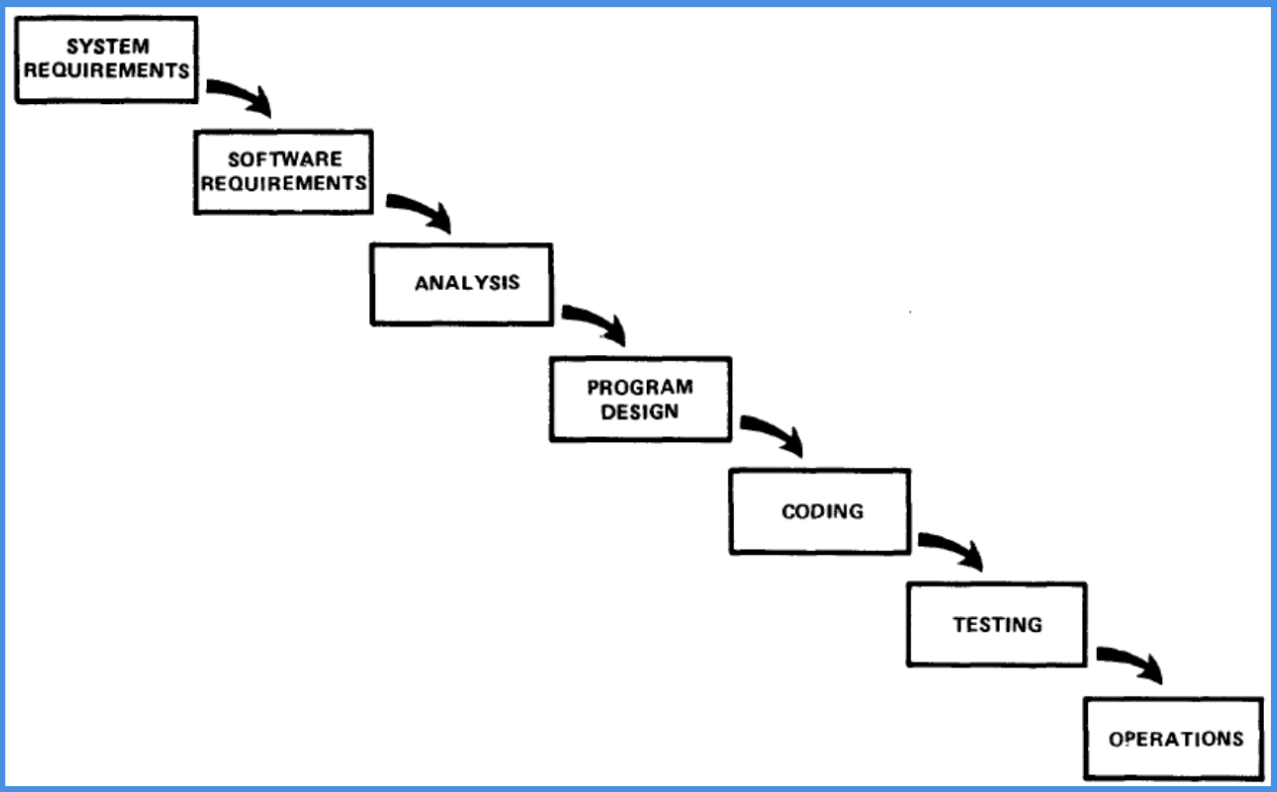
Büyük Yazılım Sistemlerinin Geliştirilmesini Yönetme



Waterfall

Winston Walker Royce

Çeviri: Cihan Yılmaz

Büyük Yazılım Sistemlerinin Geliştirilmesini Yönetme

Winston Walker Royce

Çeviren: Cihan Yılmaz

Önsöz

Büyük küçük birçok şirket Çevik Dönüşüm başlattı ya da başlatmayı düşünüyor. Hatta Steve Denning bu durum için "Why Agile Is Eating The World[1]" adlı bir makale yazdı ve bu makale oldukça popüler oldu. Çevik Dönüşüm, temelde geleneksel yaklaşımda işlemeyen ve üretken olmayan organizasyonu işler ve üretir duruma getirmektir. Geleneksel yaklaşım deyince aklıma gelen ilk yaklaşım Waterfall. Winston Royce tarafından adına **basit metot** (1970) denen daha sonra Bell ve Thayer tarafından adına **Waterfall** (1976) denen yaklaşımdan Çevik yaklaşımlara doğru bir evrim geçiriyoruz. Bu evrimi geçirirken nereden geldiğimizi hatırlamak ve nereye gidebileceğimize karar vermek bu makaleyi çevirmemdeki ana nedendi. Diğer bir neden Waterfall'un neden bu kadar popüler olduğunu anlayabilmekti. Bunlar:

- Basit
- Başlangıç maliyetinin düşük olması
- Öğretmesi ve öğrenmesi kolay (Üniversitede hala Waterfall öğretiliyor, makalenin yayınlanmasının üzerinden 49 yıl geçtiğini unutmayın)
- Zamanın üretim anlayışına paralel olması
- Mantıklı olması ☺
- Başlangıç ve bitişinin olmasının sağladığı yanılsama

Çok büyük çoğunluğumuz bu makaleden doğan yaklaşımlarla iş yaptı. İş yaptı derken hayatını kazandı ve hayatını yaşadı. Kimileri emekli bile oldu. Kimileri projeleri bitirdiğinde mutlu oldu bitiremediğinde üzüldü. Kimimiz bu yapıya yatkın olmadığı için iş değiştirdi. Peki, tüm bunlar olurken aslında başlangıç noktası neydi? Kaçımız bu başlangıç noktasını düşündü? Ne yazık ki çok azımız. Bugünlerde Çevik yaklaşımların çok popüler olduğunu yazımın başında söylemiştim. Yine ne yazık ki Çevik Bildiri'yi bir defa bile okumamış Çevik Koç, Scrum Kılavuzu'nu bir defa bile okumamış Scrum Master artıyor. Bu çeviri birazda birey ve toplum olarak Waterfall'a yaptığımızı Çevik yaklaşımlara yapmamak için yapıldı.

Makaledeki önemli noktalar: Makalede bence önemli olan yerlerin arka rengini sarı yaptım. Böylece buraların daha dikkatli okunması gerektiğini vurgulamak istedim.

Makalenin orijinaline erişebileceğiniz bağlantı: http://www-scf.usc.edu/~csci201/lectures/Lecture11/royce1970.pdf

Mart 2019,

Cihan Yılmaz

[1] İng. "Why Agile Is Eating The World", Türkçe "Çeviklik Neden Dünyayı Yiyor"

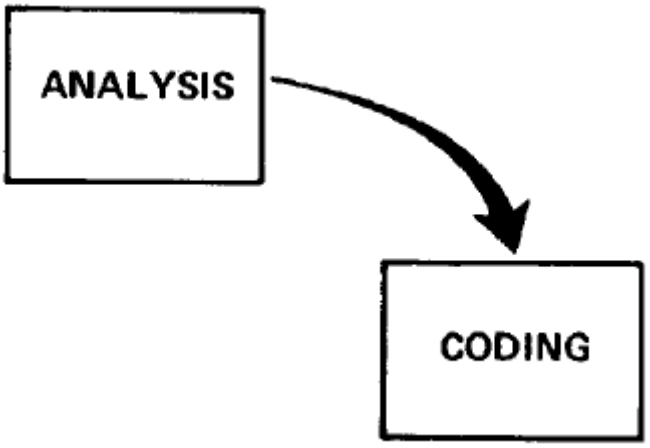
Büyük Yazılım Sistemlerinin Geliştirilmesini Yönetme

Giriş

Büyük yazılım geliştirme yönetimine dair kişisel görüşümü anlatacağım. Geçen dokuz yıl süresince birçok farklı görevim oldu, bu görevler çoğunlukla uzay aracı görev planlaması, komuta ve uçuş sonrası analizler için yazılım paketlerinin geliştirilmesiyle ilgiliydi. Bu görevlerde zamanında, bütçesinde operasyonel duruma gelme gibi farklı derecelerde başarıları deneyimledim. Deneyimlerimden kaynaklanan önyargılım oldu ve bu sunumda bazı önyargılarımı anlatacağım.

