Gümüş Kurşun Yok: Yazılım Mühendisliği'nin Asli ve Arızi Özellikleri

Frederick P. Brooks, Jr.

*(Bu yazı, Frederick Brooks'un "**No Silver Bullet: Essence and Accidents of Software Engineering**" başlıklı makalenin bir kısmının Türkçe çevirisidir. Vakit buldukça hepsini çevireceğim. Bu belgeye http://www.javaturk.org/store/swe/No%20Silver%20Bullet%20-%20TR.pdf adresinden ulaşabilirsiniz. Öneri ve eleştiriler için akin@javaturk.org adresine ulaşabilirsiniz.)*

Halk kültürümüzün kabuslarında ortaya çıkan canavarların en çok korkutanı kurt adamlardır, çünkü onlar, tanıdık bir sima iken aniden dehşet verici bir hale dönüşürler. Bu yaratıklar için insanlar da onları yere serecek gümüşten yapılmış kurşun ararlar.

Benzer şekilde yazılım projeleri de en azından teknik olmayan yöneticiler için aynı karakteri taşırlar, ilk başta genelde masum görünüşlüdürler ve herşey yolunda gibidir ama tutturulamamış planlara, aşılmış bütçelere ve hatalı ürünlere sahip bir canavara dönüşeme yeteneğine sahiptirler. Bu yüzden yazılım projeleri için de, aynı donanım maliyetlerindeki düşüş hızında bir maliyet düşüşü sağlayacak, gümüş bir kurşun arayan, umutsuz çığlıkları duyar dururuz.

Fakat önümüzdeki 10 yıllık ufka baktığımızda herhangi bir gümüş kurşun görünmemektedir. İster teknolojide isterse yönetim tekniklerinde, üretkenlik, güvenirlik ve basitlikte, 10 katlık bile bir ilerleme sağlayacak bir gelişmenin olması mümkün değildir. Bu makalede, yazılım problemlerinin tabiatını ve teklif edilen kurşunların özelliklerini inceleyerek bunun neden böyle olduğunu göstemeye çalışacağım.

Öte taraftan, şüphecilik karamsarlık değildir. Her ne kadar heyecan verici buluşlar olmasa ve böyle bir şeyin zaten yazılımın doğasıyla çeliştiğini düşünsem de pek çok cesaret verici yenilikler olmaktadır. Bu yenilikleri geliştirmek, yaymak ve kullanmak konusunda dispilinli ve kararlı olmak, 10 katlık bir gelişmeyi sağlamalıdır. Bu konuda *müthiş bir yol* olmasa da en azından *yol* vardır.

Hastalığı ortadan kaldırmaya ilk adım, şeytanlarlarla ilgili teorileri bir kenara bırakıp mikrop teorisine dönmektir. Yani ilk adım, umudun başı olarak, sihirli çözümlerden medet ummayı bırakmaktır. Bu konuda söylenebilecek şey, ilerlemenin büyük gayretlerle, ancak adım adım olacağı

ve temizliğe, kararlı ve aralıksız bir özenin verilmesinin gerektiğidir. Yazılım mühendisliği dünyasında bugün olan biten de budur.

Zor Olmak Zorunda mı? Asli Zorluklar

Gümüş bir kurşunun olmaması sadece şu an görünürde olan birşey değildir, yazılımın doğası onun gelecekte de olmasını pek de ihtimal dahilinde kılmamaktadır. Elektroniğin, tranzistörlerin ve büyük çaplı entegre devrelerin bilgisayar donanımı için yaptığını hiçbir buluş yazılımın üretkenliği, güvenirliği ve basitliği için yapmayacaktır. Hiç bir zaman, her iki yılda bir, iki katına çıkan kazanımlar göremeyeceğiz.

Birincisi, şu teslim edilmelidir ki anormallik yazılımdaki gelişmenin yavaşlığı değil, bilgisayar donanımındaki ilerlemenin çok hızlı oluşundadır. Medeniyetin başlangıcından bu yana hiç bir teknoloji 30 yılda 1 milyon katında bir performans kazanımı elde edemedi. Hiç bir başka teknolojide ileri performans ile düşük maliyet arasında bir seçimde bulunulamaz. Bütün bu kazanımlar, bilgisayar üretiminin montaj endüstrisi olmaktan çıkıp bir süreç endüstrisine dönüşmesinin sonucudur.

