Original Article

Kafkas Üniversitesi Fen Bilimleri Enstitüsü Dergisi / Institute of Natural and Applied Science Journal

Cilt 16, Sayı 2, 54-62, 2023 / Volume 16 Issue 2, 54-62, 2023

Kafkas Üniversitesi Fen Bilimleri Enstitüsü Dergisi
Institute of Natural and Applied Science Journal

Dergi ana sayfası/ Journal home page: https://dergipark.org.tr/tr/pub/kujs

E-ISSN: 2587-2389

Çevik ve Şelal Metodolojilerinin Karşılaştırılması ve Uygulama İlkeleri: Bir Modelleme Çalışması

Ezgi Pelin YILDIZ [1*], Deniz ŞENGÜL [2]

[1] Kafkas Üniversitesi Kazım Karabekir Teknik Bilimler Meslek Yüksekokulu, Bilgisayar Programcılığı Bölümü, Kars, Türkiye
[2] İzmir Kâtip Çelebi Üniversitesi / Fen Bilimleri Enstitüsü Yazılım Mühendisliği Anabilim Dalı, İzmir, Türkiye

(İlk Gönderim / Received: 20. 11. 2023, Kabul / Accepted: 20. 05. 2024, Online Yayın / Published Online: 05. 08. 2024)

Anahtar Kelimeler:
Yazılım/Sistem Geliştirme Modeli (YGM),
Waterfall Metodolojisi (Şelale Modeli),
Agile Metodolojiler (Çevik Metodolojiler),
Proje Özellikleri,
Modelleme Çalışması.

Özet: Yazılım geliştirme projeleri uzun bir süre boyunca plan odaklı süreçlerle yönetilmiştir; ancak Agile (Çevik) Metodolojilerin büyümesi, yazılım/sistem geliştirmeye daha adapte bir yaklaşım sunmuştur. Bu makalenin amacı, iki Yazılım Geliştirme Modelini kısaca tanıtmak; Waterfall Model (Şelale Modeli) ve Agile Metodolojiler (Çevik Metodolojiler) ve her iki model için karşılaşılabilecek sorunlar ile tipik proje özellikleri sunmaktır. Sonuç olarak bu çalışmada iki Yazılım Geliştirme Modeli (YGM) tanıtılmıştır. Plan odaklı Şelale Model (Waterfall Model) ve uyarlamalı Çevik Metodolojiler (Agile Methodologies) olmak üzere. Her iki modelin de kullanım alanları, avantajları ve dezavantajları bulunmaktadır. Küçük projelerin neredeyse her zaman bir Çevik Metodoloji yaklaşımı için uygun olduğu ve neredeyse asla Şelale Model bir yaklaşım için uygun olmadığı tespit edilmiştir. Hem Şelale Model hem de Çevik Metodolojiler, orta büyüklükteki projelerle uğraşırken zorluklar yaşandığı saptanmıştır. Zorlayıcı bir Şelale Model, nispeten basit bir projeye gereksiz karmaşıklık ekleyebilirken, aynı projeye esnek bir Çevik Metodoloji yaklaşımın daha uygun olduğu da diğer sonuçlar arasındadır.

Comparison of Agile and Waterfall Methodologies and Application Principles: A Modeling Study

Keywords:
Software/System Development Model (YGM),
Waterfall Methodology,
Agile Methodologies, Project Characteristics, Modeling Work.

Abstract: Software development projects have been managed with plan-driven processes for a long time, but the growth of Agile Methodologies has introduced a more adaptive approach to software/system development. The purpose of this paper is to provide a brief introduction to two Software Development Models (SDMs), the Waterfall Model, and Agile Methodologies, and to present issues and typical project characteristics for both SDMs. As a result, two Software Development Models (SDM) are introduced in this study. including the plan-oriented Waterfall Model and adaptive Agile Methodologies. Both models have areas of use, advantages and disadvantages. It has been found that small projects are almost always suitable for an Agile Methodology approach and almost never for a Waterfall Model approach. Both the Waterfall Model and Agile Methodologies have been found to have difficulties when dealing with medium-sized projects. While a challenging Waterfall Model can add unnecessary complexity to a relatively simple project, a flexible Agile Methodology approach is more appropriate to the same project.

*İlgiliyazar: yildizezgipelin@kafkas.edu.tr
DOI: 10.58688/kujs.1393276

1. GİRİŞ

Karmaşık yazılım sistemlerinin geliştirilmesi tarihsel olarak birçok kişi ve çalışma ile şekillenmiştir. Ancak, yazılım mühendisliği disiplininin temelleri ve yazılım geliştirme yöntemlerinin ilkeleri konusunda önemli bir katkı, Dr. Winston W. Royce tarafından yapılmıştır. Royce, yazılım geliştirmenin planlı bir süreçle yönetilmesi gerekliliğini 1970 yılında yayımladığı bir makalede ayrıntılı olarak belirtmiştir.

Royce'un 1970 tarihli bu makalesi, "Managing the Development of Large Software Systems" başlığını taşır ve yazılım mühendisliği tarihinde önemli bir kilometre taşıdır. Bu makale, Waterfall Model olarak bilinen yazılım geliştirme metodolojisinin temelini atmıştır. Waterfall Model, yazılım geliştirme sürecini aşağıdaki sıralamada ele alır: gereksinimler analizi, tasarım, uygulama, test, sürüm ve bakım. Her aşama bir öncekini takip eder ve geri dönüşe izin vermez.

Royce'un makalesi, yazılım geliştirme süreçlerini daha önce belgelenmiş ve açıklanmış bir şekilde sistematik bir şekilde tanımlamıştır. Özellikle, bu makalede yazılım projelerinin gereksinimlerin daha iyi anlaşılması ve yönetilmesi gerektiği, tasarımın önemine vurgu yapıldı ve yazılım testlerinin ayrıntılı bir şekilde ele alınması gerektiği belirtilmiştir.