Bilgisayar Programı Geliştirme Fonksiyonları

Bilgisayar programı geliştirmelerinde büyüklükten ya da karmaşıklıktan bağımsız olarak, ortak iki temel adım vardır. İlk önce bir analiz adımı, bunu takiben Şekil 1'de gösterilen kodlama adımı vardır. Eğer efor az ve eğer son ürün onu geliştirenler tarafından işletilecekse böyle çok basit bir uygulama konsepti aslında ihtiyaç olan her şeyi verir –tıpkı iç kullanımda olan bilgisayar programlarında olduğu gibi. Bu aynı zamanda birçok müşterinin ödemekten mutlu olduğu bir geliştirme efor türüdür çünkü her iki adımda direkt olarak son ürünün kullanılabilirliğine katkıda bulunan gerçekten yaratıcı işleri içerir. Büyük yazılım sistemleri üretimi için bir uygulama planı sadece bu adımlara sıkıştırılırsa başarısızlık kaçınılmazdır. Ek birçok geliştirme adımına ihtiyaç vardır. Hiçbiri analiz ya da kodlama gibi son ürüne direkt olarak katkıda bulunmaz ve tümü geliştirme maliyetini yukarı çeker. Müşteri personeli genellikle bunlar için ödeme yapmamayı seçer ve geliştirme personeli bunları yapmamayı tercih eder. Yönetimin birincil fonksiyonu bu konseptleri iki gruba da satmaktır ve daha sonra geliştirme personelini bunlara uymaya zorlamaktır.

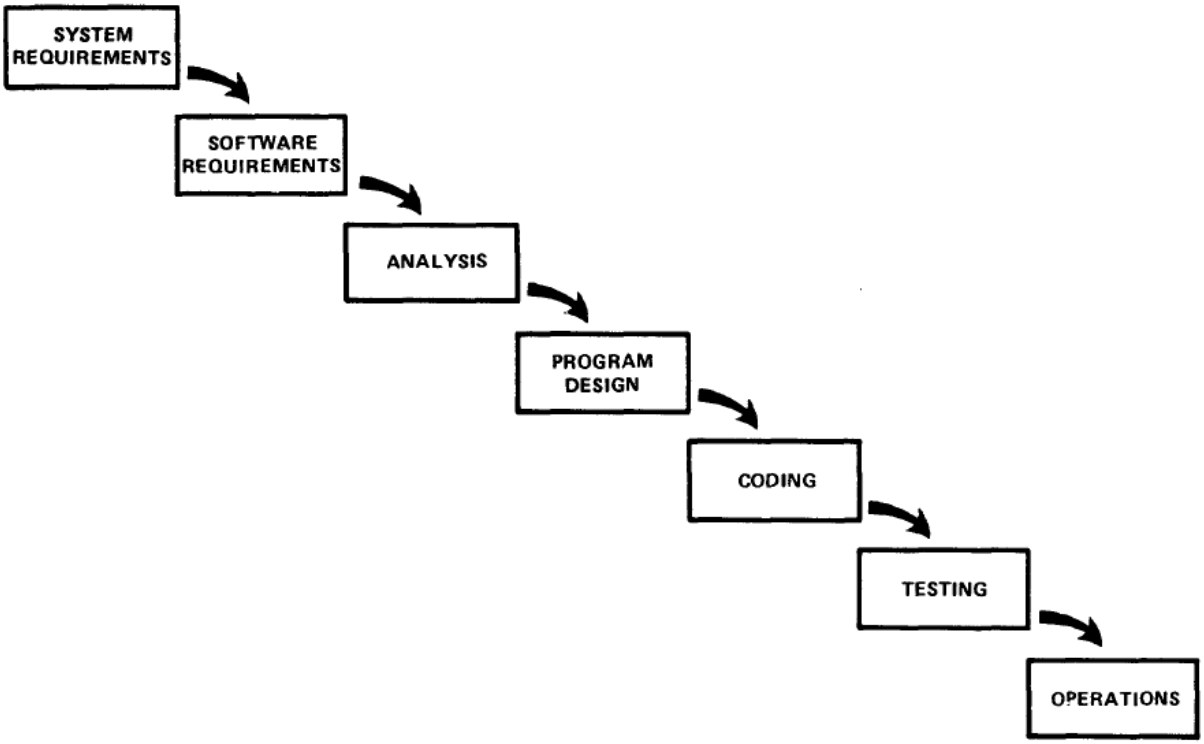


Şekil 1. İç işlemlerde kullanmak üzere küçük bir bilgisayar programı teslim etmek için uygulama adımları

Yazılım geliştirme için daha muazzam bir yaklaşım Şekil 2'de gösterilmiştir. Analiz ve kodlama adımları hala resimdedir fakat öncesinde iki aşamalı ihtiyaç analizi bulunur ve program tasarım adımıyla birbirlerinden ayrılmışlardır ve bunları takip eden bir test adımı bulunur. Bu eklemelere analiz ve kodlamadan farklı davranılır çünkü onların gerçekleştiriliş yolu oldukça farklıdır. Program

kaynaklarının en iyi şekilde kullanılması için bu eklemeler farklı şekilde planlanmalı ve bu eklemeleri yerine getirecek personel farklı olmalıdır.

Şekil 3, bu şema için başarılı geliştirme fazları arasındaki iteratif ilişkiyi gösterir. Adımların sıralanması, takip eden konsepti temel alır: her adım ilerledikçe tasarım daha çok detaylandırılır, önceki ve başarılı adımlar arasında döngü vardır fakat daha uzaktaki adımlarla nadir olarak bir sekans bulunur. Bunların hepsinden ötürü tasarım ilerledikçe değişim süreci yönetilebilir limitlere gelir. İhtiyaç analizi tamamlandıktan sonra tasarım sürecinde herhangi bir noktada sıkı ve yakın çekim, görülemeyen tasarım zorlukları için hareketli bir temel bulunur. Sahip olduğumuz şey, amacı erkenden yapılması gereken işlerin kapsamını maksimize etmek olan etkin bir geri çekilme pozisyonudur.



