Yazılım – yaşam döngüsü bir yazılımın fikir aşamasından hayata geçtiği süreci ve hayata geçtiği süreçten sonraki dönemi kapsayan oluşumdur. Bir yazılımın ortaya çıkabilmesi için öncelikle bir fikrin ya da bir sorunun olması gerekir. Bunun sonucunda eğer yeni bir fikir varsa yazılım başlı başına sanal alemde bir iş fikri olabileceği gibi önceden var olan bir işin sanal aleme taşınması gibi bir işleve de sahip olabilir. Buna örnek olarak tamamıyla internet dünyası üzerine kurulmuş olan iş fikirlerini örnek verebiliriz. Ya da önceden var olan bir iş fikrini çok daha geniş bir alana duyurmak için de yazılım gerekli olacaktır. Tabi bunlar gibi yeni fikirler ortaya atılıp yazılım üretilebileceği gibi herhangi bir iş ile ilgili bir sorunla karşılaşıldığında da yazılıma başvurabileceğimizden söz etmiştik. Özellikle de yaşadığımız yüzyılda çoğu şey otomasyon haline gelmişken ve makine gücünün insan gücünden üstün hale gelmişken bundan daha çok bahsetmemiz gerekir.

Bir yazılımın oluşumunu ele alalım. Ortada bir iş fikri ya da bir sorun olduğunu düşünürsek burada yazılım yaşam döngüsünde başlamamız gereken nokta ilk olarak gereksinimleri belirlemektir. Gereksinimleri belirlemek bir nevi o yazılımın ne işe yarayacağını, hangi işleve sahip olacağını belirlemek demektir. Yazılımı oluştururken karşıdaki müşterinin fikrinden analizler yapıp müşterinin istediklerine, beklentilerine göre çıkarımlar yapılmalıdır. Bu aşamada önemli olan şey müşterinin söylediklerine göre mümkün olabilecek en faydalı ve yapılabilir yazılım fikrini müşteriye sunabilmektir. Yani burada önemli olan müşterinin ne istediği değil, müşterinin istediklerine göre çıkabilecek en mantıklı ve kabul edilebilir yazılımın fikrini ortaya koyabilmektir. Bunun için de müşterinin gereksinimlerini doğru olarak algılayabilmek gerekir. Yazılımın işlevi müşterinin gereksinimlerinde ne fazla ne de eksik olmalıdır. Yazılımın işlevinin eksik olması durumunda beklenileceği üzere başarısız bir proje ortaya çıkacağı gibi fazla olması durumunda ise ortaya çıkacak hedef ürün amacından sapabilir.

Gereksinimleri belirleme kısmını tamamlayıp yazılımın işlevlerini belirledikten sonra ele almamız gereken kısım analiz kısmıdır. Bu kısım ile gereksinimleri belirleme kısmı birbiriyle iç içe konulardır. Gereksinim kısmında belirlediğimiz şartları sağlayan yazılımı oluşturmak için bu konudaki pazarları, yazılımın oluşturulacağı süreyi, bu sürede yazılımın maliyetinin ne kadar olacağını iyi analiz edip müşteriye doğru bir biçimde aktarmak gereklidir. Pazar analizi kısmında dikkat edilecek konuların başında rakip yazılımlar gelmektedir. Eğer yazılım yeni bir fikir ise öncelikle bu fikrin daha önce düşünülüp düşünülmediği kısmı önemlidir. Eğer fikir daha önce hayata geçirilmişse rakip yazılımlar incelenmeli ve bunlar ile ilgili analizler yapılmalıdır. Rakip yazılımlar ile ilgili analizlerde nasıl bir kitleye hitap ettiği, şu ana kadar nasıl bir pazara sahip olduğu gibi konular ele alınabilir. Eğer fikir daha önce hayata geçirilmemişse oluşturulacak yazılımın nasıl bir kitleye hitap etmesi gerektiği, bu kitleye nasıl ulaşacağı, kendisine piyasa da nasıl bir yer edinebileceği ya da piyasada ne kadar sürekli kalabileceği gibi konuların ele alınması yararlı olabilir. Daha önce de söz ettiğimiz gibi bir yazılım projesinin hayata geçmesi için yeni bir iş fikrinin oluşması şart değildir. Daha önce var olan bir işin sanal dünyaya aktarılması ya da daha önce var olan bir işte üretimin hızlandırılıp maliyetin azaltılması gibi konularda da yazılım devreye girebilir. Bunun gibi konularda ise işin sanal dünyaya geçtiğinde nasıl bir pazara sahip olacağı, yapılacak yazılımın sanal aleme açıldığında nasıl bir tepki bulacağı gibi konularda araştırma ve analizler yapılmalıdır. Eğer bir iş fikrinde üretimde yazılım kullanılacak ise daha önceki maliyetler ile yazılım devreye girdikten sonra oluşacak maliyetler saptanmalı, insan gücünden ne kadar tasarruf edileceği gibi konular ele alınmalıdır. Pazar analizi kadar önemli bir diğer konu ise yazılımın ne kadar sürede yapılacağı ve ne kadar maliyeti olacağı konusudur. Burada önemli olan şey müşteriye somut olarak yapılan planın anlatılması ve müşteriye bilgilerin verilmesidir. Yapılan planlamada gerçekçi rakamlar kullanılmalı ve zamanında teslim edilecek şekilde devreye sokulmalıdır.