Royce'un bu makalesi, yazılım mühendisliği alanında birçok geliştirme metodolojisinin temelini atmıştır. Ancak, zaman içinde yazılım geliştirme yaklaşımları büyük ölçüde değişmiş ve dönüşmüştür. Özellikle 2000'lerin başından itibaren "Çevik Yazılım Geliştirme" yaklaşımı, daha esnek ve işbirlikçi bir yazılım geliştirme yaklaşımı olarak popülerlik kazanmıştır. Çevik yaklaşım, Waterfall Model'in daha katı ve lineer yapısına bir alternatif sunmuştur (Royce, 1970). Bu yazıda yazılım geliştirme konusunda genel bir fikir verilmesinin yanı sıra, günümüzde hala uygulamada olan birçok yazılım geliştirme kavramı tanıtılmıştır. Bu makale, yazılım geliştirme hakkındaki bu genel fikri Waterfall Modeli'nin bir şekli olarak ve bu modelin sorunlarıyla açıklayacaktır. Aynı zamanda, Waterfall Modelinin doğal bir karşıtı olarak Agile Metodolojileri tanıtacak ve bu SDM ailesiyle ilgili sorunları ele alacaktır. Genel olarak, Waterfall Modeli bir plan odaklı yaklaşımı benimserken, Çeviklik (Agility) bir adapte yaklaşımı takip eder. Bu makale, bir projenin Agile veya Waterfall yaklaşımına uygunluğunu belirleyen etkenlere sınırlı bir bakış sunacaktır.

Bölüm 1.1'de, bir SDM olarak Waterfall Modeli hakkında kısa bir açıklama, başlangıcının tarihi ve ilgili sorunlar verilecektir. Bölüm 2'de Agile Metodolojileri, Agile özellikleri, uygulamaları ve sorunları hakkında bir genel bakış sunacaktır. Bölüm 3, belirli bir proje için hangi geliştirme metodolojisinin en iyi kullanılacağına karar vermek için bir başlangıç noktası olacaktır. Bu bölüm, tipik Agile projelerini ve tipik Waterfall projelerini içeren ve plan odaklı ve Agile projelerin tipik özelliklerine sınırlı bir bakış sunmaktadır. Ayrıca, değişen Agile ve Waterfall metodolojileri ile ilgili makalelere, SDM'lerin popülerliğini ölçen bir makaleye, hangi yazılım metodolojisinin kullanılacağına karar verme konusunda bir makaleye ve dağıtık bir ortamda Çevik süreçlere dair bir makaleye referanslar ekler. Bölüm 4, bu makalenin avantajlarını, Agile veya Waterfall yaklaşımı için uygun projeleri ve her iki yaklaşım için öngörülmesi gereken sorunları özetleyerek belirtir.

1.1. Şelale Model (Waterfall Model): Yazılım Mühendisliği Tarihine Kısa Bir Giriş

1.1.1. Yazılım Mühendisliği Tarihine Kısa Bir Giriş:

Yazılım mühendisliği, modern bilgi teknolojilerinin ve dijital dönüşümün temelini atmış ve günümüzün karmaşık bilgisayar sistemlerinin geliştirilmesine yönelik sistematik bir yaklaşımın doğmasına katkı sağlamış önemli bir alanı ifade eder. Bu yazı, yazılım mühendisliği teriminin ortaya çıkışını ve gelişimini inceleyerek, bu alandaki temel tarihsel gelişmeleri aydınlatmayı amaçlamaktadır.

Yazılım mühendisliği terimi, 1960'ların sonlarında ve 1970'lerin başlarında özellikle NATO (Kuzey Atlantik Antlaşması Örgütü) tarafından düzenlenen konferanslarla ilişkilendirilir (Ganis, 2010). Bu dönemde, yazılımın giderek artan karmaşıklığı ve büyüklüğü, yazılım projelerinin yönetimi açısından ciddi zorluklar ortaya çıkarmıştır. Bu bağlamda, yazılım geliştirmenin mühendislik disiplini içinde ele alınması gerektiği fikri ortaya çıkmıştır. Özellikle 1968'de Almanya'nın Garmisch kentinde düzenlenen bir konferans, yazılım mühendisliği teriminin önemli bir şekilde öne çıkmasına yol açmıştır. Yazılım mühendisliği terimi, yazılım geliştirme süreçlerinin daha önceki rastgele ve düzensiz yöntemlerden çıkarak daha sistematik ve mühendislik temelli bir yaklaşım gerektirdiğini yansıtır. Yazılımın geliştirilmesi, planlanması, tasarlanması ve sürdürülmesi gibi süreçler, mühendislik ilke ve yöntemleri çerçevesinde ele alınmaya başlanmıştır. Bu dönüşümün temel nedenlerinden biri, yazılım projelerinin sıklıkla maliyet ve zaman aşımına uğraması, eksik belgelenmesi ve düzensizliğiydi.

Bu dönemdeki yazılım geliştiricileri, mühendislik disiplinlerinden gelen birikime dayalı olarak yazılımın donanım mühendisliği gibi ele alınması gerektiğini savunmuşlardır. Yazılım projelerinin daha sistematik ve planlı bir şekilde yönetilmesi gerektiği düşünülmüş, "bir kere ölç, bir kere kes" prensibi benimsenmiştir. Bu, yazılım geliştirme süreçlerinin daha disiplinli ve mühendislik odaklı hale gelmesinin ilk adımlarından biridir.