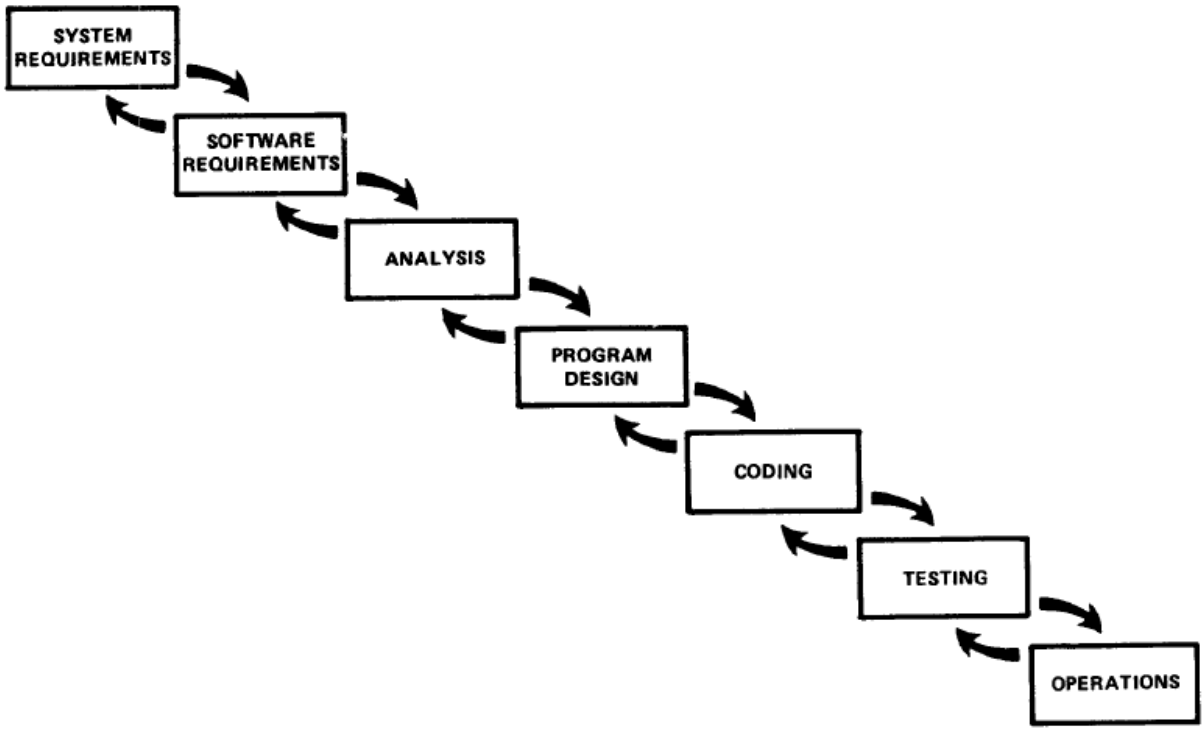
Şekil 2. Müşteriye teslim edilecek büyük bilgisayar programları geliştirmek için uygulama adımları.

Bu konsepte inanıyorum fakat yukarıda tanımlanmış uygulama riskli ve başarısızlığa davetiye çıkarıyor. Problem Şekil 4'te gösterilmiştir. Geliştirme döngüsünün sonunda gerçekleştirilen test fazı, analiz edilenden farklı olarak gerçekleşen zamanlama, depolama, giriş/çıkış transferler vb. için ilk etkinliktir. Bu fenomenler tam olarak analiz edilebilir değildir. Onlar, örneğin matematiksel fiziğin standart kısmi diferansiyel denklemlerine çözümler değildirler. Ancak bu fenomenler çeşitli dış kısıtlamaları karşılamada başarısız olursa daha sonra her zaman yeniden büyük bir tasarım gerekir. Basit bir yama ya da izole edilmiş kodun yeniden yazılması bu tür zorlukları çözmeyecektir. Gerekli tasarım değişiklikleri muhtemelen yıkıcı olabilir ki tasarımın temel aldığı ve her şeye mantıklı bir açıklama sağlayan yazılım ihtiyaçlarını bozar. Ya ihtiyaçlar değiştirilmelidir ya da tasarımda önemli

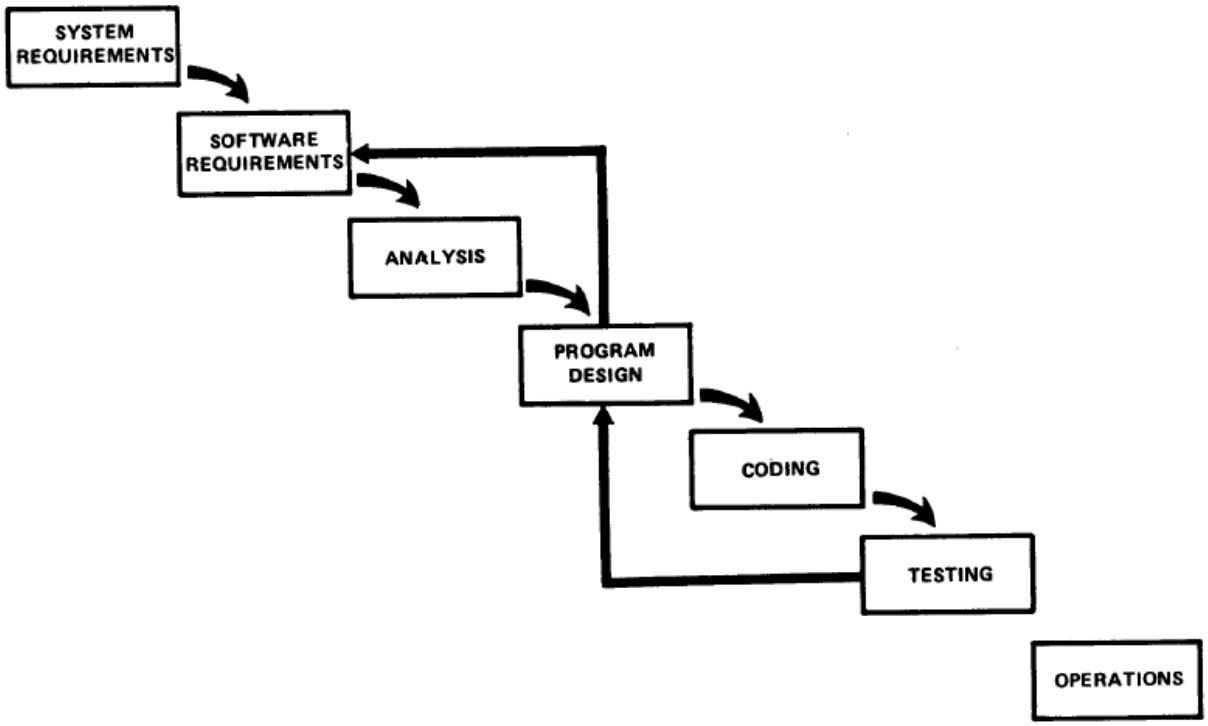
değişiklikler yapmak gereklidir. Geliştirme süreci başa döner ve zaman ve/veya maliyetin %100'ü aşacağı beklenebilir.

Analiz ve kod fazlarının atlandığına dikkat edebilirsiniz. Elbette bu adımlar olmadan yazılım üretemezsiniz fakat genel olarak bu fazlar nispi olarak yönetilmesi kolaydır ve ihtiyaçlar, tasarım ve test üzerindeki etkileri azdır. Deneyimlerime göre orbit mekaniklerinin, uzay aracı konum belirleme, taşıma kapasitesinin matematiksel iyileştirmeleri ve benzeri analizleriyle tümüyle tüketilen departmanlar vardır. Fakat bu departmanlar zor ve karmaşık işlerini tamamladıklarında sonuç olarak çıkan program adımları bir dizi aritmetik kod içerir. Eğer zor ve karmaşık işlerinin yerine getirilmesinde analistler hata yaparsa düzeltme her zaman diğer geliştirme temellerine yıkıcı bir geri dönüş olmadan, kod içinde küçük değişiklikler ile yapılabilir.

Gösterilen yaklaşımın temelde sağlam olduğuna inanıyorum. Bu makalenin geri kalanında geliştirme risklerini elimine etmek için bu temel yaklaşıma eklenmesi gereken ilave beş özelliği anlatacağım.



