüşülür ve sürüm ve iterasyonun içeriği tespit edilir. Planlama oyununa müşteri, kullanıcılar ve programcılar katılır. Müşteri ve kullanıcılar daha önce kullanıcı hikayesine (user story) dönüştürdükleri isteklerine öncelik sırası verirler. Programcılar her kullanıcı hikayesi için gerekli zamanı tahmin ederler. Kullanıcı hikayelerinin öncelik sırası bu tahmine bağımlı olarak değişebilir. Planlama oyunlarında sürüm ve iterasyon planları oluşur.

4.1.4.4 Short Releases (Kısa aralıklarla yeni sürüm)

XP projelerinde yeni implemente edilen ve değişikliğe uğrayan komponentler yeni sürümler oluşturularak müşteri ve kullanıcının beğenisine sunulur. Bu sayede hem müşteriler çalışır durumda olan programdan faydalanabilir hem de yeni sürümü inceleyerek, gereksinimleri ile örtüşüp, örtüşmediğini kontrol edebilirler. Eğer yeni sürüm müşteriyi tatmin edecek durumda değilse, gereksinimler değişikliğe uğrayabilir. Bu değişiklikler bir sonraki iterasyonda göz önünde bulundurularak, müşteri istekleri ile yüksek derecede örtüşen bir programın oluşturulması sağlanır.

4.1.4.5 Retrospective (Geriye bakış)

Proje çalışanları düzenli aralıklarla geriye bakarak, meydana gelen sorunları gözden geçirirler. Buradaki amaç gelecekte bu sorunların tekrarını önlemektir. Geriye bakış bir ile altı aylık zaman birimleri için tüm proje çalışanları ya da seçilen bireyler tarafından yapılır. Geriye bakış toplantıları yarım gün ile üç gün arasında sürebilir.

4.1.4.6 Metaphor (Mecaz)

XP projelerinde hazırlanan program için bir veya birden fazla, programın nasıl bir işlevi olacağını ekibin gözünde canlandırmalarını sağlayacak mecazi isim, öge ya da resimler kullanılır. Bunlar proje çalışanlarının ortak bir payda da buluşarak, ne yapılması gerektiği hakkında bir fikir sahibi olmalarını kolaylaştırır.

4.1.4.7 Collective Ownership (Ortak sorumluluk)

XP projelerinde programcılar ortak sorumluluk taşırlar. Bu her kod parçasının herhangi bir programcı tarafından gerekli durumlarda değiştirilebileceği anlamına gelir. Böylece yapılması gereken işler aksamaz, çünkü belli kod bölümlerinden belli programcılar sorumlu değildir. Aksine her programcı programın her bölümü üzerinde çalışma hakkına sahiptir. Bir programcının işe gelmemesi durumunda, başka bir programcı kolaylıkla onun görevlerini üstlenebilir.

4.1.4.8 Continuous Integration (Sürekli entegrasyon)

Sistem değişiklikleri ve yeni komponentler hemen sisteme entegre edilerek test edilir. Sürekli entegrasyon sayesinde yapılan tüm değişiklikler her programcının sistem üzerinde yapılan değişiklikleri görmesini sağlar. Ayrıca sistem entegrasyonu için gerekli zaman azaltılır, çünkü oluşabilecek hatalar erken teşhis edilerek, ortadan kaldırılır.

4.1.4.9 Coding Standards (Kod standartları)

Programcılar tarafından aynı kalitede kod yazılımı yapılabilmesi için kod yazarken kullanılacak kuralların oluşturulması gerekmektedir. Kodun nasıl formatlanacağı, sınıfların, metot isimlerinin ve değişkenlerin nasıl isimlendirileceği kod standartlarında yer alır.

4.1.4.10 Sustainable Pace (Kalıcı tempo)

XP projelerinde programcılar haftalık belirli mesai saatlerini aşmazlar. Gereğinden fazla çalıştırılan ve yorulan bir programcıdan verimli iş yapması beklenemez. Programcılarının motivasyonun ve çalışma enerjilerinin yüksek olması için günde sekiz saatten fazla çalışmalarına izin verilmemelidir. Bazen fazla mesai saatlerine ihtiyaç olabilir. Eğer durum devamlı böyle ise, bu proje gidişatında bazı olumsuzlukların göstergesi olabilir.

4.1.4.11 Testing (Test etmek)

Oluşturulan programların kalite kontrolünden geçmesi gerekmektedir. Bu yazılım esnasında oluşturulan testlerle yapılır. Programcılar komponentler için birim testleri hazırlar. Sınıf bazında yapılan bu testlerle komponentlerin işlevleri kontrol edilir. Müşteri gereksinimlerini test etmek için onay/kabul testleri hazırlanır. Komponentlerin entegrasyonunu test etmek için entegrasyon testleri hazırlanır.

4.1.4.12 Simple Design (Sade tasarım)

Programcılar üstlendikleri görevleri (task) en basit haliyle implemente ederler. Bu programın basit bir yapıda kalmasını ve ilerde değiştirilebilir ve genişletilebilir olmasını sağlar. Sade bir tasarım yazılım sisteminin karmaşık bir yapıda olmasını önler. Bunun yanı sıra basit tasarımlar daha kolay ve daha hızlı implemente edilebilir. Basit bir implementasyonu anlamak ve anlatmak daha kolaydır.

4.1.4.13 Refactoring (Yeniden düzentme)

Tasarım hataları yazılım sisteminin daha ilerde tamir edilemeyecek bir hale dönüşmesine sebep verebilir. Bu yüzden bu hatalar hemen giderilir. Hazırlanan birim testleri ile yapılan değişikliklerin yan etkileri kontrol edilir. Bu açıdan bakıldığında birim testi olmayan bir sistem üzerinde yeniden yapılandırma işlemi hemen hemen mümkün değildir, çünkü değişikliklerin doğurduğu yan etkileri tespit etme mekanizması bulunmamaktadır.

4.1.4.14 Pair Programming(Eşli çalışma)

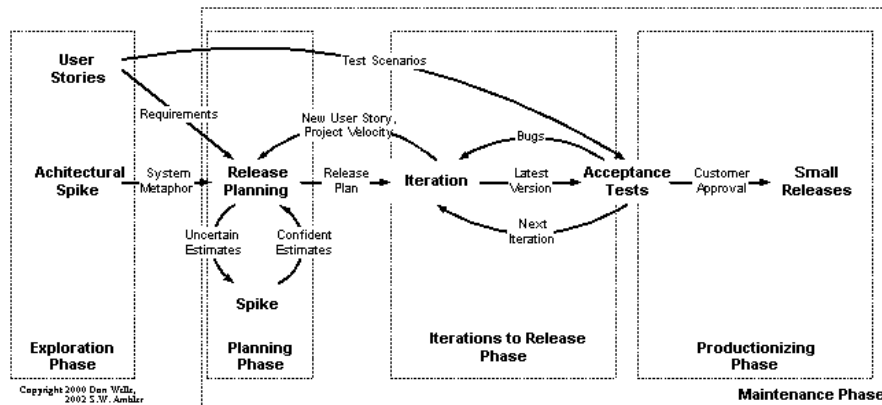
XP projelerinde iki programcı aynı bilgisayarda çalışır. Bu sayede programcıların kısa bir zaman içinde aynı seviyeye gelmesi sağlanır. Ayrıca bu kalitenin yükselmesini sağlar.

4.1.5 Extreme Programing Avantajları

- ❖ **Hata oranını azaltma:** Yazılımın erken safhasında somut gelişmeler sağlayacağından, oluşan hataların farkına varabilir. Bu hataları da kendi içerisinde oluşturduğu küçük yaşam döngüleri ile telafi edebilir.
- ❖ **Projenin gelişme süresini kısaltır:** Artırımsal yaklaşım sayesinde hızlı bir şekilde genel planın oluşması sağlanır. Burada aynı bütün ekip tarafından proje süresi tahmini de yapılır.
- ❖ **Değişikliklere izin verir:** Esnek iş yürütme ve fonksiyonellik sayesinde, işletmenin değişen ihtiyaçlarına cevap verir.
- ❖ **Yazılımsal olarak ortaya çıkabilecek tıkanıklıkları azaltır:** Müşteriler ile birlikte monitörlerden programcılar ve müşteriler ortak bir şekilde test yaparak ileride tıkanıklıkların ortadan kaldırılmasını sağlarlar.

4.1.6 XP Proje Safhaları

Bir XP projesi değişik safhalardan oluşur. Her sahfa, bünyesinde kendine has aktiviteler ihtiva eder. Bir sonraki resimde bir XP projesinde olması gereken safhalar yer almaktadır.



Şekil 7. XP Proje Safhaları

XP projesi şu safhalardan oluşur:

4.1.6.1 Keşif safhası (Exploration Phase)

Projenin başlangıcında keşif safhasını oluşturan aktiviteler yer alır. Bu safhada müşteri kullanıcı hikayelerini (user story) oluşturur. Programcılar teknik altyapı için gerekli deney (spike) ve araştırmayı yaparlar.