Yazılımın gereksinimleri belirlenip analizleri ve planlaması yapıldıktan sonra hayata geçiş süreci yani projenin oluşturulma aşaması başlamaktadır. Bu aşamada tamamı ile planlı bir şekilde hareket edilmelidir çünkü bu aşamada yaşanılacak bir aksaklık yazılımın teslim tarihini geciktirebilir ya da yazılımın işlevlerini yerine getiremeyecek bir halde ortaya çıkmasına neden olabilir. Proje geliştirilirken dikkat edilmesi gereken konu yapılan projenin güncelliğini koruyabilmesidir. Eğer yapılan proje en ufak bir yeniliği kaldıramayacak kapasitede ise bu projenin çok kullanışlı olduğunu söyleyemeyiz. Bunun yanında elbette yapılacak güncelleme ve bakımlarda maliyetin düşük tutulması da önemlidir. Yazılımın ana hedefi değişmediği sürece yapılacak güncellemeleri ve geliştirmeleri kaldıracak kapasitede olması beklenir. Projede bir diğer önemli konu ise yazılımın vereceği hatalardır. Yazılım elbette hata verebilir, nihayetinde yazılımı geliştirenlerin de hata payı olduğu için bu gayet normaldir. Ancak burada önemli olan hata payını en aza indirgemektir. Programda hata payını en aza indirgemek için bazı etkenler vardır. Bu etkenlerden en önemlisi programı kaç kişinin kullanacağını ve programın sınırlarını saptayabilmektir. Programın yoğun olacağı zaman dilimlerini belirleyip bu zamanlarda önlem almak bu konudaki hata payını indirgemekte yardımcı olacaktır. Bunun yanında programın kullanışlı olması ve program kodlanırken programın temiz bir şekilde kodlanması, bu sayede program herhangi bir şekilde başka bir yazılımcı tarafından ele alındığında yazılan kodların anlaşılabilir olması sağlanmalıdır.