Şekil 3. Çeşitli fazlar arasındaki döngüsel etkileşim ardışık adımlarla sınırlıdır.



Şekil 4. Ne yazık ki gösterilen işlem için tasarım döngüleri hiçbir zaman ardışık adımlarla sınırlı değildir.

ADIM 1: PROGRAM TASARIMI ÖNCE GELİR

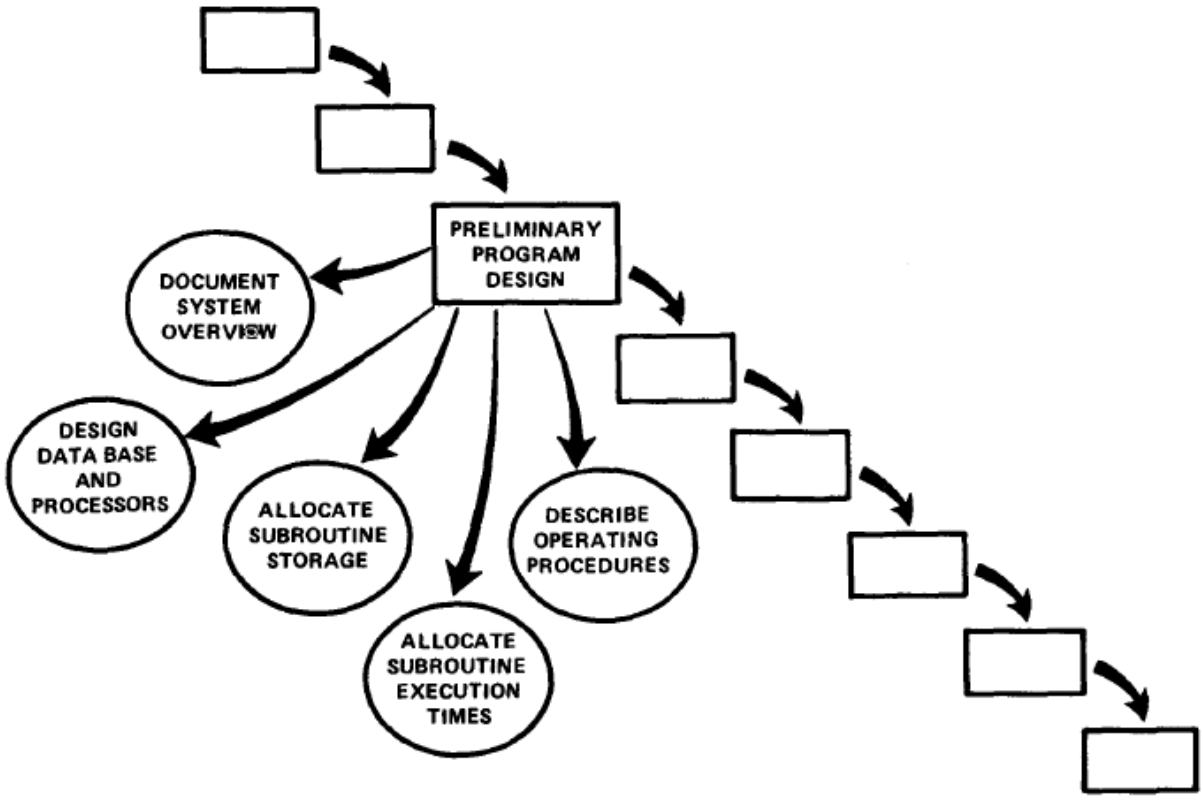
Düzeltme için ilk adım Şekil 1'de gösterilmiştir. Ön program tasarım fazı yazılım ihtiyaçlarının belirlenmesi ve analiz fazlarının arasına eklenmiştir. Bu prosedür program tasarımcısının mevcut herhangi bir analiz olmadan ilk yazılım ihtiyaçlarının nispi boşluğunda tasarım yapmaya zorlanması temelinde eleştirilebilir. Sonuç olarak tasarımcının ön tasarımı, analizin bitmesini bekleyip yapacağı tasarımla karşılaştırıldığında hatalı olabilir. Bu eleştiri doğru fakat önemli bir noktayı gözden kaçırıyor. Bu teknikle program tasarımcısı yazılımın, depolama, zamanlama ya da veri akışı nedenleriyle hatalı olmasının önüne geçebilir. Analiz ilerledikçe sonraki fazda program tasarımcısı, sonuçların farkında olacak şekilde depolama, zamanlama ve operasyonel kısıtlar konusunda analist üzerinde etkide bulunmalıdır. Program tasarımcısı haklı olarak kendi denklemini uygulamak için bu tip kaynaklara daha fazla ihtiyaç duyduğunda eş zamanlı olarak analist arkadaşlarından daha fazlası kapılmalıdır. Bu şekilde tüm analistler ve tüm program tasarımcıları, işlem zamanının ve depolama kaynaklarının uygun bir şekilde tahsis edilerek sonuçlandığı anlamlı bir tasarım sürecine katkıda bulunacaklardır. Eğer uygulanabilecek kaynakların toplamı yeterli değilse ya da embriyo işlem tasarımı hatalıysa bu erken aşamada yakalanacaktır ve ihtiyaçlar ve ön tasarım iterasyonu son tasarımdan, kodlamadan ve test başlamadan önce tekrar yapılabilir.

Bu süreç nasıl uygulanır? Aşağıdaki adımlar gereklidir.

1) **Tasarım sürecine program tasarımcılarıyla başla,** analistler ya da programcılarla değil.
2) **Veri işleme yöntemlerini tasarla, tanımla ve paylaştır,** hatalı olma pahasına. İşlem, fonksiyonları paylaştır, veri tabanını tasarla, veri tabanı işlemlerini tanımla, işlem zamanını

paylaştır, arayüzleri ve işletim sistemiyle süreç modlarını tanımla, giriş ve çıkış işlemini tanımla ve ön işlem süreçlerini tanımla.

3) Anlaşılabilir, bilgilendirici ve içinde bulunulan durumu anlatan **bir genel bakış dokümanı yaz.** Her çalışan sisteme dair temel bir anlayışa sahip olmalıdır. En azından genel bakış dokümanını yazabilecek bir kişinin sisteme dair derin bir anlayışa sahip olması gerekir.



Şekil 5. Adım 1: Analiz başlamadan önce ön program tasarımının tamamlandığından emin olun.