4.1.6.2 Planlama Safhası (Planning Phase)

Keşif safhasını planlama safhası takip eder. Bu safhada müşteri programcılar yardımıyla iterasyon ve sürüm planlarını oluşturur. İterasyon planlaması için oluşturulan kullanıcı hikayelerinin implementasyon süresi programcılar tarafından tahmin edilir. Müşteri kullanıcı hikayelerine öncelik sırası vererek, iterasyonlarda hangi kullanıcı hikayelerinin öncelikli olarak implemente edilmeleri gerektiğini tespit eder. Programcılar tarafından herhangi bir kullanıcı hikayesinin implementasyon süresi tahmin edilemese, programcılar spike solution olarak bilinen basit bir çözüm implemente ederek, kullanıcı hikayesinin gerçek implementasyonu için gerekli zamanı tahmin etmeye çalışırlar.

4.1.6.3 İterasyon ve Sürüm Safhası (Iterations to Release Phase)

Kullanıcı hikayelerinin implementasyonu iterasyon ve sürüm safhasında gerçekleşir. Bir iterasyon bünyesinde implemente edilmesi gereken kullanıcı hikayeleri müşteri tarafından belirlenir. Implementasyonunun işlevini kontrol etmek için müşteri tarafından akseptans testleri belirlenir. Bu testler programcılar yada testçiler tarafından implemente edilir. Her iterasyon sonunda müşteriye, çalışır bir yazılım sistemi sunulur. Bu şekilde müşterinin sistem hakkındaki görüşleri alınır (geridönüm). İterasyon son bulduktan sonra çalışma hızını tahmin etmek için bir önceki iterasyonunda elde edilen tecrübeler kullanılır ve iterasyon planı bu değerler doğrultusunda gözden geçirilir. Bir önceki iterasyonda oluşan hatalar bir sonraki iterasyonda gözden geçirilmek ve giderilmek üzere planlanır.

4.1.6.4 Bakım Safhası (Maintenance Phase):

Bu programın bakımının ve geliştirilmesinin yapıldığı safhadır. Bu safhada kullanıcılar için eğitim seminerleri hazırlanır ve küçük çapta eklemeler ve sistem hatalarının giderilmesi için işlemler yapılır. Müşterinin istekleri doğrultusunda bir sonraki büyük sürüm için çalışmalara başlanır. Bu durumda tekrar keşif safhasına geri dönülmesi ve oradan işe başlanması gerekmektedir.

4.2 Feature Driven Development

FDD, yazılım geliştirme aşamalarında sadece tasarım ve geliştirme süreçlerini etkileyen çevik ve adaptasyonu yüksek bir yazılım geliştirme yöntemidir. Ana aktiviteleri beş bölümde özetlenebilir:

- Tüm sistemi modelleme,
- Özellik listesi hazırlama,
- Özellikleri planlama,
- Özelliğe göre tasarım,
- Özelliğe göre geliştirme

Sistemin modelleme safhasında, takım üyeleri ve varsa danışmanlar sistem için bir “yol planı” oluştururlar. İkinci safhada, yazılım ürününde ihtiyaç olan özellikler tüm takım üyeleri ve uzmanların anlayabileceği basitlikte parçalara ayrılır. Üçüncü ve son aşamada “tasarım paketleri” adı verilen parçalar önceliklendirilir ve her bir parça, programcılardan sorumlu bir şef programcıya atanır. Yazılım parçacığının özelliğine göre tasarım aşamasında sürecin yinelemeli kısmı başlar. Şef programcı kendisine atanmış tasarım paketinden 1-2 hafta içerisinde tamamlayabileceği bir alt iş seçer. Geliştirme aşamasında bu alt iş yeniden ayrıntıları ile analiz edilir, tasarlanır, kodlanır, test edilir ve entegre edilir (Palmer & Felsing, 2001). FDD, diğer çevik yöntemlerle kıyaslandığında, büyük ve kritik projelerde kullanılmasının oldukça uygun olduğu söylenebilir nitekim tüm yazılım geliştirme süresi boyunca kaliteye ağırlık verilir ve kaliteden hiçbir müddetçe taviz verilmez.

FDD yönteminde, yazılım geliştirmeler tamamlandıktan sonra birim testi (Unit Test) ve kod inceleme (Code Review) safhalarına geçilir. Hangi testin önce yapılacağı şef programcı tarafından karar verilir. Birim testin, manuel mi veya otomasyon aracılığı ile yapılacağına karar verilir. Kod inceleme FDD için zorunlu bir kuraldır ve bunun için yazılım geliştirme süresinin dörtte biri kadar zaman ayrılır. Kod inceleme, iyi bir biçimde yapıldığında, kod içinde ki yazılım kusurları tespit edilebilmekte ve ileride yaşanabilecek karmaşıklıklarının önüne geçilebilmektedir (McConnell, 2004). Bu inceleme safhası sadece yazılım geliştiriciler tarafından yapılmakla kalmayıp, takım içindeki tecrübeli programcılar tarafından da bilahare yapılmaktadır. Bu durum ekibe yeni katılmış az tecrübeli programcılarının gelişmesine de yardımcı olmaktadır. Ayrıca proje kapsamında yazılan kodların başka bir programcı tarafından incelendiğini bilmek, daha dikkatli ve kod standartlarına sadık kalarak uygun kod yazılmasını da sağlar. Kod incelemenin seviyesi, yinelemede geliştirilen özelliğin etkisi ve karmaşıklığına göre şef programcı tarafından belirlenir. Önem ve karmaşıklık derecesine bağlı olarak daha az öneme sahip işler takım tarafından incelenmesi yeterli iken, daha karmaşık ve önemli işler, şef 14 programcılar ve bazen de diğer takımlardan programcılar ile incelenmektedir. Bu sayede XP programlama da yapılan iki kişi programlama pratiğinden daha iyi sonuçlar elde edilir. Bu noktada programcıya koda bir süre sonra tekrardan göz atma şansı verilmekle birlikte, kodlara farklı insanların bakması ve fikir yürütmesi ile çok daha kaliteli, hızlı ve doğru

kodların elde edilmesi mümkün olabilmektedir. Tüm bu aşamalar maruz kalan kod, entegrasyona hazır hale gelmektedir (Flora ve Chande, 2014).

4.3 Adaptive Software Development

ASD, çok hızlı ve değişken bir yapıda olan internet ekonomisi için geliştirilmiş yazılım geliştirme yöntemidir. Çok hızlı ve değişimi fazla olan projeler, tahmin edilmesi zor ve geleneksel yazılım geliştirme yöntemleri ile yönetilemeyecek kadar karmaşık bir yapıya sahiptir. Bu nedenle yazılım geliştirme yapısının adaptasyonlu olması, sürekli değişime maruz kalan bölümlerde hızlı bir biçimde cevap vermesi gerekir. ASD sürekli prototip geliştirmesi yaparak yinelemeli ve artırimsal geliştirmeyi teşvik eder. Yöntemde “kaosun sınırında dengeleme” mantığı ile, “yeterli kılavuzluk ile projelerin kaosa düşmesi engellenebilir” düşüncesi temeldir. Bu yöntemde proje üç safhada değerlendirilir:



Tahmin etmek



İş birliği yapmak



Öğrenmek

Belirsizliklerin az olduğu durumlarda plan yapmak yerine, bu yöntemde tahmin yapılması tercih edilir.

Birlikte çalışma hızla değişen sistem geliştirmede oldukça önemlidir. Hatalar üzerinden bilgi üretme ve geliştirme sırasındaki ihtiyaç değişimlerini sağlamak öğrenme fazı olarak tanımlanmaktadır.

Proje başlama, adaptasyonlu çevrim planlama, eşzamanlı özellik geliştirme, kalite gözden geçirme ve final kalite güvence-sürüm yayınlama olmak üzere beş genel adımdan oluşur. Kalite gözden geçirme ile çevrim planlama arasında öğrenme geri beslemesi bulunmaktadır (Highsmith, 2000).

Müşterinin (iş sahibi) geliştirme aşamasında takımla birlikte olması nedeniyle Ortak Uygulama Geliştirme ((Joint Application Development [JAD])) temel olarak alınmaktadır. Müşteri ile birlikte çalışma kabul test sürecinin daha doğru ve kolay geçilmesini sağlar. Bilinirliği az olan veya ilk kez yapılacak işler ve araştırma geliştirme çalışmaları için bu yöntemdeki öğrenme geri beslemesi oldukça fazla önem arz etmektedir. Bu geri besleme ile birlikte kabul testi gerçekleştirilmiş olup hem müşteri bakış açısından hem de teknik açıdan incelemeler yapılmış olur. Bu incelemeler ile birlikte takım kendi performansını değerlendirme şansı bulur. Ayrıca JAD toplantılarında tüm paydaşlarının projenin son durumunu anlık olarak gözden geçirme fırsatı bulur (Abrahamsson vd., 2002; Highsmith, 2000).