Yazılım yaşam döngüleri oluşturulurken işleyen bu süreçte artık kalıplaşmış bazı modeller kullanılmaktadır. İlk olarak 1960 yıllarında ortaya çıkan gelişigüzel model ile hayatımıza giren yazılım geliştirme modelleri günümüzde hala devam etmektedir. Gelişigüzel model belli kalıpları olmayan bir model olduğu için bu yöntemi bir model olarak ele almak çok doğru olmayabilir. Kullanılabileceği alanlar genellikle tek kişiye bağlı ve tek kişinin yönettiği basit projelerde kullanılır. Bu yöntemde bakım güncelleme gibi işlemleri yapmak oldukça zorlaşır. Bu yöntemin eksiklerinden yola çıkarak 1970 yıllarında geliştirilen bir diğer yazılım geliştirme modeli ise Barok modelidir. Bu modelin ele alınacak en temel unsuru yazılımın oluşturma adımlarını çok tekdüze ele alıyor olmasıdır. Bunun sonucunda oluşacak bir hatada geri dönüşlerin çok zor olması sorunuyla karşılaşılabilir. Yazılımın gerçekleştirilme aşamasına ağırlık veren bir model olduğu için ve dokümantasyonu projeden ayrı olarak ele aldığı için günümüz yazılım geliştirme projelerinde tercih edilen bir yöntem değildir. Günümüzde kullanılmamasının yanı sıra elbette ki daha önce ortaya çıkan gelişigüzel modele göre daha sağlıklı bir yöntemdir. Gelişigüzel yönteme göre doğrusal da olsa yazılım yaşam döngülerinin temel adımları takip edildiği için Barok modeli daha avantajlıdır. Bundan sonra bahsedeceğimiz model çağlayan yaşam döngü modelidir. Bu modeli diğerlerinden ayıran en önemli özelliği geçmişte kullanılan en popüler yazılım geliştirme modeli olmasıdır. Geçmişte en popüler olmasının en önemli sebebi ise kullanımının basit oluşu, hemen anlaşılabilir olması ve yönteminin kolay olmasıdır. Bu modeli Barok modelinden ayıran temel etken dokümantasyonu yazılımın bir parçası olarak görmesi ve yazılım geliştirme aşamalarının sadece doğrusal olarak değil geri dönüşlerinin de tanımlı olmasıdır. Genellikle çok büyük olmayan ve gereksinimi basit ve belirgin olan projelerde daha kullanışlı olduğu gözlemlenmiştir. Bu modelin metodu ise yazılım yaşam döngüsü aşamalarının en az birer kez tekrarlanmasından meydana gelir. Bir aşama bitmeden diğer aşamaya geçilmez, eğer bir aşamada dokümantasyon ve test olmamışsa o aşama tamamlanmış olarak kabul edilmez. Ancak bu modelin kullanılabilmesi için müşteriden gereksinimleri ve istekleri çok detaylı olarak alınmalıdır. Bu modelin küçük projelerde kullanılmasındaki amaç ise büyük yazılım projelerinde sürekli farklı gereksinimler ve farklı ihtiyaçlar ortaya çıkabilme ihtimalinin daha yüksek olmasıdır. Çağlayan(şelale) modeli daha çok statik olduğundan bu durumlarda biraz elverişsiz kalmaktadır. Ancak şelale modeli her ne kadar büyük projeler için elverişsiz olsa da temel bir model olması, basitliği ve diğer modellerin temeli gibi bir niteliğe sahip olması açısından oldukça önemlidir. Günümüzde kullanımı gittikçe azalmaktadır bunun nedeni ise günümüzde geliştirilen yazılımların oldukça detaylı ve geri dönüşlerin oldukça fazla olduğu projeler olmasıdır. Bundan sonra ele alacağımız model ise V süreç modelidir. V şeklindeki bu modelin sol tarafını üretim, sağ tarafını kontrol(sınama) olarak düşünebiliriz. V süreç modelinin bazı temel çıktıları vardır. Bunlardan kullanıcı modeli, mimari model ve gerçekleştirim modeli olarak bahsedebiliriz. Kullanıcı modelinde proje geliştirilirken aynı zamanda kullanıcı ile olan etkileşimler ele alınmakta ve bu sayede ürün geliştirilirken kullanıcı tarafından bakılarak planlama yapılmaktadır. Mimari modelde ise yapılan yazılım projesinde ana sistemin ve ona bağlı alt sistemlerin sınama işlemleri gerçekleştirilir. Gerçekleştirim modelinde ise yazılım modüllerinin kodlanması ve sınanmasına ilişkin fonksiyonlar ele alınır. Projede kullanıcının katkı payını artırmaya yönelik bir modeldir. Projenin iki aşamada gerçekleşmesine uygun bir modeldir. İlk aşamada kullanıcıya yönelik model ele alınarak iş analizlerine göre bir model tasarlanıp, ikinci aşamada bu modelden de yararlanılarak gerçekleştirim modeli oluşturulmaktadır. Modelin dezavantajı olarak aşamalar arası tekrarlamayı kullanmamasını örnek olarak verebiliriz. Çağlayan model ile karşılaştıracak olursak burada V modelin öne çıkan özelliğinin her aşamada sınama gibi bir özelliği olmasıdır. Bunların dışında günümüz yazılım süreçleri en çok yaklaşan bir diğer yazılım yaşam döngüsü modeli ise Helezonik (spiral) modeldir. Bu modelde bir döngü esastır. Bu model kendi içerisinde 4 ana başlıkta incelebilir. Bunlar planlama, risk analizi, kullanıcı değerlendirme ve üretim kısımlarıdır. Bu kısımların her biri kendi içerisinde farklı başlıklara ayrılır. Her bir aşama için planlama, risk analizi, üretim ve kullanıcı değerlendirmesi yapılır ve bunların uygulandığı her aşamada bir prototip oluşturularak sonuçta bu prototipler birleştirilir ve ana ürün ortaya çıkarılır. Bu modelin avantajlarının başında üretim süresi boyunca kullanıcının bu ürünü test etme olanağı bulabiliyor olmasıdır. Bunun sağladığı avantaj ise eğer projede istenmeyen bir durum var ise bunun erken aşamada fark edilip daha düşük maliyetle bu sorunun çözüme kavuşturulmasıdır. Burada sınanma aşamalarının erken başlaması ve her aşamada yapılması sadece projenin sorunsuz bir şekilde ortaya çıkmasını değil üretim aşamasında sorumlu mühendislerin işinin kolaylaşması bakımından da oldukça önemlidir. Bu kısma kadar incelediğimiz modellerden farklı olarak ele alabileceğimiz bir model olarak Artımsal Geliştirme Süreç Modelini örnek olarak verebiliriz. Bu modelin şimdiye kadar incelediğimiz modellerden farklı olmasının ana nedeni projeyi bir bütün olarak değil de parça parça teslim etme prensibine göre çalışmasıdır. Modelde gereksinimler ön plandadır. İşlevselliği en önemli olan parça en erken olarak teslim edilir, daha sonrasında teslim edilen parçalar birbirini bütünleyecek biçimdedir ve diğer parçalar teslim edildikçe programın işlevselliği gitgide artmaya başlar. Bu modelin avantajı olarak yapılacak değişikliklerde sorun yaşanmamasını gösterebiliriz. Eğer kullanıcı tarafından bir değişiklik yapılmak istenirse bu değişikliğin bir sonraki teslim edilecek parçada hayata geçirilmesi zor olmayacaktır. Bu modelin kullanıldığı projelerin genelde uzun sürede bitebilecek ve bazı işlevleri eksik olsa bile amacını yerine getirerek çalışabilecek projeler olduğunu söyleyebiliriz. Yani bir taraftan ürün ortaya çıkmış ve kullanılmasına devam edilirken diğer yandan da üretimin devam ettiğini söyleyebiliriz. Bu modelin bir diğer avantajı olarak da proje başarısız olsa bile bunu erkenden teslim edilen ana işlevleri karşılayan parçalardan anlayarak üretimi başlangıçtayken bitirme şansı olmasıdır. Son olarak bahsetmemiz gereken yazılım modeli olarak kodla düzelt modelinden bahsedebiliriz. Bu model de gelişigüzel modeli gibi küçük programlarda kullanılır. Öncelikle program geliştirilir ve bitirilir. Sonra hatalara bakılır ve düzeltilir. Sınama aşaması olmadığından dolayı yapılacak düzeltmeler çok maliyetlidir. Bu yüzden de yazılım geliştirmenin en kolay ve en pahalı yolu olduğunu söyleyebiliriz.