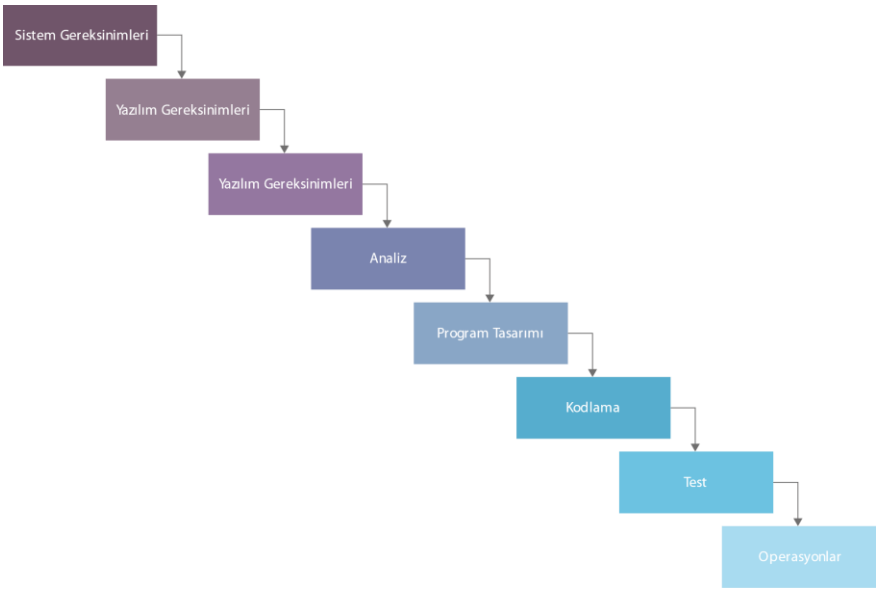
Yazılım mühendisliği terimi, yazılım geliştirme süreçlerinin daha mühendislik temelli ve sistemli bir yaklaşımla ele alınmasının gerekliliğini yansıtan önemli bir tarihsel gelişmenin ürünüdür. NATO konferansları, bu yaklaşımın kabul edilmesine büyük katkı sağlamış ve yazılım mühendisliği disiplininin temellerini atmıştır. Bugün, yazılım mühendisliği terimi, yazılım projelerinin planlama, tasarım, geliştirme ve sürdürülme aşamalarında mühendislik ilke ve yöntemlerinin uygulanmasını ifade eden temel bir kavramdır (Ganis, 2010).

1.1.2. Şelale Modeli (Waterfall Model):

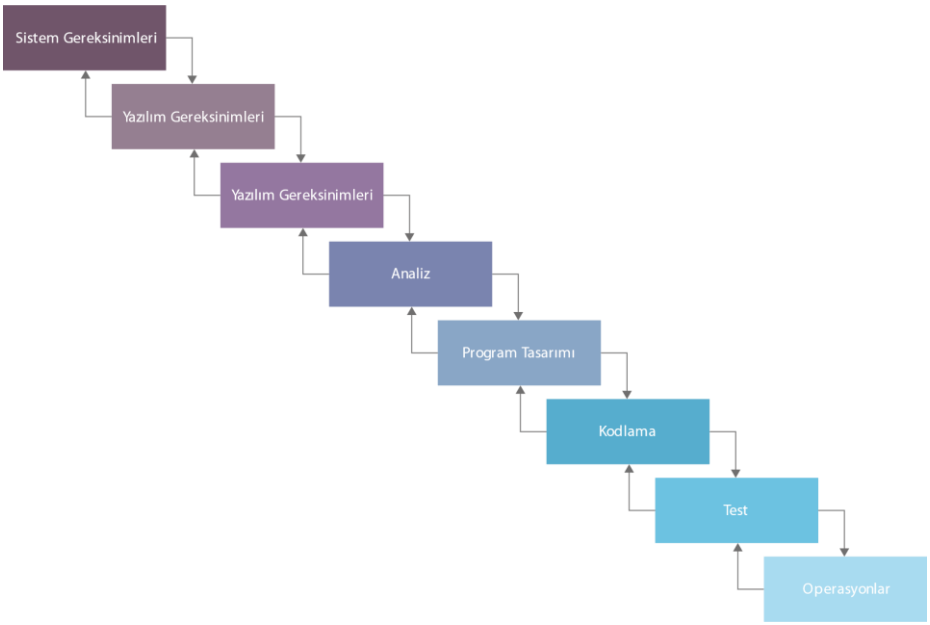
Dr. Winston W. Royce'un "Managing the Development of Large Software Systems" adlı makalesi, yazılım geliştirme

süreçlerinin temelini atmış ve yazılım mühendisliği tarihinde önemli bir kilometre taşıdır. Bu makale, 1970 yılında IEEE WESCON konferansının bir parçası olarak sunulmuştur. Makalede, Royce, büyük yazılım sistemlerinin geliştirilmesi konusunda önemli konuları ele almaktadır (Ganis, 2010). Çalışmada Royce, yazılım geliştirme süreçlerini daha sistematik ve mühendislik odaklı bir şekilde ele almanın gerekliliğini vurgular.

Royce, yazılım projelerinin başlangıcından sonuna kadar geçen sürecin planlanması gerektiğini vurgular. Planlamada, gereksinimler analizi, tasarım, kodlama, test ve sürdürme gibi aşamaların belirlenmesi ve sıralanması önemlidir. Çalışmada belirtilen ve daha sonra "Şelale Modeli" olarak adlandırılacak olan kavram, yazılım geliştirme sürecini aşamalara ayırma fikrini tanıtır. Royce, bu aşamaların sırasıyla izlenmesi gerektiğini savunur: gereksinimler analizi, tasarım, kodlama, test ve sürdürme. Bu, yazılım projelerinin daha önceki adımın tamamlanmasının ardından devam etmesi gereken bir lineer süreç olarak kabul edilir. Royce, aşamalar arasında bir geri dönüşün olmayacağını, her aşamanın bir sonrakine bağımlı olduğunu belirtir. Bu, önceki aşamanın tamamlanmadan sonraki aşamaya geçilemeyeceği anlamına gelir.



Şekil 1. Şelale Model.



Şekil 2. Şelale Modeli- Ardışık aşamalar arasındaki tekrar eden ilişki.

Şekil 1'de analiz aşaması ve kodlama aşaması, son ürüne en doğrudan değeri sağlar; diğer aşamalar, program kaynaklarının en iyi şekilde kullanılması için farklı şekillerde ele alınmalı ve planlanmalıdır (Ganis, 2010).

Bu Şelale Modeli ile ilgili olarak (Unhelkar, 2016) önceki teslim edilenlere bağımlılığı vurgular. Örneğin, analiz modeli henüz onaylanmadıysa sistem tasarımını geciktirir ve tasarım henüz onaylanmadıysa kodlamayı geciktirir.

Royce'un incelenen bir sonraki adımı, ardışık geliştirme aşamaları arasındaki iteratif ilişkiyi kapsar. Şekil 2, bu adımı Royce'un detaylarını göstermektedir. Royce (1970), her geliştirme adımının ilerledikçe ve tasarım daha detaylı hale geldikçe, önceki ve sonraki adımlarla bir iterasyon olacağına inanır, ancak daha uzak adımlarla nadiren bir iterasyon olur. Royce (1970), böyle dar kapsamlı bir değişim sürecinde bir değer görmektedir. Tasarım sürecinin herhangi bir anında, gereksinim analizi tamamlandıktan sonra beklenmedik tasarım zorlukları durumunda geri dönülebilecek sıkı ve yakın bir referans noktası bulunur.