4.4 Dynamic Systems Development Method

Dynamic Systems Development Method, 1995 yılında sektörde önde gelen proje uzmanları tarafından Hızlı Uygulama Geliştirme (Rapid Application Development [RAD]) için kaliteyi hedeflemek amacıyla geliştirilen ve 1995’de Stapleton ve Millington tarafından tanımlanan bir çerçevedir (Flora ve Change, 2014)

İlk çevik yöntem olarak ortaya çıkan bu çerçeve, proje ürün yönetimi yaşam döngüsünün en başarılı süreçlerini bir araya getiren yinelemeli ve artırımı bir metodolojidir. Sonuçlara efektif ve hızlı bir biçimde ulaşmayı amaçlayarak, maliyet, risk, zaman ve kaliteyi kontrol altında tutup iş faydalarını artırimsal olarak ulaşmayı sağlamak bu yöntem için stratejik bir amaçtır. Takımın sürekli güçlendirilmesi, kaliteyi sürekli iyileştirmek yerine sürümlerin hızlı ulaştırılmasını sağlamak, geliştirme esnasında herhangi bir değişikliği anında geri alınabilmesi, en üst seviye ihtiyaçlarının değişmemesi, iyi haberleşme ve tüm paydaşlarının birlikteliği ile tüm proje süreci boyunca sürekli test yapılması bu çerçevenin temel prensipleridir. Bu yöntemde kullanılan temel teknikleri arasında, Zaman kutuları, Prototipleme, Test, İmalat ve Modelleme gösterilebilir. Bu çerçeve kapsamında ön-proje, fizibilite çalışması, iş çalışması, fonksiyonel model yineleme, tasarım ve geliştirme yinelemesi, uygulama ve proje sonu aşamaları gerçekleştirilir (Millington ve Stapleton, 1995; Cohen vd., 2004).

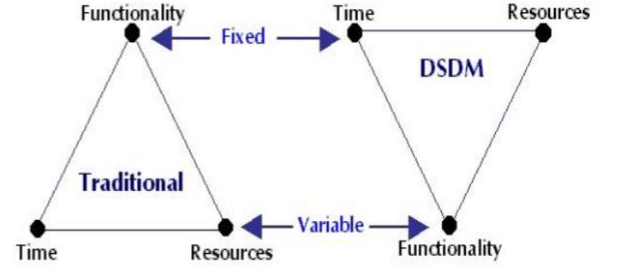
Fonksiyonel model tekrarlama, işin doğru yapıldığını ispat etmek için dokümanlar gözden geçirilip, prototipler sergilenerek yazılım parçasının başarılı bir şekilde test edildiği kanıtlanır. Bu noktada ürünün fonksiyonelliğe katkısı üzerinde durulur. Fonksiyon olmayan kısımlar ise tasarım ve gerçekleştirme aşamalarında test edilir. Kabul 15 testlerinin zaman kaybetmeden hızlıca geçilebilmesi için kullanıcıların aktif katılımı gereklidir. DSMS test araçlarının kullanımını tavsiye etmektedir. Yakalama ve yeniden oynatma (capture and replay) araçları ile testlerin yapılması, test doğruluklarının kanıtlanması içinde başarılı bir yol olmakla birlikte, kâğıt üzerinden takibinden daha az efor harcayan bir işlem olarak kabul edilir. Kod gözden geçirme (Code Review) DSDM’de önerilen bir pratiktir. Dinamik analiz araçları, demo sırasında dizi boyutları, hafıza kullanımı gibi hususları test etmek için kullanılabilmektedir (Millington ve Stapleton, 1995).

4.4.1 DSDM ayırt edici özellikleri :

- Pek çok yönden XP ve/veya ASD’ye benzer
- Dokuz yönlendirici prensip
- Kullanıcıların aktif katılımı önemlidir.
- DSDM takımları karar vermede yetkilidirler
- Sık teslim edilen ürünlere odaklanılır.

- Teslim edilebilirliğin kabul kriteri olarak iş amacına uygunluk kullanılır
- Tekrarlanan (iterative) ve artımlı (incremental) geliştirme doğru iş sonucuna ulaşmak için gereklidir.
- Geliştirme sırasında gerçekleştirilen tüm değişiklikler geri alınabilir..
- Gereksinimler yüksek seviyede temellendirilirler
- Test tüm yaşam döngüsü boyunca entegredir.

- 🎨 Basit
- 🎨 Genişletilebilir
- 🎨 Düz ileri Framework tabanlı
- 🎨 Tüm proje türlerinde çözüm sağlayıcı değildir



Şekil 8. DSDM

4.4.2 Dinamik Sistem Geliştirme Proje Yapısı

DSDM projesi, organize rolleri, gömülü rolleri ve sorumlulukları ile zengin bir küme ve çeşitli çekirdek teknikleriyle desteklenen 7 aşamalı adımdan oluşur.

- Roller ve Sorumluluklar
- Takım Organizasyonu ve Boyutu
- 7 Aşamalı kurallar. Bunlar:
 1. Ön Proje
 2. Fizibilite Çalışması
 3. İş Çalışması
 4. Fonksiyonel Model Yineleme
 5. Tasarım & Yapı Yineleme
 6. Uygulama
 7. Proje Sonrası

4.4.3 Dinamik Sistem Geliştirme Süreci

The DSDM Development Process



Şekil 9. Dinamik Sistem Geliştirme Süreci

4.5 Agile Unified Process

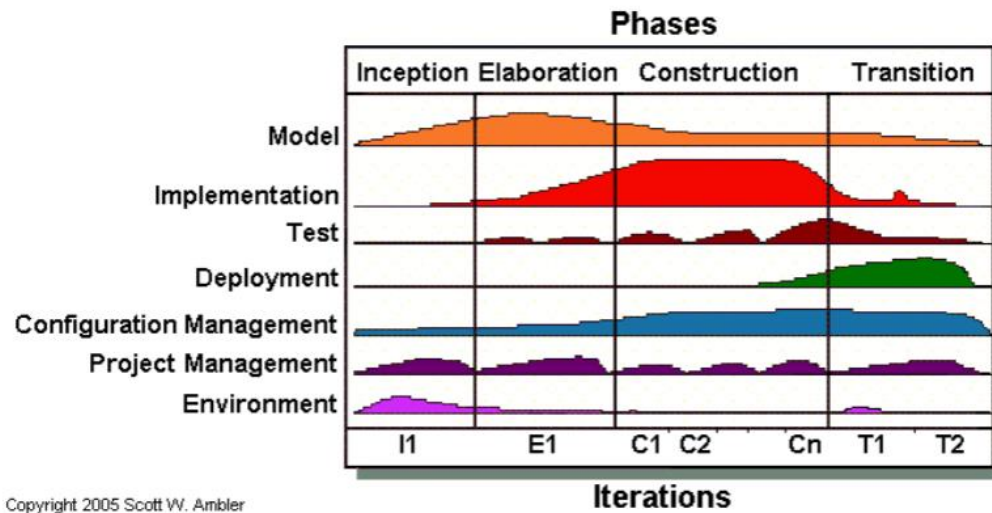
Çevik Tümlleşik Süreç (AgileUnifiedProcess–AUP’); RationalUnifiedProcess(RUP)’un, çevik yaklaşıma göre adapte edilerek basitleştirilmiş halidir.

Test-güdümlü geliştirme (“test driven development–TDD”), çevik deęişiklik yönetimi, veri tabanını yeniden yapılandırma gibi çevik pratikleri uyguladır.

RUP’dan farklı olarak, 7 adet disiplin içerir.

4.5.1 AUP Yaşam Döngüsü: 7 Disiplin

AUP'nin yedi (7) disiplini ve bu disiplinlerin dört (4) yinelemesi vardır. Toplamda dokuz (9) aşamaya sahip olan RUP'nin aksine, AUP'nin yaşam döngüsü iki disiplin grubuna ayrılmamıştır. Bunun yerine, yedisinin tümü AUP yaşam döngüsünün aşamaları olarak sayılır. AUP'nin temel özellikleri olan işleri basit ve anlaşılır tutar. Bu yedi aşama:



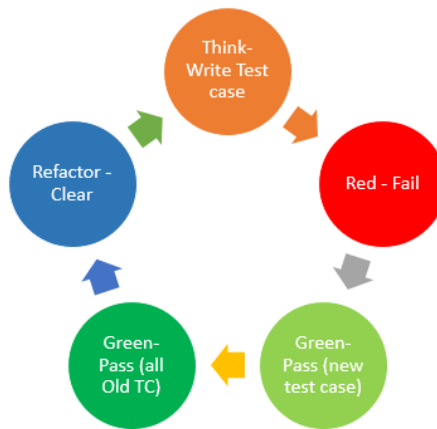
Şekil 10. AUP Yaşam Döngüsü: 7 Disiplin

- **Model:** Bu disiplinin amacı, kuruluşun işini, proje tarafından ele alınan sorun alanını anlamak ve sorun alanını ele almak için uygulanabilir bir çözüm belirlemektir.
- **Uygulama:** Bu disiplinin amacı, model (ler) inizi çalıştırılabilir koda dönüştürmek ve özellikle birim testi olmak üzere temel düzeyde bir test yapmaktır.
- **Test edin:** Bu disiplinin amacı kaliteyi sağlamak için objektif bir değerlendirme yapmaktır. Bu, kusurları bulmayı, sistemin tasarlandığı gibi çalıştığını doğrulamayı ve gereksinimlerin karşılandığını doğrulamayı içerir.