İkinci olarak yazılım teknolojisinde ne hızda bir ilerleme umabileceğimizi görmek için bu teknolojilerin zorluklarını inceleyelim. Aristo'ya uyarak bu zorlukları asli yani yazılımın doğasında var olanlar ile arızi yani bugün yazılımın üretiminde olan ama aslında doğasında olmayanlar olmak üzere ikiye ayırıyorum.

Bir yazılım varlığının aslı, içiçe geçmiş kavramlardan oluşmuş bir yapı olmasıdır: Veri kümeleri, veriler arasındaki ilişkiler, algoritmalar ve fonksiyonların çağrıları. Bu asıl, böyle kavramsal bir yapının, pek çok farklı görünümün altındaki aynı yapı olmasından dolayı, soyuttur. Yine de böyle bir yapı yüksek bir kesinlikte ve zengin bir detaydadır.

İnancım şudur ki yazılım inşa etmenin zor tarafı böyle kavramsal bir yapıyı, tarif etmek, tasarlamak ve test etmektir, onu sunma/oluşturma eforu ya da sunumun/oluşturulanın olması gerekene sadakatini test etmek değildir. Hala pek çok sözdizim hatası yapıyoruz ama emin olun ki onlar pek çok sistemdeki kavramsal hatalarla kıyaslanamayacak kadar basit şeylerdir.

Eğer bu doğruysa, yazılım inşa etmek her zaman zor olacaktır. Tabiati itibariyle de gümüş bir kurşun hiçbir zaman olmayacaktır.

Şimdi de modern yazılım sistemlerinin bu indirgenemez aslının doğal özelliklerine bir bakalım: Karmaşıklık, uyumluluk, değişebilirlik ve görünmezlik.

Karmaşıklık: Yazılım varlıkları, belki de büyüklük itibarıyla diğer insan yapılarından daha karmaşıktırlar çünkü hiçbir parçası birbirinin benzeri değildir (en azından dilin ifadelerinin üzerindeki yapılarda). Zaten benzer olsalar, bu iki benzer parçayı – açık ya da kapalı – tek bir yordam olarak

birleştirirdik. Bu açıdan bakıldığında yazılım sistemleri, tekrarlı parçaların bir arada olduğu bilgisayarlar, binalar ya da otomobillerden, temelden ayrılır.

Dijital bilgisayarlar zaten insanların bina ettiği pek çok şeyden daha karmaşıktır: Örneğin çok fazla sayıda durumları vardır. Bu onları anlamayı, anlatmayı ve test etmeyi zaten zor kılar. Yazılım sistemleri ise bilgisayarlardan onlarca kat daha fazla duruma sahiptirler.

Benzer şekilde bir yazılım varlığını büyütmek sadece aynı elemanların daha büyük ebattakilerini tekrarlamak değildir, farklı elemanlarla sayıca büyümek demektir. Pek çok durumda elemanlar birbirleriyle doğrusal olmayan bir şekilde iletişimde bulunurlar ve bütünün karmaşıklığı, doğrusal iletişimde oldukları duruma göre çok daha fazla artar.

Yazılımın karmaşıklığı asli bir özelliktir, arizi değildir. Bu yüzden bir yazılım varlığının karmaşıklığını ortadan kaldıran bir tanım, çoğu zaman onun aslını da ortadan kaldırır. Üç yüz senedir matematik ve fizik bilimleri, modellerden özellikler elde edip, bu özellikleri deneyle test ederek, karmaşık olayların basitleştirilmiş modellerini yapıp ciddi ilerlemeler elde ettiler. Bu paradigma başarılı oldu çünkü modellerde önem verilmeyen karmaşıklıklar, ilgili olayın asli özelliklerinden değildi. Karmaşıklıklar asli olduğunda başarılı olamazdı.