Bir sonraki adımda Royce (1970), yazılım gereksinimleri aşaması ile analiz aşaması arasına ön tasarım aşamasını ekler. Bu aşama beş bileşene ayrılır. Bu teknikle, program tasarımcısı yazılımın depolama, zamanlama ve veri akışı nedeniyle başarısız olmayacağını garanti eder. Bu aşama, bu yazılım geliştirme sürecinin oluşturulduğu dönemi gösterir ve mühendislerin karşılaştığı sorunların karmaşıklığına dair bir içgörü sunar.

Sonuç olarak, Royce'un açıklamalarının sonunda, Royce'un vizyonu her geliştirme aşaması için belgelendirme, tüm aşamaları içeren daha kısa bir pilot aşama, müşteri katılımını içerir ve program tasarım aşamasından sonra ve test aşamasından sonra hem müşteri katılımını içerir.

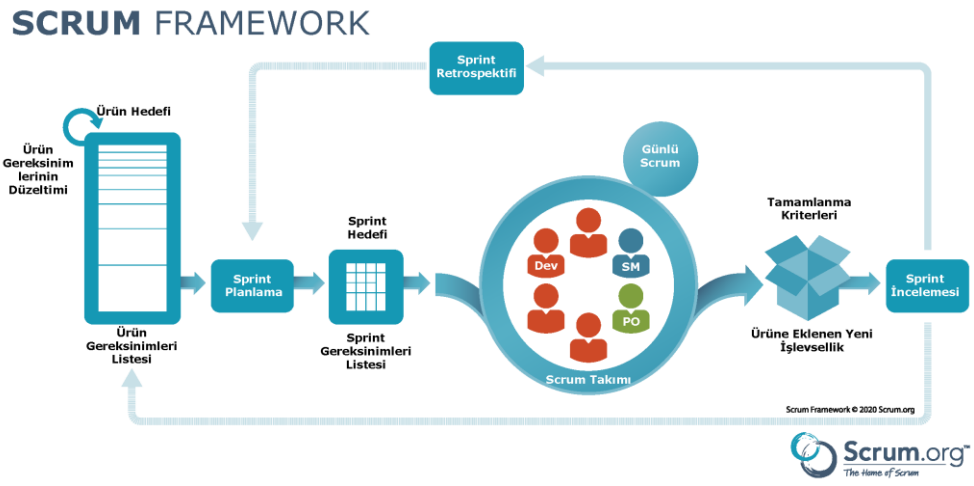
1.1.3. Şelale Model Sorunları:

Ganis (2010), Şelale Model'in geliştirme yaşam döngüsünün çok erken aşamalarında gereksinim spesifikasyonlarını veya yüksek düzeyde tasarımı dondurma üzerine odaklandığını düşünmektedir, daha kapsamlı tasarım ve uygulama çalışmalarına girişilmeden önce. Bu nedenle, gereksinimlerin iyi anlaşılmadığı, tanımlanmadığı veya projenin seyri içinde değişebileceği durumlarda Şelale modelinin uygun olmadığını düşünmektedir. Petersen ve ark. (2009), Şelale Model'i yüksek maliyetler ve çabalarla ilişkilendirir. Her aşamada onaylanması gereken belgelerin sayısı, değişiklik yapmanın zorluğu, iterasyonların başlatılması ve başarılması için gereken zamanın zorluğu ve sadece daha sonraki aşamalarda ortaya çıkan sorunlar, onların inancını doğrulamaktadır. Bu sorunların sonucunda, müşterilerin mevcut ihtiyaçları ele alınmaz ve uygulanmış ancak kullanılmayan özellikler ortaya çıkar. Bu bölümü sonlandırmak için, Munassar ve Govardhan (2010), Şelale Model'i, birçok - daha çağdaş - Yazılım Geliştirme Yaşam Döngüsü Modeli için bir temel olarak kabul ederlerken, Dr. Winston Royce'u plan-odaklı yazılım geliştirme metodolojilerinin kurucu babası olarak kabul edebiliriz.

2. MATERYAL VE METOT

2.1. Çevik Metodolojiler (Agile Methodologies): Motivasyon, İlkeler ve Uygulama

Fowler ve Highsmith tarafından yayımlanan Çevik Bildiri, modern Çevikliğin temelini resmi olarak duyurdu (Fowler ve Highsmith, 2001). Çevikliğin kurucuları ve destekçileri, yazılım geliştirme süreçlerini daha iyi hale getirmenin yollarını bulma konusundaki motivasyonlarını, bu işi yaparak ve diğerlerinin de yapmasına yardımcı olarak bulurlar. Çevik Bildiri'nin temel ilgisi değerli yazılım geliştirmek olsa da, Unhelkar (2016), Çevikliğin işin tüm boyutlarında stratejik bir rol oynadığını vurgular. İşlevsel yazılım sistemlerinin hemen hemen tüm iş fonksiyonlarıyla yakın bir şekilde iç içe geçmesi, Çevik işletmelerin karşılık gelen Çevik yazılım sistemlerine ihtiyaç duymasına neden olur Unhelkar (2016).



Şekil 3. Scrum - Şematik Bir Bakış.

2.1.1. Çevik İlkeler ve Uygulamalar:

Cohen ve ark. (2003), tarafından belirtildiğine göre, Çevik teknikler uygulamalar ve vurgulamalar açısından farklılık gösterse de, tekrarlayan geliştirme ve etkileşime, iletişime ve kaynak yoğun ara ürünlerin azaltılmasına odaklanma gibi ortak özellikleri paylaşırlar. Ayrıca, Çevik yaklaşımlar kısa tekrarlayan döngüleri, özellik planlaması ve dinamik önceliklendirmeyi bir araya getirirler, diye belirtir Highsmith ve Cockburn (2001). Çeviklik yüz yüze iletişimi gerektirir, bu da ekip üyelerinin kararlar almasını ve onları beklemek yerine hemen uygulamalarını sağlayan yakın bir konumda çalışmayı gerektirir.