Buraya kadar bahsettiğimiz yazılım yaşam döngü modelleri daha çok eskiye dönük daha çok temelleri barındıran modellerdi. Günümüzde kullanılan yazılım modelleri ise günümüz yazılımlarına daha uygun olan çevik yazılım geliştirme olarak adlandırılan metotlar kullanılmaktadır. Çevik yazılım geliştirme metotlarının asıl hedefleri olarak piyasaya daha seri ürün çıkarabilme, gelişen ve değişen isteklere hızla yanıt vermek gibi unsurları ele alabiliriz. Bu yazılım geliştirme metodunda projenin boyutu ya da işlevleri ne olursa olsun proje aynı işlemlerden geçirilerek sonuca ulaşılır. Proje küçük yinelemelere ayrılı ve her yineleme bir proje gibi ele alınarak geliştirilir. Her yinelemenin sonunda da yapılan işte projede ne kadar ilerleme kaydedildiği müşteriye aktarılır. Her yinelemenin kendi içerisinde çalışan bir sistem olması sonucu müşteriye sürekli çalışan bir yazılım teslim edilmesi açısından artımlı model ile bağdaştırılabilir. Genel olarak çevik yazılımlar verimliliği yüksek, hata payı düşük, hızlı ve ucuz çözümler sağladığı için günümüzde en çok kullanılan metotlardır.