Yazılım ürünleri geliştirmedeki klasik sorunların pek çoğu, bu asli karmaşıklıktan ve ebata göre doğrusal olmayan büyümeden kaynaklanmaktadır. Karmaşıklıktan takım üyeleri arasındaki iletişimin zorluğu ortaya çıkmaktadır ki bu da ürün hatalarına, aşılmış bütçelere ve gecikmiş planlara yol açmaktadır. Karmaşıklıktan, programın, olması muhtemel bütün durumlarını ortaya koymanın zorluğu, dolayısıyla da bunları daha az seviyede bir anlama ortaya çıkmaktadır ki bu da güvenirliği ortadan kaldırmaktadır. Fonksiyondaki karmaşıklık, fonksiyonu çağırmada zorluğa yol açar ki bu da programları kullanmayı zorlaştırır. Yapıların karmaşıklığından, yan etki oluşturmaksızın programları yeni fonksiyonlara sahip olacak şekilde genişletmenin zorluğu çıkar. Yapıların karmaşıklığından, gözde canlandırılamayan durumlar oluşur ki bunlar da güvenlik açıklarına sebep olur.

Karmaşıklıktan sadece teknik sorunlar çıkmaz, yönetim sorunları da baş gösterir. Karmaşıklık genel bakışı zor hale getirir ki bu durum da kavramsal tutarlılığa engel olur. Karmaşıklık bütün sıkıntılı noktaların bulunup kontrol altına alınmasını zorlaştırır. Karmaşıklık, öğrenme ve anlama konusunda o kadar büyük bir yük oluşturur ki çalışanların işten ayrılmaları bir yıkım haline gelir.

Uyumluluk: Karmaşıklıkla karşılaşma noktasında yazılımcılar yalnız değillerdir. Fizikçiler “temel” parçacık seviyesindeki son derece karmaşık nesnelerle uğraşırlar. Fakat bir fizikçi, kuarklarda olsun, birleşik alan teorilerinde olsun hepsi için genel geçer prensiplerin bulunacagına dair sağlam bir inançla çalışır. Einstein, doğanın basitleştirilmiş açıklamalarının olması gerektiğini çünkü Tanrı’nın kaprisli ya da rastgele davranmayacağını ileri sürmüştü.

Böyle bir inanç, bir yazılım mühendisini kesinlikle rahatlatmaz. Çünkü üstesinden gelmek zorunda olduğu karmaşıklığın büyük çoğunluğu rastgele karmaşıklıktır ki pek çok insani kurum ya da sistem tarafından mantıksızca oluşturulmuştur ve yazılım mühendisinin arayüzleri bu yapılarla uyum içinde çalışmalıdır. Yazılım mühendisinin geliştirdiği bu arayüzler, arayüzden arayüze, zamandan zamana değişir ama bu durumun sebebi ihtiyaçlar değildir, sadece Tanrı yerine farklı kişiler tarafından tasarlanmış olmalarıdır.

Pek çok durumda software uyumlu olmalıdır çünkü sahneye en son gelen odur. Diğer pek çok durumda da, yazılımın uyum yeteneğinin yüksek olduğu düşünüldüğü için uyumlu olmak zorundadır. Fakat bütün bu durumlarda karmaşıklığın büyük çoğunluğu, diğer arayüzlere uyumdan kaynaklanmaktadır ki böyle bir karmaşıklık sadece yazılımın tekrardan tasarlanmasıyla basitleştirilemez.

Değişebilirlik: Yazılım varlıkları sürekli olarak değişim baskısıyla karşı karşıyadırlar. Tabi ki binalar, arabalar ve bilgisayarlar da öyledir. Fakat imal edilen şeyler imalattan sonra çok seyrek değişirler; ya daha sonraki modellerle yer değiştirirler, ya da asli özellikleri, aynı temel tasarımla üretilen sonraki kopyalarının içine gömülür. Otomobillerin geri çağrılması son derece nadirdir; piyasaya yayılımış bilgisayarların değişmesi de pek sık olmaz. Bu ikisi durum da, piyasaya yayılımış yazılımların değişmesinden çok daha nadiren olur.