Çevik geliştirme yöntemleri, yakın müşteri işbirliği gerektirir, diye belirtir Highsmith ve Cockburn (2001). Bir müşterinin artanı değerlendirmesi ve geri bildirimde bulunabilmesi için bir kullanıcı arayüzü gibi görünür bir sonuca ihtiyacı vardır veya gömülü sistemler durumunda bir yazılım simülasyonu gerekir

2.1.2. Çevik İterasyonlar

Williams (2007), her bir iterasyonun, bir ürün içinde, gereksinim analizini, tasarımı, uygulamayı ve testi içeren, kendi kendine yeterli küçük bir projenin aktivitelerini temsil ettiğini açıklar. Williams, her iterasyonun bir iterasyon sürümüne yol açtığını belirtir; bu sadece içsel bir sürüm olabilir ve tüm yazılımı ekibin üzerinde birleştirir ve son sistem için büyüyen ve evrilen bir alt küme olur. Müşteri, gereksinimlerini projenin başlangıcında belirlemek yerine, mevcut sürümün gözlemlemesine dayanarak bir sonraki sürüm için gereksinimlerini uyarlar. Sık sık belirli tarihlerin yazılım sürecinin varyansını azalttığı ve bu nedenle tahmin edilebilirliği ve verimliliği artırabileceğine dair kanıtlar bulunmaktadır. Çevik geliştirme sırasında kullanılan önceden belirlenmiş iterasyon süresi, ekip için bir zaman kutusu olarak hizmet eder. Dolayısıyla, seçilen kapsamı uygun hale getirmek için iterasyon süresini artırmak yerine, kapsamı iterasyon süresine uygun hale getirir. Çevik yöntemlerle geçmişteki iterasyon yöntemleri arasındaki temel fark, her iterasyonun süresidir. Geçmişte, iterasyonlar üç ila altı ay sürebilirdi. Çeviklikle birlikte iterasyon süreleri bir ila dört hafta arasında değişir ve bilinçli bir şekilde 30 günü aşmaz. Ayrıca, kısa iterasyonların, daha düşük karmaşıklık ve risk, daha iyi geri bildirim ve daha yüksek üretkenlik ve başarı oranları anlamına geldiğini gösteren araştırmalar bulunmaktadır Williams (2007).

Çevik: Destekleyici Araçlar ve Teknikler:

Çevikliğin kod merkezli odaklanmasını göz önüne alarak, çeşitli kodlama araçlarının kullanılması şaşırtıcı olmamalıdır. Model Driven Architecture, modellerin merkezi ürünler olduğu, uygulama ve orta yazılım kodunun önemli bir kısmının otomatik olarak oluşturulması ve geliştiricilerin çalıştığı seviyede soyutlama seviyesini artırarak gelişmeyi hızlandırır, Turk ve ark (2014)'e göre böyle bir araçtır. Ayrıca, entegre geliştirme ortamları, kodun sistemli olarak iyileştirilmesi ve otomatik test takımları, hataların tespit edilmesi ve giderilmesinin maliyetini yönetmede yardımcı olabilir, hatta Çevik bir sürecin sonlarında bile. Sürüm kontrolü ve değişiklik izleme, iteratif bir ortamda çok önemli olduğundan, her aktif programcının araç kutusunda bulunabilir. Çevikliğe uygun diğer teknikler, hata enjeksiyonu ve otomatik test oluşturma teknikleri gibi dinamik güvenlik test araçlarına entegre edilebilecek tekniklerdir.

2.1.3. Bir Çevik Metodoloji Örneği: Scrum

1995 yılında Schwaber (1995), Scrum'u tanıttı. Günümüzde geniş bir şekilde kabul gören bir Çevik Metodoloji'dir. Bu nedenle detaylı bir şekilde açıklanması için iyi bir adaydır. Web sitelerinde Rawsthorne ve Shimp (2009), Scrum ekibinin (genellikle 10 kişiden az) amacının, paydaşlarla çalışmanın bir sonucu olarak ürün sahibi tarafından belirlenen hedefleri ve öncelikleri karşılayan sonuçlar üretmek olduğunu belirtirler. Normalde, eklerler, Scrum ekibi sonuç üretmeye başlamadan önce bir vizyon aşaması bulunur (bu aynı zamanda bir projenin ilk aşaması da olabilir); bu aşamada iş sahibi, ürün sahibi ve paydaşlar, bir ürün vizyonu ve ürün yol haritası oluştururlar. Vizyon, projenin genel odak noktasını sağlarken, yol haritası sürümler ve hedefleri hakkında rehberlik sağlar. Scrum'un başlıca tekniklerinin ürün talep listesi (product backlog), sprintler ve günlük scrum toplantıları (scrum, 15 dakikalık günlük ekip toplantısı) olduğunu belirtirler. İş gereksinimleri açısından, ürün talep listesi Scrum içinde özel bir rol oynar çünkü tüm gereksinimler, ürün için gerekli veya faydalı olarak kabul edilenler, ürün talep listesine listelenir. Ayrıca bu liste, tüm özelliklerin, fonksiyonların, iyileştirmelerin ve hataların öncelik sırasını içerir. Sprint (Şekil 3'e bakınız), 30 günlük bir geliştirme dönemidir. Her sprint için, talep listesinden en yüksek öncelikli görevlerin sprint talep listesine taşındığını açıklar. Bir sprint başladıktan sonra karşılanması gereken gereksinimlerin değiştirilmemesi gerekir. Sprintin sonunda, müşteriye yeni işlevselliği gösteren ve geri bildirim toplayan bir sprint gözden geçirme toplantısı yapılır. İki sprint arasında müşterinin bir sonraki sprint için gereksinimleri yeniden önceliklendirmesi için tam esneklik bulunur.