- **Dağıtım:** Bu disiplinin amacı, sistemin teslimini planlamak ve sistemi son kullanıcıların kullanımına sunmak için planı uygulamaktır.
- **Yapılandırma Yönetimi:** Bu disiplinin amacı, proje eserlerinize erişimi yönetmektir. Bu, yalnızca yapı sürümlerini zaman içinde izlemeyi değil, aynı zamanda bunlardaki değişiklikleri kontrol etmeyi ve yönetmeyi de içerir.
- **Proje Yönetimi:** Bu disiplinin amacı, proje üzerinde yer alan faaliyetleri yönlendirmektir. Bu, riskleri yönetmeyi, insanları yönlendirmeyi (görevleri atama, ilerlemeyi izleme vb.) Ve zamanında ve bütçe dahilinde teslim edildiğinden emin olmak için proje kapsamı dışındaki kişiler ve sistemlerle koordinasyonu içerir.
- **Çevre:** Bu disiplinin amacı, uygun süreç, rehberlik (standartlar ve yönergeler) ve araçların (donanım, yazılım, vb.) Gerekliğinde ekip için mevcut olmasını sağlayarak çabanın geri kalanını desteklemektir.

4.6 Test-Driven Development

Kent Beck tarafından Extreme Programming'in bir parçası olarak bulunmuş olan bir programlama tekniğidir. Tarih açısından bakacak olursak 1994 yılında Beck, Smalltalk için SUnit test framework'ünü yazmıştır. 1998 yılında XP'de "testleri çoğunlukla ilk sırada yazmalıyız" diye ifade ettiği Test First kavramını ortaya artmıştır. 2002 Kasım'ında piyasaya çıkan Test Driven Development: By Example isimli kitabında Test-First kavramını Test Driven Development olarak daha da detaylı bir teknik olarak ele alınmıştır. Üzerinden 15 yıl geçmiş olmasına rağmen TDD öğrenmek isteyenlere hala bir başucu kılavuzu olarak yer almaktadır.



Şekil 11. Test-Driven Development

Resimde göreceğimiz üzere TDD çok basit birkaç adımdan oluşmaktadır.

1. Bir test yazılır.
2. Test başarısız olur.
3. Test başarılı hale getirilir.
4. Mevcut bütün testlerin başarılı olması sağlanır.
5. Kod refactor edilir. Yani kodda iyileştirme ve(ya) temizleme yapılır.

Bu bağlamda önceden kodu yazıp sonradan bunların testini yazmanın TDD olamadığını özellikle belirtmek gerekmektedir. TDD’de mutlaka ama mutlaka test ilk sırada yazılmalıdır. Tabii her ekip yada her yazılımcı TDD yapmak zorunda değildir. Mevcut kodları için test yazılmasının da bir o kadar önemli olduğunu eklemek önemli olacaktır.

Peki test yazmak önemli ise testleri önceden yazmak ile sonradan yazmak yazılımcıya nasıl bir katma değer katacaktır?

Her ne kadar TDD’yi Test Driven Development olarak açsakta bir çok üstat bunu Test Driven Design olarak da ifade etmektedir.

Geleneksel uygulama geliştirme tekniklerinde yazılımcının önüne bir ister geldiğinde “bottom to top approach” diye ifade ettiğimiz tekniği kullanır. Yazılımcının katmanlı mimari ile geliştirme yaptığını ele alırsak. Katmanlı mimarinin en alt katmanlarından başlayarak en üst katmana doğru geliştirme yaklaşımı olarak tanımlayabiliriz.

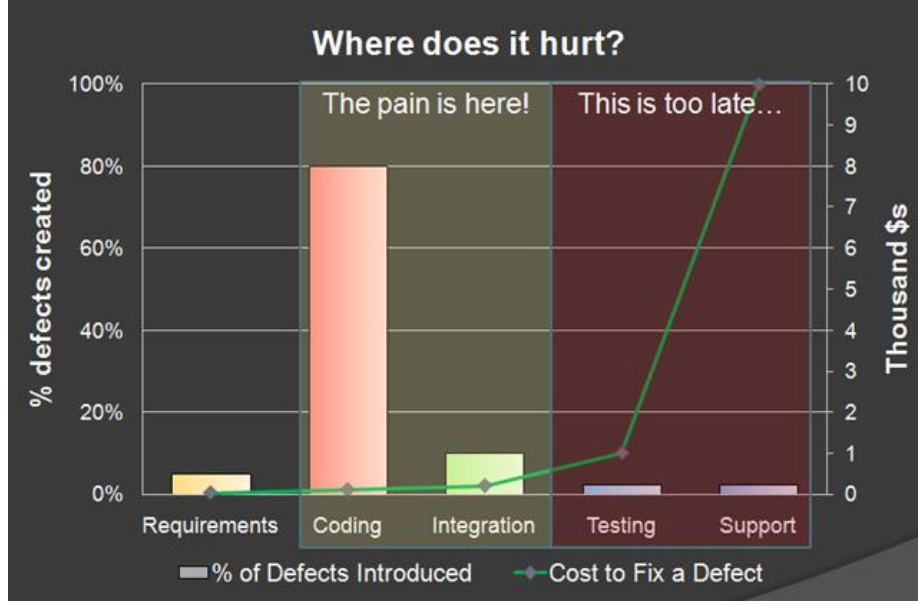
Örnek olarak, bir ödeme sistemi tasarlanmak istendiğinde yazılımcı

1. Önce veritabanını düşünür, gerekli tabloları ve bunların kolonları tasarlar. Mesela isterde sadece kredi kartı ile ödeme yapıldığını analiz edilmesine rağmen ileride başka ödeme tipleri de gelebileceğini düşünerek o ödeme tipleri içinde alanlar oluşturdu.
2. Sonra 3 katmanlı mimari de bir üste çıkarak İş Katmanını kodlar
3. Ve en son olarak Sunum Katmanını kodlar.
4. Uygulamayı Sunum Katmanı üzerinden test eder.

Eğer hangi katmanda olursa olsun bir hata çıkarsa hata giderildikten sonra gene Sunum katmanından manuel test yapılmaya devam eder.

Ama mevcutta sadece kredi kartı ile ödeme olmasına rağmen diğer ödeme tipleri de test edildi. Bu bağlamda kodda ihtiyaç olmamasına rağmen fazladan kodlar bulunmuş oldu ve kodumuz gereksiz karmaşık aldı.

Aşağıdaki resimde bir bug’ın maliyetinin SDLC’de hangi fazda ne kadar olduğu görülmekte.



Şekil 12. Bug'ın maliyetinin SDLC'de hangi fazda ne kadar olduğu görülmekte

Bu bağlamda bug'ları uygulama PROD'a çıkmadan bulup, çözebilirsek maliyetleri de o kadar azaltabilme şansına sahip oluruz.

TDD'nin sunduğu en önemli katma değer KISS (Keep It Simple Stupid) prensibidir. Yani her şeyi basit tutmalıyız. Bu yazılan kodlar ve hatta mimari için de geçerlidir.

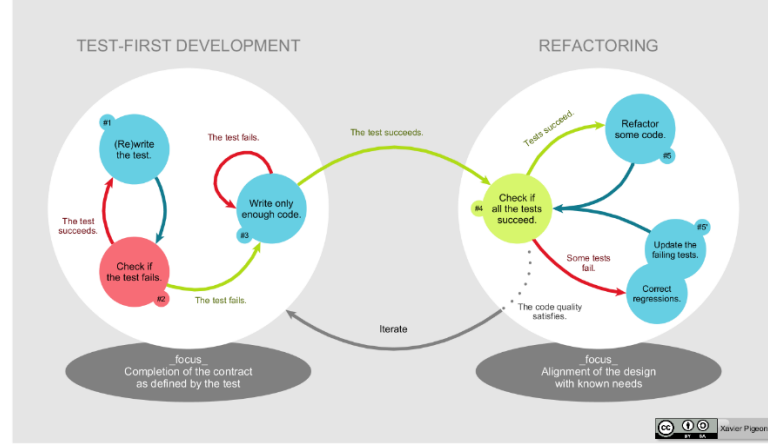
Eğer ilgili yazılımcı bu isterde TDD kullanmış olsaydı “top to bottom” yaklaşımını kullanmış olacaktı. Yani bakış açısını 180 derece değiştirmek zorunda kalacak.

1. Sunum katmanında bir test yazılır. Elimizde sadece kredi kartı ile ödeme olduğundan sadece kredi kartı ile ödeme şartını sağlayan bir test olmalıdır. Bu test sadece sunum katmanını ilgilendirmelidir. Aşağıdaki katmanlardaki kodlar henüz bulunmamaktadır.
2. Bir alt katmanda İş kurallarını test eden testler yazılır ve burası da sadece bu katmandaki kodları test eder. Her hangi bir şekilde veritabanı ile bir bilgi yoktur.
3. En altta Repository katmanında artık mevcut tasarımımıza ait veritabanı oluşturulur.