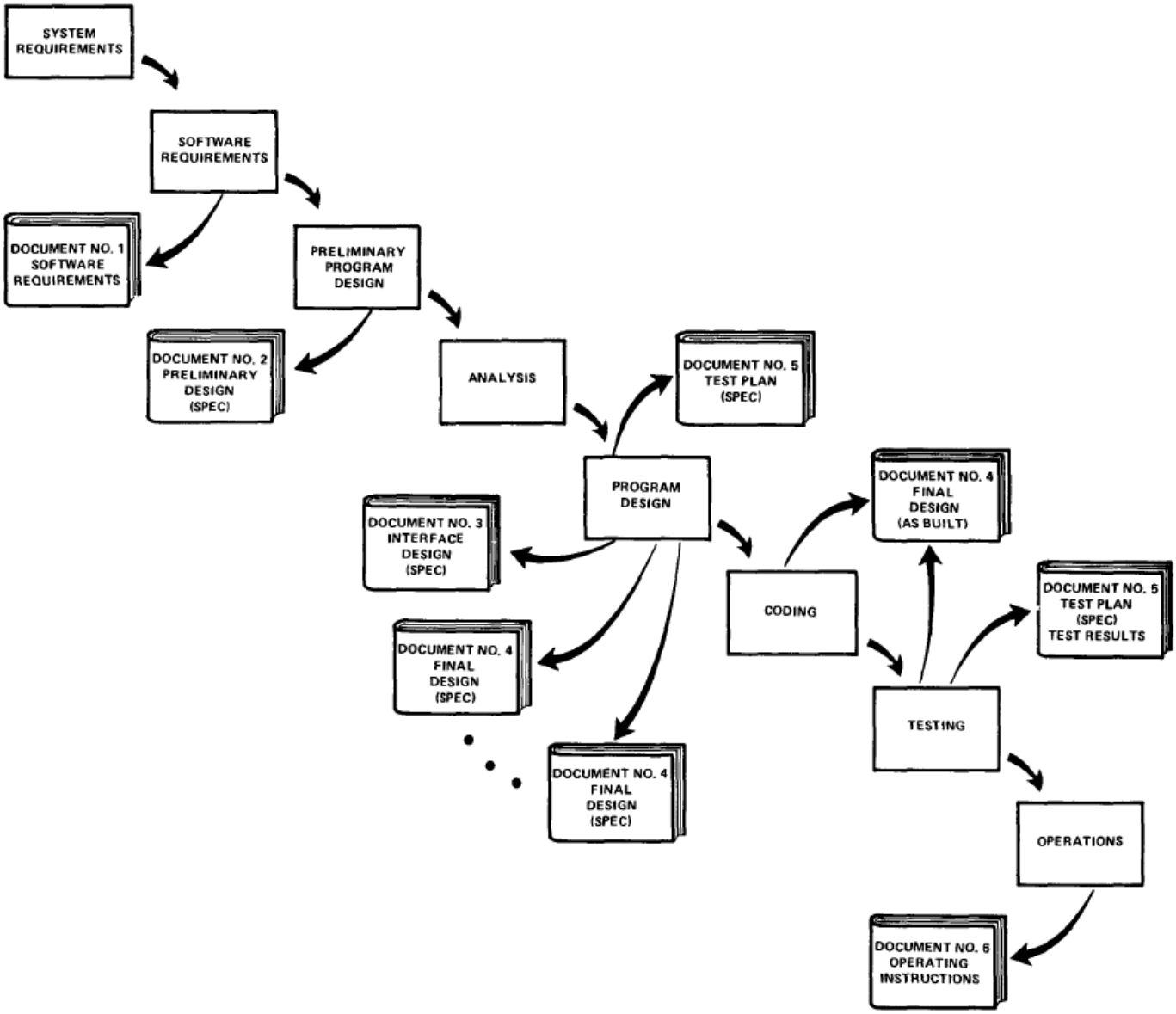
ADIM 2: TASARIMI DOKÜMANTE ETME

Bu noktada "ne kadar dokümantasyon?" konusunu ortaya çıkarmak uygun olur. Kendi görüşüm "oldukça fazla", bunu yapmaya gönüllü çoğu programcıdan, analistten ya da program tasarımcısından daha fazla. Yazılım geliştirme yönetiminin ilk kuralı dokümantasyon gereksinimlerinin acımasız bir şekilde uygulanmasıdır. Arada sırada diğer yazılım tasarımlarının ilerleyişini gözden geçirmek için çağrılırım. İlk adımım dokümantasyonun durumunu incelemektir. Eğer dokümantasyon ciddi bir şekilde ihmal edilmişse önerim basittir. Proje yönetimini değiştirin. Dokümantasyonla ilgili olmayan bütün aktiviteleri durdurun. Dokümantasyonu kabul edilebilir standartlara getirin. Yazılım yönetimi, yüksek derecede dokümantasyon olmadan en basit haliyle imkânsızdır. Örnek olarak karşılaştırmak için aşağıdaki tahmini vereyim. 5 milyon dolarlık bir donanım üretimini kontrol etmek için 30 sayfalık detaylı bir doküman beklerim. 5 milyon dolarlık bir yazılım üretiminde benzer seviyede bir kontrol elde etmek için 1500 sayfalık bir doküman beklerim.

Neden bu kadar çok dokümantasyon?

1) Her tasarımcı, arayüz tasarımcıları, yöneticisi ve muhtemelen müşteriyle iletişimde olmalıdır. Sözlü iletişim, arayüz ya da yönetim kararları için detaylı temel bilgiyi sağlama açısından çok soyuttur. Kabul edilebilir yazılı tanım, tasarımcıyı açık bir pozisyon almaya ve tasarıma dair somut kanıtlar sağlamaya zorlar. Bu, tasarımcının, aydan aya "yüzde 90 bitti" sendromunun ardına saklanmasını önler.
2) Yazılım geliştirmenin erken aşamalarında dokümantasyon şartnamedir ve tasarımdır. Kodlama başlayana dek bu üç isim (dokümantasyon, şartname ve tasarım) bir tek şeyi ifade eder. Eğer dokümantasyon kötüyse tasarım kötüdür. Eğer dokümantasyon yoksa tasarım yoktur, sadece insanlar tasarımı düşünüyor ve tasarım hakkında konuşuyordur, bunun değeri vardır ama çok değildir.
3) İyi dokümantasyonun gerçek değeri geliştirme sürecinde sonraki adımlarda (test fazında başlar, operasyon ve yeniden tasarımda devam eder) başlar. Dokümantasyonunun değeri her program yöneticisinin karşılaştığı üç somut açıdan, durumdan tanımlanabilir.
    a. **Test fazı boyunca,** iyi dokümantasyon ile yönetici, personelin program içindeki hatalarına odaklanabilir. İyi bir dokümantasyon olmadan her hata, büyük ya da küçük, bir adam tarafından analiz edilmiştir ki muhtemelen ilk etapta hatayı yapan kişidir çünkü o, ilgili programı anlayan tek kişidir.
    b. **Operasyon fazı boyunca,** iyi dokümantasyon ile yönetici operasyon odaklı personeli programı işletmesi ve daha iyi ve ucuz bir iş yapması için kullanabilir. İyi bir dokümantasyon olmadan yazılım, onu geliştiren kişiler tarafından işletilmelidir. Genel olarak bu kişiler yazılım işletmeye ilgilenmez ve operasyon odaklı personeller kadar verimli bir iş yapmazlar. Bu bağlamda belirtilmelidir ki operasyonel bir durumda eğer bir erteleme varsa her zaman ilk yazılım suçlanır. Yazılımı aklamak ya da suçu düzeltmek için yazılım dokümantasyonu açık bir şekilde konuşmalıdır.
    c. **Başlangıç operasyonlarını takiben,** sistem iyileştirmeleri düzenli olduğunda iyi dokümantasyon etkin yeniden tasarıma, güncellemeye ve alanda yeniden uyarlamaya izin verir. Eğer dokümantasyon yoksa genellikle yazılım işletmenin var olan tüm çerçevesi çöptür.

Şekil 6, daha önce gösterilen adımlara bağlanan bir dokümantasyon planı gösterir. Dikkat edin altı doküman oluşturulmuştur ve son ürünün teslim zamanında Doküman No 1, No 3, No 4, No 5 ve No 6 güncel ve geçerli olmalıdır.