2.1.4. Çevik Yaklaşım Sorunları :

Çevik yaklaşımın başlıca karşılaştığı sorunlar şunlardır. İlk olarak, çevik yaklaşımın önceliği olan yazılımın hızlı geliştirilmesi, çevik projelerde mühendislik değerlerine olan vurgunun eksikliği nedeniyle yazılımın bütünlüğünü tehdit edebilir. Çevik süreçlerde işlerin hızlı ilerlemesi, tasarım bütünlüğüne veya sistem düzeyindeki sorunların erken tespitine yeterince odaklanmadığında, sonradan düzeltmeler gerekebilir. Bu, çevik projelerde yazılım mühendisliği prensiplerine dikkat edilmediğinde ortaya çıkabilecek bir sorundur.

İkinci olarak, çevik yaklaşımın, projelerin uzmanlara bağımlı hale gelmesine neden olabilecek, bilgi paylaşımına dayalı bir işbirliği odaklı olma eğilimi vardır. Bu, projelerin sadece belirli uzmanlara dayalı hale gelmesine yol açabilir ve

bu uzmanlar olmadan işlerin yürütülmesi zorlaşabilir. Çevik yaklaşımda tacit (bilinçdışı) bilgiye odaklanma, projeleri uzmanlara bağımlı hale getirebilir.

Üçüncü olarak, çevik süreçlerin, güvenlikle ilgili veya kritik sistemlerin kalitesini belirlemek için yetersiz olduğu düşünülmektedir. Çevik süreçlerin daha bilgi eksikliği ve hızlı iterasyonlarla karakterize edilmesi, güvenlik açısından kritik sistemlerin kalitesini belirlemek için yetersiz olabilir. Bu tür sistemler için daha fazla dikkat ve ayrıntılı bir kalite değerlendirmesi gerekebilir.

Son olarak, büyük ölçekli yazılım projeleri için çevik yaklaşımın uygulanması zor olabilir. Birden fazla ekip ve çok sayıda test gerektiren karmaşık sistemler, çevik süreçlerin koordinasyonunu ve yönetimini zorlaştırabilir. Büyük ölçekli, çoklu ekip projeleri için çevik süreçlerin uyarlanması ve yönetilmesi karmaşık bir iş olabilir.

Bu nedenle, çevik yaklaşımın bazı durumlarda karşılaşabileceği zorluklar ve eksiklikler vardır ve bu sorunlar projelerin özel gereksinimlerine ve büyüklüklerine bağlı olarak değişebilir.

3. BULGULAR VE TARTIŞMA

3.1. Çevik Yaklaşımlar ve Şelale Modeli: Proje Özelliklerine Göre Bir Karşılaştırma

Unhelkar (2016), geleneksel şelale yaşam döngüsünü yazılım geliştirmenin lineer, aşamalı bir yaklaşımı olarak tanımlarken, çevik yaklaşımın bu geleneksel geliştirme yaşam döngüsüne tamamen zıt olduğunu düşünmektedir. Aşağıdaki bölümler, projelerin nasıl geliştirilmesi gerektiği konusunda bir ışık tutacaktır, Çevik mi yoksa Şelale mi? A. Prototip projeler Unhelkar (2016), en çok bir Çevik yaklaşımdan fayda sağlayan projelerin "yeşil saha" geliştirme projeleri olduğunu belirtir. Yeşil saha projesi, tamamen yeni bir ortam için bir sistem geliştirmeyi amaçlayan bir projeyi ifade eder. Ayrıca, yazılım geliştirmeye yönelik Çevik yaklaşımlar, kodlamanın faaliyetlerin merkezinde olduğu projelerde en elverişli olanlardır. Genellikle, beş programcıdan oluşan ve yaklaşık 6 ay süren küçük bir proje, Çevik prensip ve uygulamalarını yaygın bir şekilde kullanmak için ideal bir konumda olurdu. Ayrıca, bu tür projeler, Çevik yaklaşımın kendisiyle denemeler yapmak veya yeni, büyük bir geliştirmenin bir parçası olabilirler. Petersen ve ark. (2009), tarafından belirtildiğine göre, Şelale Model, büyük sistemlerle uğraşırken ayrıntılı olarak yazılım sisteminin mimari ve yapısını planlama konusunda özellikle önemli olan öngörülebilir ve dikkatli bir model sunmaktadır. B. Çevik ve plana dayalı proje özellikleri Hangi projelerin hangi geliştirme metodolojisinin seçilmesi konusunda ne tür projelere ait özellikler fark yaratacak?

1. **Başlıca hedefler:** Boehm (2002), daha geleneksel plana dayalı yöntemler için önemli bir hedef setinin öngörülebilirlik, tekrar edilebilirlik ve optimizasyon olduğunu belirtir. Buna karşılık Çevik yaklaşım, hızlı değer ve değişikliklere yanıt verme üzerine odaklanmaktadır.

2. **Boyut:** Plan odaklı yöntemler, büyük projelere göre Çevik projelere göre daha iyi ölçeklenir. Bir projenin yetkilendirilmesi ve başlatılması için ortalama bir kişi ayırmak için bir aylık süre gerektiren bürokratik, plana dayalı bir organizasyon, küçük projelerde çok verimli olmayabilir Boehm (2002).

3. **Müşteri ilişkileri:** Çevik yöntemler, müşterilerin geliştirme ekibiyle işbirliği yapmak ve müşterilerin örtülü bilgisinin uygulama sürecinin tamamı için yeterli olduğu durumlarda en iyi şekilde çalışır Boehm (2002). Bu yöntem örtülü bilgi eksikliklerine neden olma riski taşırken, plana dayalı yöntemler, belgelendirme ve mimari inceleme kurulları ile eksiklikleri telafi etmek ve bağımsız uzman proje incelemeleri aracılığıyla eksiklikleri gidermek için kullanır Boehm (2002).

4. **Planlama ve kontrol:** Unhelkar (2016)'a göre, formel proje yönetimi yazılım projesinin başarılı bir şekilde tamamlanmasında önemli bir rol oynamaktadır. Proje yönetimi, dikkatli planlama, tahmin, koordinasyon, izleme ve kontrol gerektirir. Bu konular resmen Şelale Model'de kapsanmaktadır. Çevik, planlama sürecine belgelemeye dayalı sonuçlardan daha fazla değer verir (Unhelkar**,** 2016).

5. **İletişim:** Çevik, yüz yüze iletişimi desteklerken, Şelale Model belgelenmiş bilgiye vurgu yapar.