2001 yılında Kent Beck ve 16 arkadaşı tarafından oluşturulan çevik manifesto geleneksel yazılım geliştirme modellerinin aksine süreçler ve araçlar yerine bireyler ve etkileşimler, kapsamlı belgeler yerine çalışan yazılım, sözleşme görüşmeleri yerine müşteri ilişkileri, plan izleme yerine değişikliğe açıklığın daha önemli olduğunu vurgulamıştır.

Çevik yazılım geliştirmenin kendine has bazı temel prensipleri vardır. Bunların başında ise müşteriye hızlı ve sürekli olarak kullanılabilir yazılım teslimatı yapmak gelmektedir. Bir diğer önemli özelliği ise değişime sadece erken safhada yada gereksinimleri belirlerken değil projenin ilerleyen safhalarında da değişikliğe açık olabilmeleridir. Bu esnek proje şartlarına karşılık aynı zamanda mümkün olan en kısa zaman aralıklarında çalışan, kaliteli yazılım teslimatı yapmak da esastır. Burada bahsettiğimiz kaliteli yazılım kısmı oldukça önemlidir. Günümüzde yapılan bir hata da ürünü hızlı elde etmek için yazılımın kalitesinin düşürülmesi ya da bunun önemsenmemesidir. Yazılan kodun temiz yazılmış olması, anlaşılabilir olması, yeniliklere açık olması, dinamik olması gibi konular yazılımın kalitesi alanına girmektedir. Ayrıca çevik yazılım geliştirmede bir bütünlük söz konusudur. Analistler, uzmanlar, yazılımcılar, testçiler gibi tüm ekiplerin bir arada ve koordineli çalışmaları beklenir. Ekip üyelerinin iyi seçilmesi de bu konuda önemli olacaktır. Motivasyonu yüksek, bilgili, takım çalışmasına uygun kişiler bir araya geldiğinde kaliteli bir projenin ortaya çıkma ihtimali çok daha yüksektir.

En yaygın uygulanan çevik metodolojilerden bahsedecek olursak bunlar Extreme Programming. (XP), SCRUM, Agile Unified Process, Feature-Driven Development (FDD), Test-Driven Development, LEAN Development, Dynamic System Development Methodology (DSDM) ve Microsoft Solution Framework (MSF) olduğunu söyleyebiliriz.