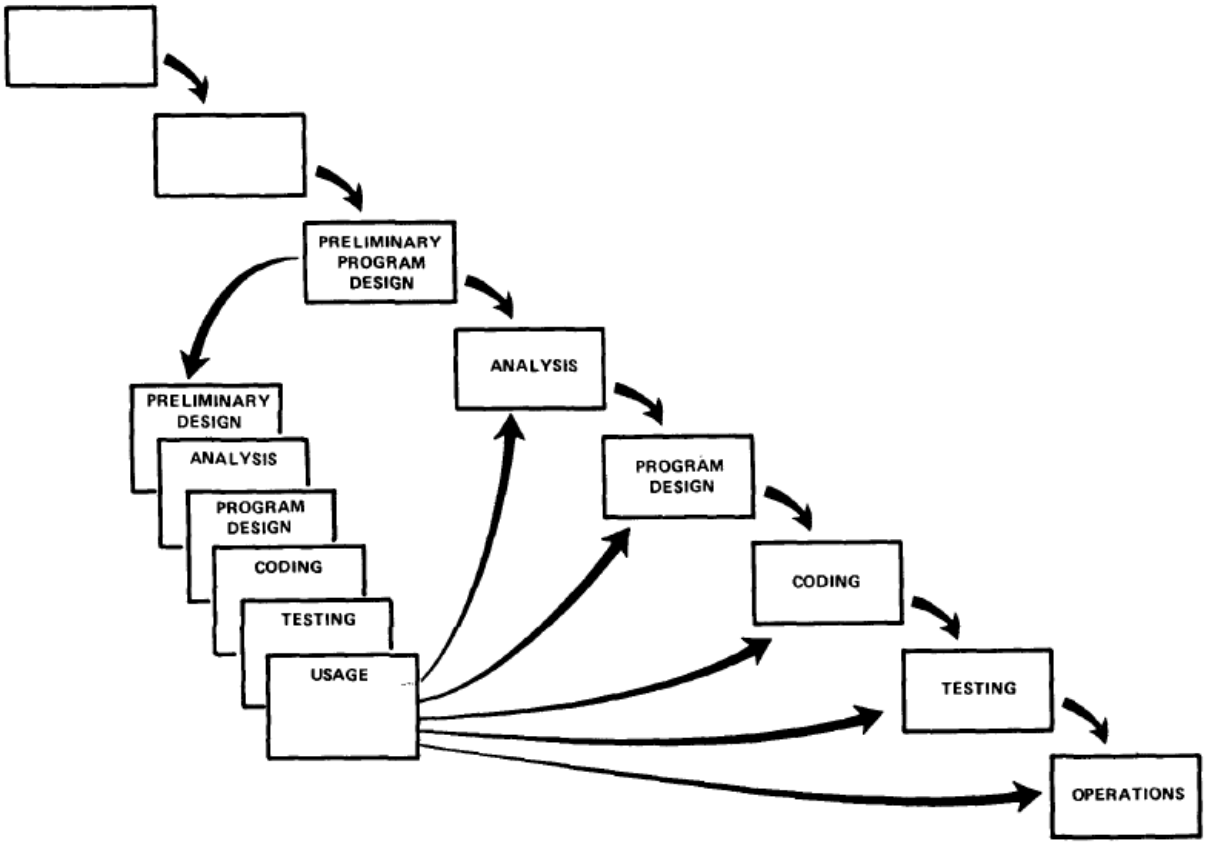


Şekil 6. Adım 2: Dokümantasyonun geçerli ve tam olduğundan emin olun – en az altı farklı doküman gereklidir.

ADIM 3: İKİ KERE YAPIN

Dokümantasyondan sonra başarı için en önemli ikinci kriter ürünün tamamıyla orijinal olup olmadığına bağlıdır. Eğer bilgisayar programı ilk defa geliştiriliyorsa müşteriye sonunda teslim edilen versiyonun gerçekte kritik tasarım/operasyon alanlarının düşünüldüğü ikinci versiyon olmasını ayarlayın. Şekil 7, bir simülasyon gibi bunun nasıl yapılabileceğini gösterir. Dikkat edin bu basitçe tüm sürecin bir minyatürüdür, bir zaman ölçeğinde tüm eforla karşılaştırıldığında görece küçüktür. Doğası gereği bu efor tüm zaman ölçeğine bağlı olarak ve modellenen kritik problem alanlarının doğası gereği oldukça değişkendir. Eğer efor 30 ayı geçerse o zaman erken geliştirilen bu pilot model 10 ay olarak belirlenebilir. Bu zamanlama için uygun resmi kontroller, dokümantasyon süreçleri, vb. kullanılabilir. Ancak tüm efor 12 aya düşürülürse daha sonra pilot eforu da ana geliştirme hattında yeterli avantaj kazanabilmek için 3 ay ile sınırlandırılabilir. Bu durumda dâhil olan personel tarafında çok özel geniş bir yetkinlik gereklidir. Dâhil olan personeller analiz, kodlama ve program tasarımı konusunda güçlü yetkinliklere sahip olmalıdır. Personeller tasarımdaki hataları hızlıca belirlemeli,

hataları ve alternatiflerini modellemeli, bu erken aşamada çalışmaya değmeyecek tasarımın basit açılarını unutmalı ve sonunda hatasız bir program geliştirmelidirler. İki durumda da simülasyon ile beraber bunların hepsinin amacı zamanlama, depolama, vb., sorulardır ki bu yapılmazsa bahsedilen konular sağduyuya kalır, yapılırsa doğruya yakın bir şekilde çalışılabilir. Simülasyon olmadan proje yöneticisi insan hükmünün merhametine kalmıştır. Simülasyon ile proje yöneticisi en azından bazı önemli hipotezlerin deneysel testlerini yapabilir ve insan hükmüne kalan yerlerin (kalkış brüt ağırlığı, tamamlanması için maliyet ya da günlük duble) kapsamını daraltabilir ki bu bilgisayar program tasarımında her zaman ve ciddi bir şekilde iyimserdir.



Şekil 7. Adım 3: İşi iki defa yapma girişimi – İlk sonuçlar son ürünün erken bir simülasyonunu sağlar.

ADIM 4: TESTİ PLANLA, KONTROL ET VE İZLE

Hiç kuşkusuz proje kaynaklarının en büyük kullanıcısı, ne adam gücü, ne bilgisayar zamanı ne de yönetim kararıdır, bu test aşamasıdır. Test fazı, dolar(para birimi) ve zaman açısından en büyük riskin fazıdır. Alternatiflerin en az olduğu ya da olmadığı zamanlamadaki en son aşamada meydana gelir.

Önceki üç öneri, analiz ve kodlama başlamadan önce programı tasarlamak, dokümantasyonu tamamlamak ve pilot model oluşturmak, hepsinin amacı problemler test fazına girmeden ortaya çıkarmak ve çözmektir. Ancak bunların hepsini yaptıktan sonra hala bir test fazı vardır ve hala yapılması gereken önemli şeyler vardır. Şekil 8, test yapmanın bazı önemli açılarını listeler. Test planlaması için aşağıdakilerin dikkate alınmasını tavsiye ederim.