Eğer yeni uygulamada bir bug çıkarsa bug düzeltilir ve yazılmış testler tekrardan çalıştırılır. Artık otomatize testlerimiz olduğu için geliştirme döngümüz hızlamaya başladı. Bu noktada test yazmanın bir maliyeti olacağı düşünülebilir ama 2 yada 3 defa manuel test yaparken harcanan süre test yazmak ile hemen hemen aynıdır. Yazılımın yaşayan bir organizmaya benzetirsek uzun süre boyunca bu testler çok sayıda çalıştırılacaktır.

Örnekte eğer yeni bir ödeme tipi geldiğinde bu ödeme tipi ile ilgili geliştirme döngüsü yukarıdaki gibi tekrar eder. Ve tasarım ilgili isteklere göre evrimleşir.

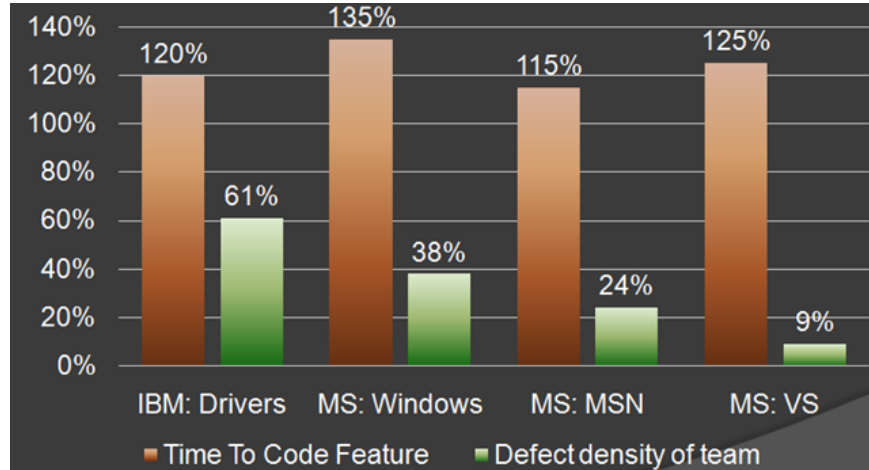
Testlerimizi başarılı ile sonuçlandırma safhasında önemli olan testlerin geçmesidir. Kodlarımızda if'ler else'ler mükerrer kodlar olabilir. Bunlar artık bizi endişelendirmemektedir. Çünkü elimizde artık otomatize testlerimiz var. Bu noktadan sonra TDD'nin bizim için sunduğu en önemli özellik olan REFACTORING kısmı devreye girmektedir. Bu istenildiği kadar yapılabilir.



Şekil 13. TDD iyileştirme yapılması

Yukarıdaki resimde da görüldüğü üzere iyileştirme yapıp testler çalıştırılır. Artık elimizde sürekli iyileştirmeye uygun bir kodlar bütünü oluşmaya başladı.

Her bir testi yazdıktan ve başarılı bir şekilde tamamladıktan sonra VCS'de kodları uygun mesaj ile commit'lemek faydalı olacaktır.



Şekil 14. TDD uygulandığında ve uygulanmadığındaki karşılaştırması

Yukarıdaki resim çeşitli kurumlardaki TDD uygulandığında ve uygulanmadığındaki karşılaştırmayı göstermektedir.

Sonuç olarak TDD'yi; basit kod yazmamızı, basit tasarım yapmamızı ve düzenli olarak refactoring yapmamızı sağlayan bir pratik olarak özetleyebiliriz.

4.7 Kristal (Crystal)

Farklı projelerin farklı yöntemlere ihtiyaç duyduğu düşüncesinden yola çıkılarak Kristal ailesi yöntemleri (Crystal Clear, Crystal Orange ve Crystal Orange/Web) ortaya atılmıştır. Bu yöntemler projelerdeki çeşitliliğe projedeki kişi sayısı ve hata sonuçları olmak üzere iki farklı açıdan yaklaşmakta ve her yöntem Konfor(Comfort), Stee Balı Kapital(Discretionary Money), Gerekli Kapital (Essential Money) ve Hayat Noktası (Life) olmak üzere 4 farklı seviye içermektedir. Crystal Clear 6 kişilik küçük gruplar için, Crystal Orange 10 ila 40 kişilik geliştirme grupları ile 1-2 sene sürecektir orta büyüklükteki projeler için kullanılır ve birden fazla grup içerir. Crystal Orange/Web ise web tabanlı projelerin geliştirilmesinde kullanılır. Kristal kendi kendine adapte olabilen, iletişim odaklı ve insan gücüne önem veren yöntemler topluluğudur. Bu yöntemler aşağıdaki varsayımları temel alır:

1. Her proje biraz farklı birtakım kural, uyum yada yöntemlere ihtiyaç duyar.
2. Proje çalışmaları insan ile ilgili konularda hassastır ve bu konuların gelişimine balı olarak daha da gelişir; insanlar daha iyi oldukça, grup içi çalışmalar daha da iyi olur.
3. Etkin iletişim ve sık yapılan teslimler orta seviye ürünlerine gereksinimi azaltır.

Kristal yöntemlerinin iki kuralı vardır: (1) 1 ila 3 ay arasında değişen en fazla 4 ay süren artımlarla geliştirim yapılır. (2) Her bir artım öncesi ve sonrası başarılar ve başarısızlıkların dile getirildiği fikir seminerleri düzenlenir. Kristal yöntemlerinden her biri yöntemin ağırlığına göre beyaz, sarı, turuncu, kırmızı renklerinden biri ile gösterilir. Renk ne kadar koyu ise yöntem o kadar ağırdır. Projenin büyüklüğü ve kritikliğine bakılarak uygun olan renkteki yöntem seçilir. Takım, artımda kodlanacak gereksinimleri ve i takvimini belirler. Her artım kodlamayı içeren yapılandırma, gösterim ve yeniden gözden geçirmeleri içerir. Süreç takımın teslim ettiği ürünlere ve zamana balı duruma göre izlenir ve takip edilir. Takım kendine ait bir süreç oluşturmak için kendi kendini kontrol eder. Aynı zamanda sürecin kendi kendine gelişimini sağlayacak yineleme sonu incelemelerine ağırlık verilir. Yazılım geliştirme süreci belirtilmediğinden XP ve Scrum ile entegre kullanılabilir. Çok kritik projelerde kullanıma çok uygun değildir, aynı ofiste çalışan geliştiricilerle iletişimin fazla olduğu küçük ve orta ölçekli projelerde kullanılabilir.

4.8 Açık Kaynak Geliştirimi (Open Source Development)

İnternet’le birlikte çok sayıda katılımcı ile açık kaynak kodlu yazılım geliştirme yaygınlaşmıştır. Yaygınlaşma süreci içinde InterNet News Server, Mozilla Web Server, ApacheWebServer, Linux iletim sistemi, Perl programlama dili gibi başarılı projeler ortaya çıkmış ve açık kaynak geliştirimine ilgi artmıştır. AKG ile şirketler projelerinin yada geliştirilen ürün sürümlerinin dünya çapında

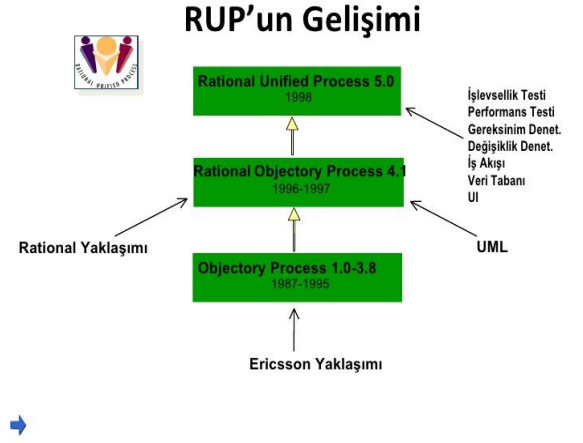
kullanılmasını sağlamaktadırlar. Hatta rafa kaldırılmış bazı projeler AKG ile geliştirilmeye devam edilebilir ve bu sayede ürünün fonksiyonallitesi, kalitesi ve kullanımı artırılabilir. Aynı zamanda birbirinden farklı donanımlarda ve durumlarda kullanıcı testinin yapılmasına olanak verdiğinden sorunların belirlenmesinde ve değişik çözüm önerilerinin geliştirilmesinde yarar sağlamaktadır. Kullanıcılar, dünyanın her yanından açık kaynak kodları görebilmekte, değiştirebilmekte ve lisans kapsamında projelerinde kullanıp geliştirdikleri kodu kapatabilmektedirler. Kullanıcı direk geliştiricilerle iletişim kurup kısa sürede teknik destek alabilmektedir. Bu hem maliyet hem i gücü tasarrufunu sağlamakta, yazılım geliştirme sürecini hızlandırmakta ve yazılım geliştiriminde tekrar kullanımını artırarak yazılım dünyasında genel gelişimi ve yetkinleşmeyi sağlamaktadır. Bu yöntem problem tanımı, değişikliği kodlayacak gönüllülerin bulunması, çözümün tanımı, kod geliştirme ve test, değişikliğin gözden geçirilmesi, kod teslimi ve dokümantasyon ile sürüm yönetimi safhalarını içerir. Açık kaynak kodlu geliştirim toplulukları proje lideri, gönüllü-kıdemli geliştiriciler, kod değişikliğine onay veren ikinci derece geliştiriciler, bakımdan sorumlu kişiler ve testleri yapıp, hataları bildiren veya kodlama yapmayıp sadece haber grubunu takip eden kullanıcıları içerir