Extreme Programming Kent Beck tarafından 1999 yılında bir yazılım geliştirme metodu olarak ortaya çıkarılmıştır. XP kolay, grup içi iletişime önem veren, geri dönüşlerin daha fazla olmasına imkan sağlayan bir yazılım geliştirme yöntemidir. XP’nin 4 temel unsuru olduğunu söyleyebiliriz: iletişim, basitlik, geri bildirim, cesaret. XP’nin ele aldığı en temel konu iletişim konusudur. Birbirleri ile iletişim halinde olan ekip üyeleri ve ekip üyeleri ile de iletişim halinde olan müşteriyi öngörür. Diğer ele aldığı konu olan basitliği ise o an ortada olan bir problemi çözebilecek en basit yöntemi kullanarak çözüme ulaştırma metodunu kullanarak gerçekleştirir. Karmaşık çözümler XP’nin mantığına aykırıdır. XP’nin belki de en can alıcı noktası cesaret kısmıdır. Projeler ne kadar karmaşık olursa olsun bunlardan vazgeçmeden sonuca ulaşmaya çalışmak son derece önemlidir. Yapılan projelerde hatalar olabilir başarısızlıklar olabilir, burada önemli olan konu projede yapılan hatalardan dersler çıkarıp en kısa sürede bunu telafi edebilmektir. XP yazılım geliştirmede 12 farklı pratiği ele alır: planlama, kısa aralıklı sürümler, müşterinin de ekibe dahil olması, yeniden yapılandırma, test, ortak kod sahiplenme, basit tasarım, metafor, çiftli programlama, kodlama standardı, sürekli entegrasyon ve haftada 40 saat. Bunlardan bahsedecek olursak planlama oyunu genel olarak müşteriye planların aktarılmasıdır. Her bir yenilemede teslimatın ne zaman yapılacağı müşteriye aktarılarak müşteri ile sürekli iletişimde kalınmış olunur. Ekipte müşteri ise bir müşteri temsilcisinin sürekli ekipte bulunması durumudur. Bunun sonucunda yazılımda müşteriden edinilmesi gereken bilgilere en kısa sürede ulaşılır ve yazılımın hız kesmeden geliştirilmeye devam edilmesi sağlanmış olur. Test prensibinde ise kod yazılmadan önce programın test edilmesi sağlanır bu sayede sorunlar erken evrede belirlenmiş olur. Basit tasarım pratiğinde ise müşterinin gereksinimlerini belirledikten sonra müşterinin gereksinimlerinden ne eksik nede fazla olacak şekilde en basite indirgenmiş yazılımın oluşturmasına dikkat edilir. Diğer bir pratik olan çiftli programlanın kullanılmasının amacı ise program yazılırken ortaya çıkabilecek sorunları çözerken geçen süreyi en aza indirgemektir. Bu yüzden bir konuda 2 yazılımcı görevlendirilirse birinin göremediği sorunu diğerinin görmesi ihtimali daha yüksek olduğundan program daha hızlı ve daha hatasız olarak ortaya çıkmış olacaktır. Yazılım geliştirilirken getirilen yenilikler ve değişikliklerin sisteme hemen entegre edilmesi ve uyum sağlaması açısında sürekli entegrasyon pratiği kullanılmaktadır. Kısa aralıklı sürümler pratiğinin projeye sağladığı avantaj ise projenin hem yaşayan hem de gelişen bir proje olarak ele alınmasıdır. Bu sayede projenin gelişimi daha çok gözle görülebilir somut bir hal alacaktır. Yeniden yapılandırma pratiğinin uygulanmasındaki amaç ise kod ve tasarımın sürekli gözden geçirilerek getirilebilecek yenililerin projeye nasıl entegre edilebileceği konusunda fikir sahibi olunmasını sağlamaktır. Ortak kod sahiplenme prensibi ise ekip üyelerinin dayanışma içinde olması açısından önemlidir. Yazılan bir kod iyi de olsa kötü de olsa bu kodun ortak bir mal olduğunu sürekli olarak benimsemek ekip üyelerinin özverisini ve çalışma isteğini artıracaktır. Metafor pratiği ise gerçekleştirilen sistem parçalarının birbirlerine benzetilmesidir. Bu sayede hepsinin birbiriyle uyum içinde çalışması hedeflenmiştir. Kodlama standardı diğer bir önemli pratik olarak karşımıza çıkmaktadır. Eğer yazılan kodun bir standardı olmazsa ve herkes kendi tarzına göre kod yazarsa projedeki kodların anlaşılabilir olması beklenemez bu sorun bir kodlama standardı belirlenerek çözüme kavuşturulmuştur. Haftada 40 saat pratiği ise çalışma verimliliğini artırmaya yöneliktir.