1) Test sürecinin birçok parçası en iyi şekilde orijinal tasarıma katkıda bulunması gerekli olmayan test uzmanları tarafından gerçekleştirilir. Eğer testi sadece tasarımcının yapabileceği tartışılıyorsa bunun nedeni olarak da geliştirdiği alanı sadece onun anladığı söyleniyorsa, bu dokümantasyonun başarısız olduğuna kesin bir işarettir. İyi bir dokümantasyon ile kendi görüşüme göre tasarımcıdan daha iyi bir iş çıkaracak yazılım ürün güvencesi uzmanları kullanmak uygundur.
2) Hataların büyük çoğunluğu doğaları gereği açıktır ve görsel bir incelemeyle kolayca belirlenebilirler. Analizin her bir parçası ve kodlamanın her bir parçası orijinal analizi yapmayan ya da orijinal kodu yazmayan fakat analiz ve kod düzeltmenin doğasında olan kayıp eksi işareti, çarpan, yanlış adrese giden kod parçaları vb. şeylerin yerini saptayan biri tarafından görsel taramaya tabi tutulmalıdır. Bu tarz şeyleri tespit etmek için bilgisayar kullanmayın – bu çok pahalıdır.
3) Bir tür sayısal kontrol kullanarak bilgisayar programında bulunan her mantıksal yolu en az bir defa test edin. Eğer ben müşteri olsaydım ta ki bu süreç tamamlanana ve onaylanana kadar teslimi kabul etmezdim. Bu adım kodlama hatalarının büyük çoğunluğunu ortaya çıkarır.

   Bu test süreci kulağa basit gelirken, büyük ve karmaşık bilgisayar programlarında her mantıksal yolu kontrollü giriş değerleriyle test etmek göreceli olarak zordur. Hatta bunun imkânsıza yakın olduğunu iddia edenler olacaktır. Buna rağmen her mantıksal yolun en azından bir otantik kontrole tabi tutulması önerimde ısrar edeceğim.

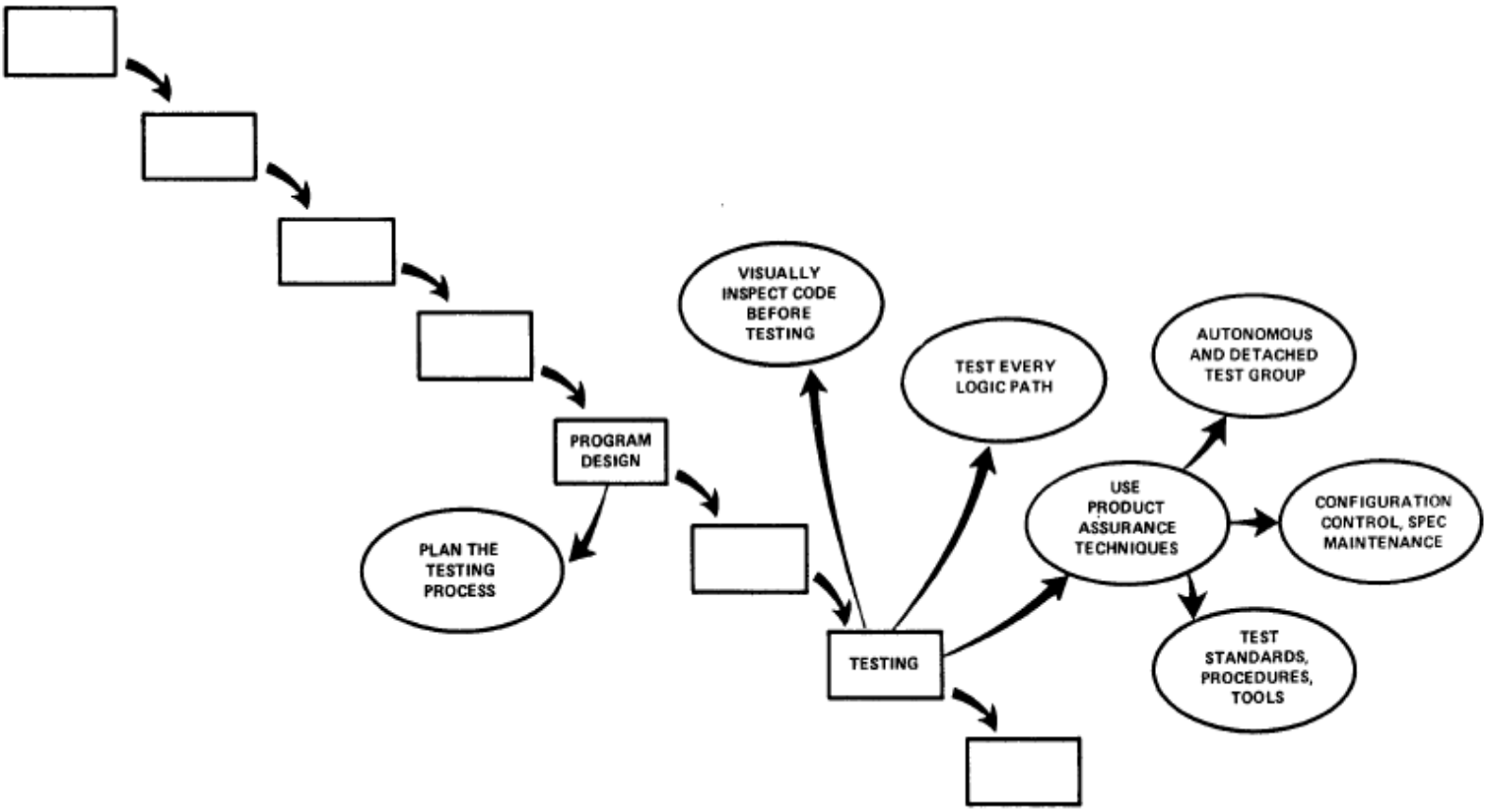
4) Kolay hatalar (ki bunlar çoğunluğu teşkil eder ve büyük hataların görünmesini engeller) çözüldükten sonra yazılımı çıkış hedefleri için teste alma zamanıdır. Geliştirme süresi boyunca uygun bir zamanda ve uygun kişinin ellerinde çıkış için en iyi araç bilgisayarın kendisidir. Önemli yönetim kararları: Ne zaman ve son çıkışı kim yapacak?