4.9 Rational Unified Process

Rational Unified Process, 30 yıllık çalışmalar sonucunda ortaya çıkmıştır. RUP mimariye dayalı ve vaka tabanlıdır, bir başka deyişle “Tekrara dayanan” ve “Artırımsal” bir modeldir. Bu özellikle büyük ölçüde “Objectory” yaklaşımından alınmıştır. Rational Software firması, RUP’un eksik olan yanlarının tamamlanması amacıyla, bu eksik yanlar konusunda deneyimli olan başka şirketleri ya satın almış ya da şirketlere partnerlik anlaşmaları yapmıştır. Buna göre, Requisite Inc., Gereksinim Yönetim konusunda, Pure-Atria Konfigürasyonu Yönetimi konusunda, Başarım ve Yük testi konularında katkıda bulunmuşlardır. Bunun yanında geliştirilecek yazılım sisteminin iş süreçlerinin modellenmesi, vaka tabanlı kullanıcı ara yüzü geliştirme alanlarında da yenilikler geliştirilmiştir. Agile Unified Process ise RUP prensiplerine sadık kalarak, basitlik, çeviklik, geliştirme araçları bağımsız, yüksek değerleri faaliyetlere odaklanma konuları üzerinde yoğunlaşmıştır.

- 2003 yılından beri IBM’in bir bölümü tarafından oluşturulan bir iteratif yazılım geliştirme süreci çerçevesidir. Başarısız bir yazılımdaki sorunların aşılıp başarılı yazılım oluşturmak için gerekli adımları saptayarak oluşturulmuş bir süreçtir.
- Başarısız bir yazılımdaki özelliklerini yazımızın devamında okuyabilirsiniz. RUP şirketlere yazılım geliştirme aşamasında bir yön sağlayar. RUP use-case ve nesne teknolojileri tabanlı; tekrarlanan (iterative) yazılım geliştirme ve iş modelleme yöntemidir. RUP’un verebileceği özellikler şunlardır;

1. Müşteriyi ve yazılımcıyı organize edebilmesi,
2. Standart tanımlı adımları olması,
3. Oluşacak yazılımdaki sık değişiklikleri öngörebilmesi,
4. Basit olması,
5. Proje yönetim aktivitelerinin çok fazla olmaması.



Şekil 15. RUP'un Gelişimi

4.10 Microsoft Solution Framework for Agile

Microsoft tarafından 1993 yılında MSF 'in ilk versiyonu yayınlanmıştır. Sonraki yıllarda; 1997'de versiyon 2.0, 1999'da versiyon 2.5, 2002'de versiyon 3.0 ve en son 2005 yılında ise versiyon 4.0 yayınlanması izlemiştir. (Keeton, vd., 2006).

MSF başarılı bir bilgi teknolojileri süreci için yazılım geliştirme ekibinin nasıl organize edileceği, projelerin nasıl planlaması gerektiği, süreç yapısının nasıl oluşturulacağı, risklerin değerlendirip kurulumlarının nasıl yapılacağı ile ilgili bilgiler veren bir süreç yaklaşımıdır. MSF, süreç ve takım olmak üzere iki farklı modelden oluşmaktadır (Turner M., 2006).

Aşağıdaki prensipleri benimsemektedir:

- Müşteri ile birlikte geliştirme
- Açık iletişim kurmak
- Ortak bir vizyonda hareket etmek
- Kalitenin herkese ait ortak bir değer olduğuna inanmak
- Çevik olmak
- Günlük kurulum alışkanlığı edinmek
- Sürekli gelişim sağlamak
- Risk Yönetimi

4.11 Kanban

Kanban, üretim operasyonlarından türetilmiş, adaptasyonu yüksek, görsel ve maliyet odaklı bir tekniktir. 1950’lerde Toyota tarafından kullanılmasına rağmen, yazılım geliştirme sürecine uygulanması ilk kez, 2004 yılında David J. Anderson tarafından gerçekleştirilmiştir. David J. Anderson, Microsoft firmasında bir takımla birlikte çalışırken problemlerin giderilmesi amacıyla bu yöntemi ortaya çıkarmıştır (Ahmad vd., 2013)

Kanban arkasındaki temel fikir yalın düşünceden gelmektedir. Kanban yönteminin ana felsefeleri üç bölümde değerlendirilebilir; iş akışını görselleştirmek, üzerinde çalışılan işleri sınırlamak (Limit Work In Progress [WIP]) ve döngü sürelerini ayarlamak. Kanban yönteminde yapılan ve olası yapılacak işler “Kanban Board” denilen bir tahtada görselleştirilir. Bu sayede tüm takım ve müşteri; mevcut işlerin son durumunu, önceliklerini, olası darboğazları görerek aksiyon alır. Bu tahtada soldan sağa; yapılacak işler (ToDo), yapılmakta olan işler (In-Progress) ve testi geçen ve biten işler (Done) not 12 kağıtlarına yazılmış şekilde bulunmaktadır. Süreç boyunca bu not kağıtları soldan sağa doğru hareket etmektedir. Amaç başlangıçta yapılacak işler listesindeki işi, biten işler bölümüne kadar hareket ettirmektir. Bununla birlikte önemli unsurlarda biri de WIP’i mümkün olduğunda küçük tutmaktır. Bu sayede yazılım geliştiren personel verilen bir zaman aralığında sadece bu işe odaklanır ve müşteri de o zaman aralığı sonunda ne elde edeceğini bilir. Zaman baskısı yönünden incelendiğinde Scrum’dan farklı olduğunu söylenebilir, nitekim kısa geri besleme döngüleri ile hızlı adapte olmayı hedefler. Kanban kullanmanın anahtar güdüsü zorlayıcı yinelemeler değil, akışa odaklanmaktır (Ahmad vd., 2013).

Kanban metodunda uygulama üzerinde ki kusurlar bir atık olarak görüldüğünden dolayı, bu unsurların yok edilmesi, yok edilemiyorsa şayet minimum hale getirilmesi hedeflenmektedir. Uygulama kusurundan kaynaklanan atığın miktarı, kusurun ürüne etkisi ve fark edilmeme süresi olarak görülür. Örneğin, üç dakika içerisinde fark edilmiş bir kritik kusur büyük bir atık kaynağı olarak sayılmaz. Haftalarca fark edilememiş küçük bir kusur ise daha büyük bir atık olarak sayılır. Kusurların etkilerini azaltmanın yolu, kusurun oluşur oluşmaz onları tespit etmektir. Böylece harcanan zamanı azaltmak için kod yazarken anında test, sık sık entegrasyon ve mümkün olabildiğince çabuk sürüm yayınlanması yapılır (Poppendieck M. ve Poppendieck T., 2013)

Ayrıca, bu bilgilere ek olarak son zamanlarda Scrum ve Kanban bazı özellikleri bir araya getirilerek “Scrumban” isimli bir hibrit çevik yöntem çalışmalarına rastlanmaktadır. Bu yöntemde Kanban yönteminde olduğu gibi işler yapılacaklar tahtasına alınır ama farklı olarak bazı işler için zaman sınırı olduğunu gösterir farklı renkte kâğıtlar kullanılır (Reddy, 2014).

4.12 Yalın Yazılım Geliştirme

Yalın Yazılım Geliştirme olgusu ilk kez Avrupa Birliği'nin ESPRIT girişimi tarafından Almanya'nın Stuttgart şehrinde Ekim 1992'de organize edilen konferansın başlığı olarak kullanılmıştır. Bağımsız olarak, 1993 yılında Robert "Bob" Charette yazılım projelerinde riskleri daha iyi risk yönetimini araştıran çalışmasının bir parçası olarak "Lean Yazılım Geliştirme" kavramını önermiştir. "Yalın" terimi James Womack, Daniel Jones ve Daniel Roos tarafından 1991 yılında "The Machine That Changed The World: The Story of Lean Production" (Dünyayı Değiştiren Makine: Yalın Üretim Hikayesi, 2017) kitabında Toyota'da kullanılan yönetim yaklaşımını ifade etmek üzere kullanılan İngilizce terim olarak önerilmiştir. Yalın düşüncesinin yazılım geliştirmede uygulanabilir olacağı fikri çok önceleri, bu terim üretim süreçleri ve endüstriyel mühendislik alanındaki trendlerle ilişkili olarak ilk kez kullanıldıktan 1 ve 2 yıl sonra yerleşmiştir.