6. **Gereksinimler:** Önemli olmayan ve çözüm bağımsız, başlangıç gereksinimlerinin analizi saf bir Çevik uygulayıcıları tarafından doğrudan kullanılmaz (Unhelkar**,** 2016). Ağır, resmi Şelale Model, hızla değişen gereksinimlerle başa çıkmada zorluklarla karşılaşabilir. Öte yandan, mimari gereksinimleri öngördüğünde ve karşılamak için uygun olduğunda, plana dayalı yöntemler bile milyonlarca satırlık uygulamaları bütçe ve zaman içinde tutabilir.

7. **Geliştirme:** Çevik, kapsamlı belgeleme yerine çalışan yazılıma ve sadeliğe vurgu yapar ve yapılmayan işin miktarını maksimize ederken. Buna karşılık Şelale Model, geliştirme sırasında önemli bir parçası olduğu için yazılım mimarisine büyük bir vurgu yapar.

8. **Test:** Beznosov ve Kruchten (2004) 'e göre, geleneksel güvence yöntemleri tasarım ve mimari prensipleri, dinamik test, statik analiz ve içsel ve üçüncü taraf incelemelerini, değerlendirmeyi ve zayıflık testini içerir. Bu yöntemler daha çok belgelenmiş ve mimariye odaklandığından Şelale

geliştirmeye çok daha uygundur. Öte yandan, Çevik Yaklaşımlar, iç tasarımı ve kod incelemeyi kolaylaştırır ve geliştiricileri kodlama standartlarını benimsemeye teşvik eder, ancak mimari veya belgelemeye odaklanmaz (Beznosov ve Kruchten, 2004).

9. Müşteriler: Çevik, geliştirmeye özgü, birlikte çalışan ve bilgili müşteriler gerektirir. Şelale Model, yeterince bilgili müşterileri gerektirir.

10. Geliştiriciler: Çevik bir projede geliştiricilerin çevik, bilgili, bir arada çalışan, iş birliği yapan, dostça, yetenekli, becerikli ve iletişim kurabilen olmaları gerekmektedir. Çevik yaklaşımlar, geliştiriciler, testçiler, konu uzmanları ve mimarlar gibi çeşitli alanlarda uzmanlaşmış ekip üyelerini vurgular **:** Unhelkar (2016). Şelale modelindeki geliştiriciler ise plana dayalı, yeterli yetkinliklere sahip ve dış bilgilere erişim sağlayan kişiler olmalıdır.

11. Kültür: Çevik yöntemler, "kaos üzerine kurulu" bir kültürde daha iyi bir şekilde başarılı olacaktır, bu, Boehm ve Turner (2003), tarafından belirtildi. Şelale Modeli için ise tam tersi geçerlidir, yani "düzen üzerine kurulu" bir kültürde daha iyi işler.

Boehm ve Turner (2003), tarafından hazırlanan Tablo 1, yukarıdaki unsurların bir özetidir. Bu tablo, proje hedefleri ve boyutları, proje yönetimi tarzı, proje araçları kullanımı, personel ve kültür, müşteriler açısından tipik özelliklerin genel bir bakışını sunar hem Çevik projeler hem de plana dayalı projeler için geçerlidir. Bu tablo, personel geliştiricilerinin beceri seviyelerini tanımlama konusunda Tablo 2'ye başvurur.

Williams (2004;2016) plan odaklı Yazılım Geliştirme Metodolojileri ile çeşitli Çevik gelişim türlerini anketlerinde tanıtır ve ayrıntılı bir bakış sunar. Çevik Metodolojiler ve plan odaklı metodolojilerin popüler olduğu, geliştiriciler tarafından nasıl algılandığı ve yöntemlerin nasıl birleştirilebileceği konularında fikirler almak için West ve diğerleri'ne (2010), başvurun. Bir proje, Çevik Metodolojiler, Şelale Model veya ikisi arasında bir model aracılığıyla geliştirilmeli mi kararı vermek için bir çerçeve, Boehm ve Turner (2003) tarafından sunulmuş ve Çevik Metodolojiler ve plan odaklı yöntemleri dengelemek için düşünülmesi gereken çizgileri belirtmiştir. Dağıtılmış bir ortama Çevik Metodolojiler süreçleri uyarlamak hakkında bir makale, Young ve Terashima (2008) tarafından yayımlanmıştır.

Tablo 1. Çevik ve plana dayalı geliştirme temelleri.

Proje Özellikleri	Çevik Geliştirme Temelleri	Planlı Geliştirme Temelleri
Uygulama		
Ana hedefler	Hızlı değer oluşturma, Değişikliklere yanıt verme	Tahmin edilebilirlik, Kararlılık, Yüksek güvence
Büyüklük	Daha küçük ekipler ve projeler	Daha büyük ekipler ve projeler
Yönetim		
Müşteri ilişkileri	Özgü müşteriler, öncelikli artışlara odaklanmış	İhtiyaca göre müşteri etkileşimleri, sözleşme hükümlerine odaklanmış
Planlama ve kontrol	İçselleştirilmiş planlar, nitel kontrol	Belgelenmiş planlar, nicel kontrol
İletişim	Elde edilen kişisel bilgi	Açıkça belgelenmiş bilgi
Teknik		
Gereksinimler	Öncelikli gayri resmi hikayeler ve test vakaları, öngörülemeyen değişikliklere tabi tutuluyor	Resmi proje, yetenek, arayüz, kalite, öngörülebilir evrim gereksinimleri
Geliştirme	Basit tasarım, kısa dönemler, refaktoringin düşük maliyetli olduğu varsayılır	Kapsamlı tasarım, uzun dönemler, refaktoringin maliyetli olduğu varsayılır
Test	Çalıştırılabilir test vakaları gereksinimleri tanımlar, test edilir	Belgelenmiş test planları ve prosedürleri.
Personel		
Müşteriler	Dedicated, colocated Crack[1] uygulayıcıları	Crack[2] uygulayıcıları, her zaman aynı yerde bulunmazlar
Geliştiriciler	En az %30'u tam zamanlı olarak 2. ve 3. seviye uzmanlar; 1B veya -1. seviye personel yok[3]	%50'si 3. seviye personellerin başında; %10'u boyunca; %30'u 1B seviyesinde çalışabilir; -1 seviyesinde personel yok[4]
Kültür	Çok sayıda serbestlik derecesi ile konfor ve yetkilendirme (kaostan zevk alma)	Politika ve prosedürler çerçevesinde konfor ve yetkilendirme (düzen ve düzen üzerinden zevk alma)

[1] ve [2] İşbirlikçi, Temsilci, Yetkilendirilmiş, Bağlı ve Bilgili.
[3] ve [4] Tablo II'ye bakın, 'Personel yazılım beceri seviyesi'

Tablo 2. Personel yazılım beceri seviyesi.