ADIM 5: MÜŞTERİYİ DÂHİL ETME

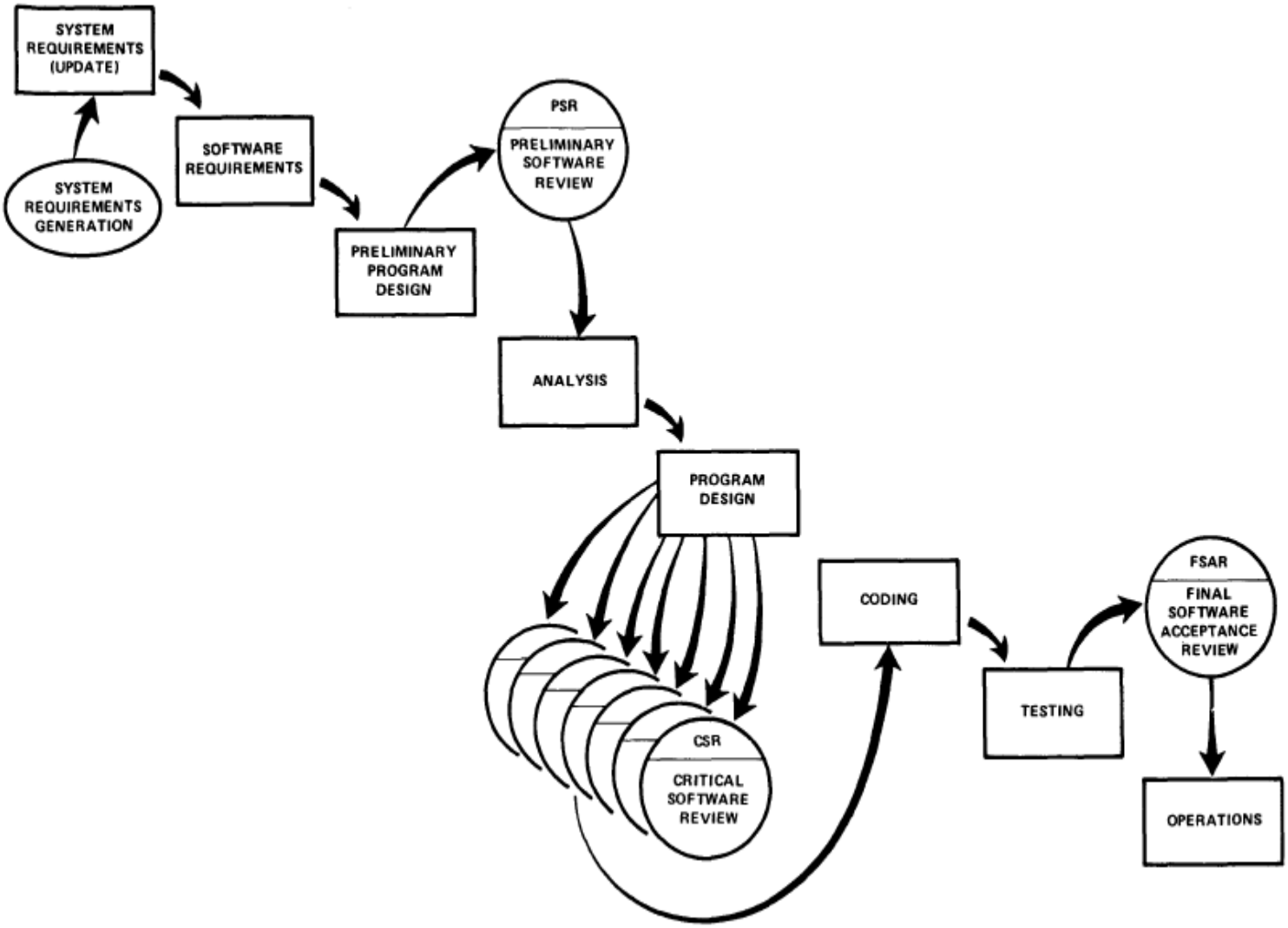
Bazı nedenlerden dolayı yazılım tasarımının ne yapacağı önceki anlaşmaya rağmen çok daha geniş bir yorumun konusudur. Müşteriyi resmi bir şekilde dâhil etmek önemlidir böylece müşteri son teslimden önce taahhüdünü gerçekleştirmeye çalışır. Sözleşmeyle çalışan kişiyi ihtiyaçların belirlenmesi ve operasyon arasında boş bırakmak sorunlara davetiye çıkarır. Şekil 9, ihtiyaç tanımları için üç noktayı, müşterinin iç görüsü, yargısı ve taahhüdünün geliştirme eforunu desteklediğini gösterir.

ÖZET

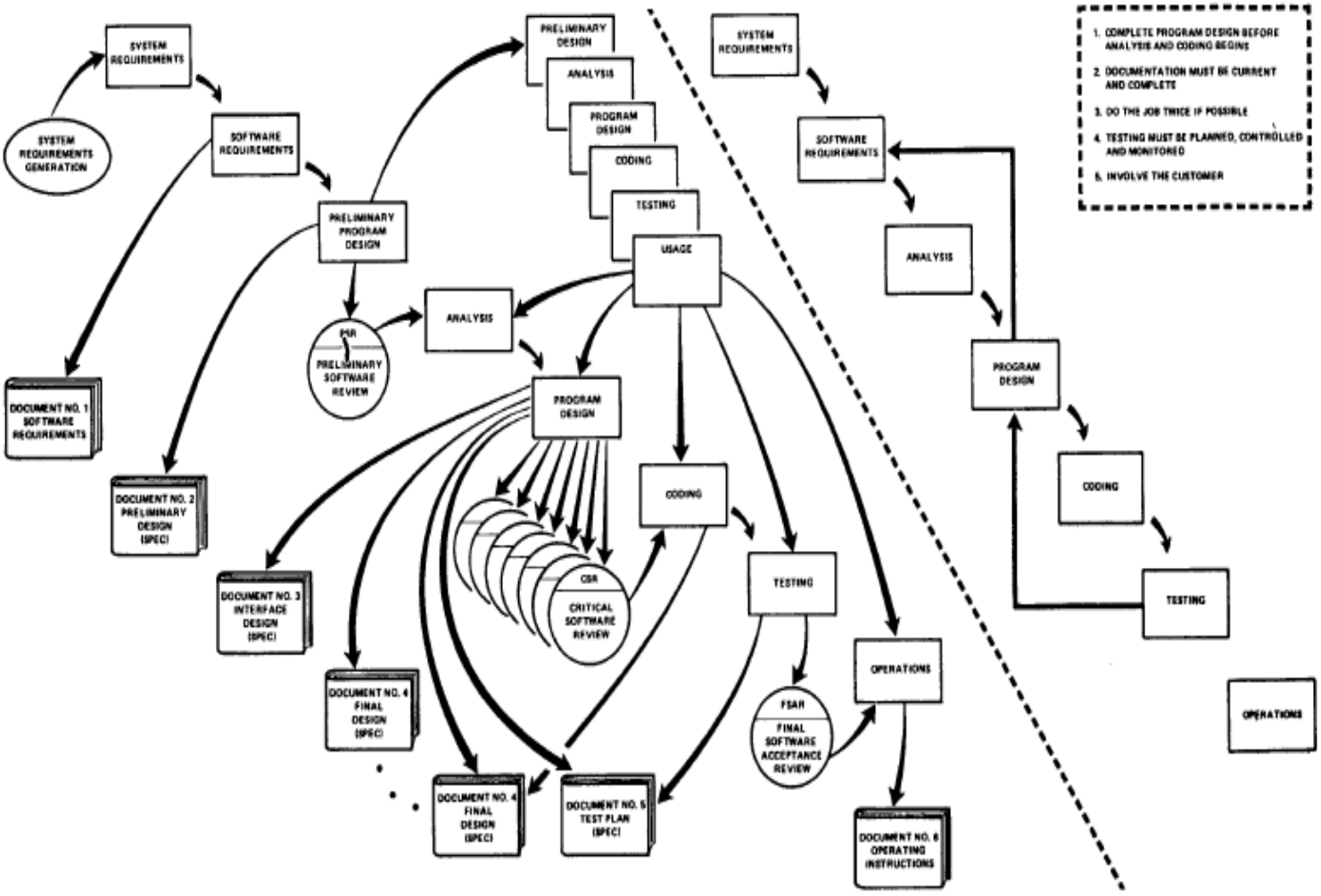
Şekil 10, riskli bir geliştirme sürecini talep edilen ürünün teslim edilebileceği bir sürece dönüştürmek için gerekli olduğunu hissettiğim beş adımı özetler. Her bir maddenin fazladan bir maliyeti olduğunu vurgulamak isterim. Burada belirtilen beş karmaşıklık olmadan görece daha basit bir süreç başarılı bir şekilde işliyorsa o zaman tabi ki ek maliyet doğru bir harcama olmaz. Deneyimlerime göre basit metot büyük yazılım geliştirme süreçlerinde asla işe yaramadı ve bunu iyileştirmek için yapılan harcamalar, yukarıda tanımladığım beş adımlı süreç için gerekli olan maliyeti çok çok daha fazla aştı.



Şekil 8. Adım 4: Bilgisayar programı testini planla, kontrol et ve izle.



Şekil 9. Adım 5: Müşteriyi dâhil edin. Dâhil olma resmi, derinlemesine ve sürekli olmalıdır,



Şekil 10. Özet.