Womack ve Jones'un 1995'de yayınladıkları 2. Kitaplarında (Womack ve Jones, 2003) Yalın düşünmenin beş ana ögesi tanımlanmıştır. Bunlar:

- I. Değer
- II. Değer Akışı
- III. Akış
- IV. Çek
- V. Mükemmellik

Bu ilerleyen on yıldan daha fazla süre boyunca Yalın için varsayılan tanım haline geldi. Önerildiği üzere mükemmellik hedefine fazlalığı yok ederek ulaşılmıştır. Beş dayanak olmasına karşın, bu beşinci olanıydı, zararlı etkinliklerin ve bunların ortadan kaldırılmasının sistematik tanımlamasıyla mükemmellik arayışı geniş bir kitleye yankılanmıştır. Yalın, 1990'ların sonlarında ve 21. Yüzyılın başında özellikle önleme uygulamalarıyla ilişkilendirilmiştir.

Womack ve Jones'un Yalın tanımı evrensel olarak kabul edilemez. Toyota'daki yönetim prensipleri çok inceliklidir. Türkçe 'deki "Atık" kelimesi üç Japonca terimle daha zengin bir şekilde açıklanmaktadır:

- I. Muda – gerçek anlamda "atık" demektir, ancak katma değeri olmayan etkinliği belirtir
- II. Mura- "düz olmayan" anlamına gelmektedir ve akışta çeşitlilik olarak yorumlanır
- III. Muri- "fazla yüklenme" veya "mantıksızlık" anlamına gelmektedir.

Bu bilgiler ışığında, yalın yazılım geliştirmenin en öncelikle felsefesinin üretim sürecindeki atıkları atmak olduğu söylenebilir. Atık, kısmi yapılmış işler, ekstra yapılan işlemler, program hataları, ekstra

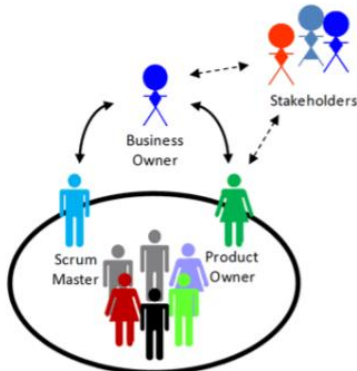
özellikler, görev değişiklikleri, beklemler, hareketlilik ve program kusurları gibi müşteri açısından ürünün değerine katkısı olmayan her şeydir. (Poppendieck ve Poppendieck 2013). Bu şekilde sadece müşterinin istediğini karşılayacak ürünü elde etme süreci yürütülmüş olur. Süreç boyunca tüm varsayımlar tekrar tekrar doğrulanır. Bir metrik veya pratik artık geçerli değilse çöpe atılır. Kısa yinelemeler sonunca geri beslemeler olarak ürün doğrulama (validation) gerçekleştirilir. Kararlar, mümkün olduğunca ertelenir ve alınması gereken son zamanda alınır. Bu şekilde kesinleşmemiş kararlara uygun atık işlemlerle gereksiz yere uğraşılmamış olunur.

4.13 Scrum



Şekil 16. Scrum Süreci

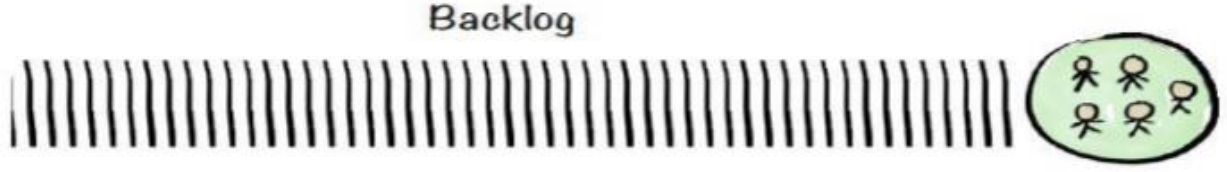
4.13.1 Scrum Takımı:



Şekil 17. Scrum Takım

Scrum Takımı: Ürün Sahibi, Geliştirme Ekibi ve Scrum Master'dan oluşur. Takım kendi kendini örgütler. Böylece kendi içerisinde uyum içinde olan takımlar daha başarılı sonuçlar alırlar. Scrum takım modeli esneklik, yaratıcılık ve verimliliği optimize etmek için tasarlanmıştır.

4.13.2 Backlog



Şekil 18. Backlog

Backlog :

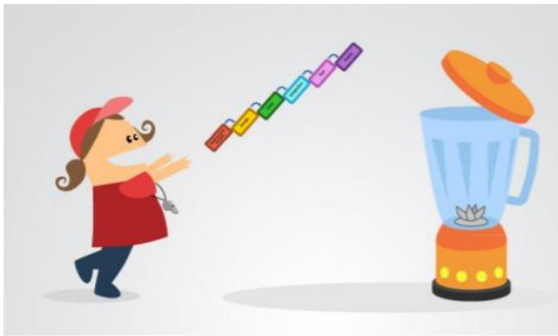
- Müşteriden ve son kullanıcıdan gelen gereksinimleri içerir.
- "Ne yapacağız" sorusunun yanıtını içerir.
- Herkese açık ve herkes tarafından müdahale edilebilir.
- Risk, iş değeri, zaman gibi kavramlara göre ürün sahibi tarafından sıralandırılır.
- User Story'lerden oluşur.

4.13.3 Sprint



Şekil 19. Sprint

- Belirli bir süreye sahiptir.
- Sonunda ortada değeri olan bir çıktı olmalıdır.
- Toplantılarla içerik belirlenir.
- Sprint süresi boyunca her gün toplantılar yapılır

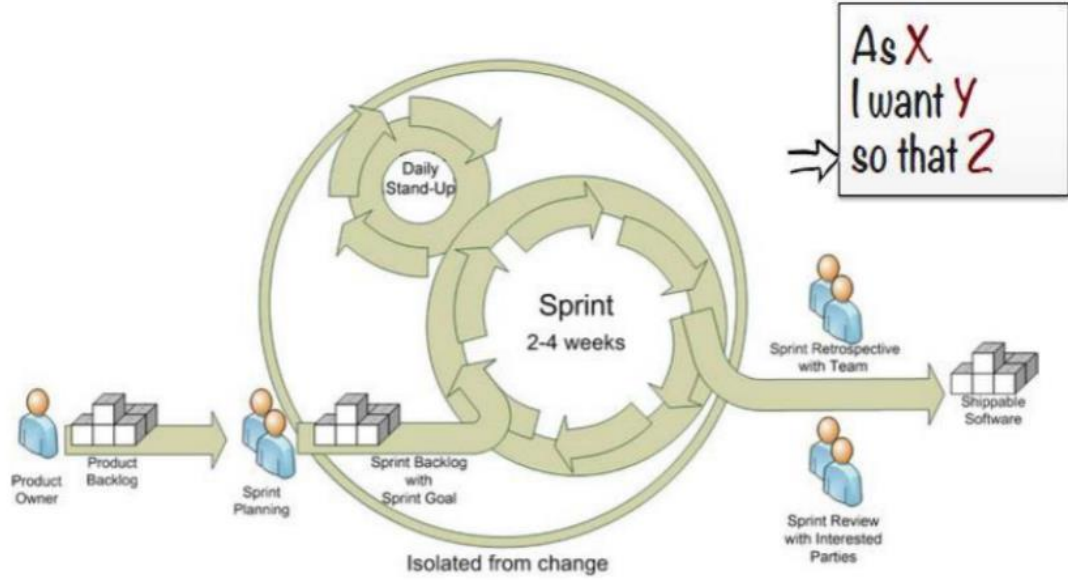


Sprint Gösterimi

Şekil 20. Sprint Gösterimi

4.13.4 User Story

Müşteri, son kullanıcı veya ürün sahibi için değerli olan ve anlam ifade eden genellikle fonksiyonel özelliklerin belirtildiği ifadelerdir.



Şekil 21. User Story

Örnek User Story: Online alışveriş yapan biri olarak, alışverişe daha sonra devam edebileyim diye, alışveriş kartımın kaydedilmesini istiyorum.

As online buyer
I want to save my shopping cart
so that I can continue shopping later

Fact: Features have different sizes

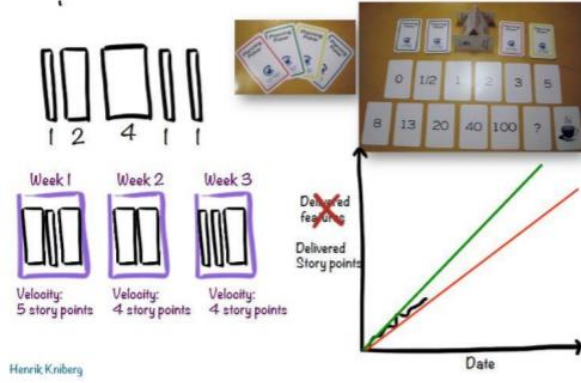


Şekil 22. Her bir User Story farklı bir boyutu

Her bir user story farklı bir boyuttadır. Somut olarak bakarsak, bir projedeki her bir gereksinim için gereken iş gücü ve zaman aynı değildir. Bu sebeple ürün backlogları sprintlere bölünürken, user storylerin boyut ve öncelikleri göz önünde bulundurulur. Örneğin bir sprint 3 user story içerirken diğeri daha küçük boyutlarda 5 user story içerebilir. Peki boyutları nasıl belirleyeceğiz?