SCRUM Jeff Sutjerland ve Ken Schawaber tarafından 1990ların ortalarında geliştirilen, çevik yazılım metodolojileriyle uygulanabilecek bir proje yönetim yaklaşımıdır. Burada proje yönetim yaklaşımı denilmesinin sebebi sadece yazılım için değil her proje için SCRUM’un kullanılabilir olmasını gösterir. Temel olarak karmaşık projeleri küçük parçalara bölerek geliştirmeyi amaçlar. SCRUM genellikle gereksinimleri çok fazla ve kaosla sonuçlanabilecek projelerde kullanılır. SCRUM’da üç temel kavram vardır: roller, toplantılar, bileşenler. Roller belirlenirken ürün sahibi, SCRUM yöneticisi ve SCRUM takımı olarak 3 e bölünerek roller dağıtılır. Toplantılar kısmı da 3 farklı alanda incelenebilir. Planlama toplantılarında geniş kapsamlı bir gereksinim listesi çıkarılır. Takımlar belirlenir. Risk değerlendirmeleri yapılır. Olası gereksinim değişiklikleri gözden geçirilir. Geliştirme araçları ve altyapı onaylanır. Dağıtım, geliştirme ve pazarlama maliyetleri hesaplanır. Gözden geçirme toplantılarında ise ilk olarak ürün sahibi ile takım, ürün gereksinim listesini gözden geçirir ve gereksinim elemanlarının hedeflerini ve içeriklerini belirler. Toplantının ikinci kısmında ise takım üyeleri ürün gereksinim listesinin en üstünden başlayarak koşu sonucunda gerçekleştirilecek gereksinimlerden oluşan koşu gereksinim listesini oluştururlar. Bu SCRUM içi en can alıcı noktadır. Gözden geçirme toplantılarında son olarak takım ürün sahibi tarafından belirtilen önceliği olan gereksinimlerden ne kadarını ve hangilerini yapacağını belirler. Her gün gerçekleşen günlük SCRUM toplantılarında ise takım üyeleri dün ne yaptım? Bugün ne yapacağım? Karşılaştığım sorun ve engeller neler? Gibi soruların cevaplarını verir.

Sonuç olarak geleneksel yazılım geliştirme metotlarından daha çok bu metotların kullanılmasının nedeni günümüzün karmaşık yazılım programlarına ayak uydurabiliyor olmalarıdır. Eskiden kullanılan yazılım programlarının basit makine dili ile kodlanmış basit programlar olması ve günümüz teknolojisine göre yazılım da geliştiği için çevik yazılım metotları ortaya çıkmıştır. Çevik yazılım metotlarında başlıca bahsettiğimiz XP ve SCRUM’u birbirinden ayıran bazı farklar vardır. XP sadece yazılım geliştirme metodu olarak kullanılırken SCRUM yazılım dışında bir proje için de bir geliştirme metodu olarak kullanılabilir. XP küçük ya da büyük, karmaşık ya da basit her yazılım türünde kullanılabileceği gibi SCRUM’un daha karmaşık proje yapılarında kullanılması daha mantıklı olacaktır.

Günümüzde XP den daha çok SCRUM un kullanılmasının nedenleri arasında özellikle son zamanlarda yazılımın ve teknolojinin çok hızlı gelişmesini sebep olarak ele alabiliriz. Bunun sonucunda ortaya çıkarılmak istenen projeler daha karmaşık yapılar haline gelince burada SCRUM un XP ye göre karmaşık yapılarda daha başarılı olduğu gerçeğini göz önünde bulundurmamız gerekir. Bu yüzden SCRUM günümüzde XP den daha çok kullanılıyor diyebiliriz

AHMET ENES ALBAK