Seviye	Karakteristikler
3	Yeni ve önceden karşılaşılmamış bir duruma uymak için bir yöntemi değiştirebilen.
2	Bir yöntemi önceden karşılaşılan yeni bir duruma uydurabilme yeteneği.
1A	Eğitimle, örneğin hikayeleri artışlara uygun şekilde boyutlandırma, desenler oluşturma, karmaşık COTS entegrasyonu veya karmaşık refactoring gibi takdiri yöntem adımlarını gerçekleştirebilme yeteneği. Deneyim kazandıkça, Seviye 2 olabilir.
1B	Eğitimle, basit bir yöntemi kodlama, basit refactoring, kod standartlarını takip etme ve CM prosedürlerini çalıştırma gibi prosedürel yöntem adımlarını gerçekleştirebilme yeteneği. Deneyim kazandıkça, bazı Seviye 1A becerilerini ustalaşabilir.
-1	Teknik becerilere sahip olabilir, ancak iş birliği yapma veya paylaşılan yöntemlere uyma konusunda isteksiz veya yeteneksiz olabilir.

4. SONUÇ

Sonuç olarak bu çalışmada iki Yazılım Geliştirme Modeli (YGM) tanıtılmıştır. Plan odaklı Şelale Model (Waterfall Model) ve uyarlamalı Çevik Metodolojiler (Agile Methodologies) olmak üzere. Her iki modelin de kullanım alanları, avantajları ve dezavantajları bulunmaktadır. Küçük projelerin neredeyse her zaman bir Çevik Metodoloji yaklaşımı için uygun olduğu ve neredeyse asla Şelale Model bir yaklaşım için uygun olmadığı tespit edilmiştir. Hem Şelale Model hem de Çevik Metodolojiler, orta büyüklükteki projelerle uğraşırken zorluklar yaşandığı saptanmıştır. Zorlayıcı bir Şelale Model, nispeten basit bir projeye gereksiz karmaşıklık ekleyebilirken, aynı projeye esnek bir Çevik Metodoloji yaklaşımın daha uygun olduğu da diğer sonuçlar arasındadır. Büyük ve karmaşık projeler, farklı uygulama parçaları üzerinde aynı anda çalışan çoklu ekiplerin olduğu projeler olarak kabul edilir ve genellikle Şelale Model bir projedir. Şelale Model ve Çevik Metodoloji gibi sınıflandırılması zor diğer YGM'ler için daha fazla araştırma gerekmektedir.

5. KAYNAKLAR

Beznosov, K. & Kruchten, P. (2004). Towards agile security assurance. In Proceedings of the 2004 workshop on New security paradigms, 47–54. ACM.

Bhuvan, U. (2016). The Art of Agile Practice: A Composite Approach for Projects and Organizations. CRC Press.

Boehm, B. (2002). Get ready for agile methods, with care. Computer, 35(1):64–69.

Boehm, B. & Turner, R. (2003). Using risk to balance agile and plan-driven methods. IEE Computer Science.

Cohen, D., Lindvall, M. & Costa, P. (2003). Agile software development. DACS SOAR Report, 11.

Fowler, M. & Highsmith, J.(2001). The agile manifesto. Software Development, 9(8):28–35.

Ganis, M. (2010). Agile Methods: Fact or Fiction. tcf.pages.tcnj.edu adresinden 12.01.2024 tarihinde erişilmiştir.

Highsmith, J. & Cockburn, A. (2001). Agile software development: The business of innovation. Computer, 34(9): 120–127.

Kai, P., Claes, W. & Dejan, B. (2009). The waterfall model in large-scale development. In International Conference on Product-Focused Software Process Improvement, 386–400. Springer.

Matt, G. (2010). Agile methods: Fact or fiction.

Munassar, N. M. A & Govardhan, A. (2010). A Comparison between Five Models of Software Engineering. International Journal of Computer Science Issues (IJCSI) 7 (5), pp. 94-101.

Nabil, M., A., M. & Govardhan, A. (2010). A comparison between five models of software engineering. IJCSI, 5:95–101, 2010.

Petersen, R., C., Roberts, R., O., Knopman, D., S. & Boeve, B., F. (2009). Mild Cognitive Impairment. Archives of Neurology 66(12):1447-55. DOI:10.1001/archneurol.2009.266

Rawsthorne, D., & Shimp, D (2009). Scrum in a nutshell. https://www.scrumalliance.org/community/articles/2009/december/scrum-in-a-nutshell .

Royse, W. (1970). Managing the Development of Large Software System. Proceedings of IEEE WESCON, August, 1-9.

Schwaber, K. (1995). Scrum development process, oopsla'95 workshop on business object design and implementation. Austin, USA.

Turk, D., France, R. & Rumpe, B. (2014). Assumptions underlying agile software development processes. arXiv preprint arXiv:1409.6610.

Unhalker, B. (2016). The Art of Agile Practice A Composite Approach for Projects and Organizations. Computer Science, Economics, Finance, Business & Industry https://doi.org/10.1201/b13085

West, D., Grant, T., Gerush, M. & D'silva, D. (2010). Agile development: Mainstream adoption has changed agility. Forrester Research, 2(1):41.

Williams, L. (2004). A survey of plan-driven development methodologies.

Williams, L. (2007). A survey of agile development methodologies. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=f97bfc7125c862a3411862d0d522ec6ebc1b9764

Winston, W, R. (1970). Managing the development of large software systems. In proceedings of IEEE WESCON, 8, 328–338. Los Angeles.

Young, C. & Terashima, H. (2008). How did we adapt agile processes to our distributed development? In Agile, 2008. AGILE'08. Conference, 304–309. IEEE.