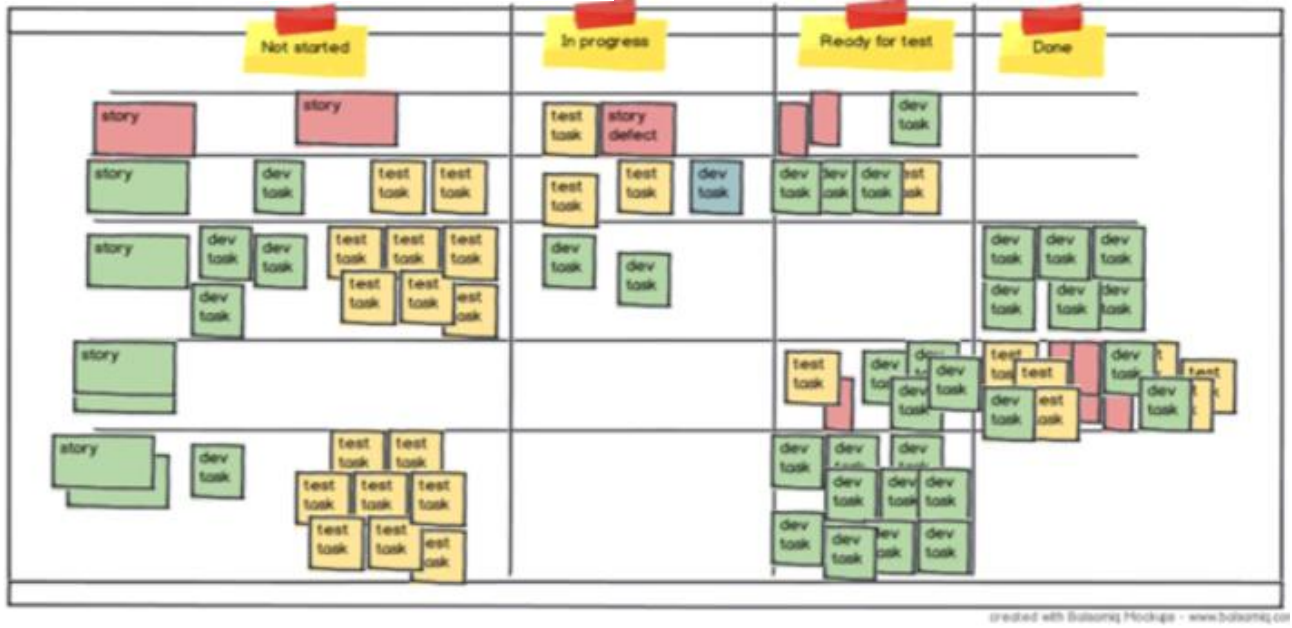
İş görüldüğü gibi değil!

4.13.5 Poker Kartları



Şekil 23. Poker Kartları

Scrum takım üyeleri bir araya gelir. Scrum master bir user story okur. Takımdaki her bir üye user story için uygun gördüğü poker kartlarından birini seçer. Herkes kartları seçtikten sonra tüm kartlar açılır ve değerlendirilir. Böylece herkesin ortak görüşü sonunda user story'lerin büyüklüğü belirlenir.



Şekil 24. SCRUM Board

4.13.6 Scrum XP karşılaştırılması

SCRUM	XP
<ul style="list-style-type: none"> ➤ Sprint (2 hafta-1 ay) ➤ Sprintler en son halini aldıktan, toplantı yapıldıktan sonra değişmez. ➤ Özellikler geliştiriciler tarafından derecelendirilir ➤ Herhangi bir mühendislik pratiği tanımlamaz 	<ul style="list-style-type: none"> ➤ Sprint (1 yada 2 hafta) ➤ Sprintler değişebilir. ➤ Özellikler ürün sahibi tarafından derecelendirilir. ➤ Mühendislik pratikleri tanımlar. Eşli programlama, otomatik test, basit dizayn vs.

KAYNAKLAR

- [1] Yazilimheryerde, “Agile (çevik) metodoloji”. Erişim 23 Nisan 20201. <http://www.yazilimheryerde.com>
- [2] Medium, “Agile nedir? Scrum nedir? Başarılı proje yönetimi yöntemleri nelerdir?”. Erişim 23 Nisan 20201. <https://medium.com>
- [3] Bukrek, “Artıları ve Eksileri ile Agile (Çevik) Yöntemler”. Erişim 23 Nisan 20201. <https://www.bukrek.com/>
- [4] Argenova, “Agile Proje Yönetimi Nedir?”. Erişim 23 Nisan 20201. <https://www.argenova.com.tr/>
- [5] Nttdata, “Agile: Çağımızın Proje Yönetimi Metodolojisi”. Erişim 23 Nisan 20201. <https://nttdata-solutions.com/>
- [6] Acmagile, “Agile nedir?”. Erişim 23 Nisan 20201. <https://www.acmagile.com/>
- [7] Pem360, “Agile Yöntemlerin Avantajları ve Dezavantajları?”. Erişim 24 Nisan 20201. <https://www.pem360.com/>
- [8] Toptalent, “Agile Nedir? Agile Metodu Nasıl Uygulanır?”. Erişim 24 Nisan 20201. <https://toptalent.co/>
- [9] İakademi, “YAZILIM GELİŞTİRME METODOLOJİLERİ NELERDİR?” Erişim 21 Nisan 20201. <https://www.iakademi.com/>
- [10] Fırat üni, “Çevik Yazılım Geliştirme Nedir?” Erişim 21 Nisan 20201. <http://web.firat.edu.tr/>
- [11] Selma SÜLOĞLU (2018) . “Yöntem Çevik Olunca” . Datasel Bilgi Sistemleri A.Ş., Ankara, Türkiye
- [12] Fikirjeneratuoru, “Yazılım Süreç Yönetim Modelleri ve Karşılaştırılması”. Erişim 26 Nisan 20201. <https://fikirjeneratuoru.com>
- [13] Slideplayer, “Birleşik Süreç ve Çevik (Agile) Yazılım Süreç Modelleri”. Erişim 26 Nisan 20201. <https://slideplayer.biz.tr>
- [14] AYKUT ŞAHİN (2018). “ÇEVİK YAZILIM GELİŞTİRME YAKLAŞIMI: PERAKENDE SEKTÖRÜNE UYARLAMA”. İSTANBUL
- [15] Docplayer, “Yazılım Tasarım ve Mimarisi. Birleşik Süreç ve Çevik (Agile) Yazılım Süreç Modelleri”. Erişim 26 Nisan 20201. <https://docplayer.biz.tr>
- [16] Bilgisayarmuhendisleri, “Extreme Programming (XP) Nedir”. Erişim 27 Nisan 20201. <https://www.bilgisayarmuhendisleri.com/>
- [17] Slideshare, “Extreme Programming (XP) Nedir”. Erişim 27 Nisan 20201. <https://www.slideshare.net/>

- [18] Yazilimprojesi, “Extreme Programming (XP) Nedir”. Erişim 27 Nisan 20201. <http://www.yazilimprojesi.com/>
- [19] Burakkutbay, “Extreme Programming (XP) Nedir”. Erişim 28 Nisan 20201. <https://blog.burakkutbay.com/>
- [20] Blogspot, “Extreme Programming (XP) Nedir”. Erişim 28 Nisan 20201. <http://bilgisayar-muhendisleri.blogspot.com>
- [21] Yazilimcildunyasi, “Extreme Programming (XP) Nedir”. Erişim 28 Nisan 20201. <http://www.yazilimcildunyasi.com>
- [22] Kurumsaljava, “Extreme Programming (XP) Nedir”. Erişim 28 Nisan 20201. <http://www.kurumsaljava.com>
- [23] Medium, “Extreme Programming (XP) Nedir”. Erişim 28 Nisan 20201. <https://medium.com/>
- [24] Ambysoft, “The Agile Unified Process (AUP) Nedir”. Erişim 28 Nisan 20201. <http://www.ambysoft.com/>
- [25] Bydrec, “Agile Software Development Methods: What is the Agile Unified Process?”. Erişim 29 Nisan 20201. <https://blog.bydrec.com>
- [26] Medium. “TDD nedir?”. Erişim 29 Nisan 20201. <https://medium.com/>
- [27] Odayibasi. “TDD nedir?”. Erişim 29 Nisan 20201. <https://odayibasi.medium.com/>
- [28] Devnot. “TDD nedir?”. Erişim 29 Nisan 20201. <https://devnot.com/>
- [29] Kodlayarakhayat. “TDD nedir?”. Erişim 29 Nisan 20201. <https://kodlayarakhayat.com/>
- [30] Yilmazcihan, “Feature Driven Development Nedir?”. Erişim 29 Nisan 20201. <http://www.yilmazcihan.com/>