Bu durum bir yönüyle, bir sistemin yazılımının, o sistemin fonksiyonlarını meydana getirmesi ve fonksiyonların da değişimin baskısını en fazla hisseden taraf olmasındadır. Bu durum diğer bir yönüyle de yazılımın daha kolay değiştirilebilmesindendir, çünkü yazılım tamamen zihinseldir ve sınırsızca değiştirilebilir. Gerçekte binalar da değişir, ama değişimin yüksek maliyeti ki herkes tarafından bilinir değiştireceklerin isteklerini kursaklarında bırakır.

Bütün başarılı yazılımlar değişir. Burada iki süreç vardır. İlki, yazılım ürünü faydalı bulunur, insanlar da onu esas alanının dışındaki ya da kenarındaki durumlar için denerler. Fonksiyonlarını genişletme baskısı daha çok, temel fonksiyonlarını seven ve kullanan kullanıcılardan gelir.

İkincisi başarılı yazılım, kendisi için yazıldığı makinanın normal hayatından çok daha uzun yaşar. Yeni bilgisayarlar olmasa bile yeni diskler, yeni ekranlar, yeni yazıcılar gelmeye devam eder; ve yazılım, bütün bu yeni olanaklar getiren yeni araçlara uyum sağlamak zorundadır.

Kısaca yazılım ürünü, uygulamalar, kullanıcılar, kurallar ve makina araçlarının kültürel matrisinde gömülü olarak bulunmaktadır. Bunlar sürekli değiştiğinden, onlardaki değişimler ister istemez yazılım ürününü de değişmeye zorlayacaktır.

Görünmezlik: Yazılım görünmez ve gözde canlandırılamaz. Geometrik soyutlamalar güçlü araçlardır. Bir binanın kat planı, hem mimarın hem de müşterinin, mekanı, trafik akışlarını, görünümü

değerlendirmelerine yardımcı olur. Çelişkiler ve unutulanlar açığa çıkar. Mekanik parçaların küçültülmüş çizimleri ve moleküllerin şekilleri, her ne kadar soyutlama olsalar da, aynı amaca hizmet eder. Geometrik gerçeklik, geometrik bir soyutlamayla yakalanır.

Yazılımın gerçekliği tabiati itibariyle mekanda gömülü değildir. Dolayısıyla da yazılım, arazilerin haritaları, silikon yongaların diyagramları, bilgisayarların bağlantı şemaları gibi hazır bir geometrik sunuma sahip değildir. Yazılım yapılarının diyagramlarını çizmeye girişir girişmez, bir değil ama birkaç tane birbiri üzerine getirilmiş genel yönlü grafikler elde ederiz. Birkaç graifk kontrol akışını, data akışını, bağımlılık şablonlarını, zamandaki adımları, isim-uzay ilişkilerini temsil eder. Bu grafikler düzlemsel bile değildir, çok daha az hiyerarşiktir. Gerçekte böyle bir yapının üzerinde kavramsal bir kontrol oluşturmanın yollarından birisi bir ya da daha fazla grafiğin hiyerarşik olması için bağlantıları kesmeye çalışmaktır. [1]

Yazılım yapılarını kısıtlamak ve basitleştirmek yolundaki ilerlemelere rağmen yazılım yapıları, doğaları itibariyle canlandırılamaz olmaya devam edecektir ve bu durum da zihnin, en güçlü kavramsal araçlarının bir kısımını kullanmasına imkan vermeyecektir.

Böyle bir eksiklik, sadece bir zihindeki tasarım sürecini engellemekle kalmaz aynı zamanda birden fazla zihin arasındaki iletişimi de sekteye uğratır.

Geçmiş Yenilikler Arızi Zorlukları Çözdü

Yazılım teknolojisinin gelişmesinde geçmişte en verimli olmuş üç adımı ele alsak, her birinin yazılımı inşa etmede farklı bir ana zorluğu çözmeye giriştiğini görürüz, ama bu güçlükler arızi güçlüklerdir, asli değil. Aynı zamanda böyle her bir girişimin varacağı yerle ilgili doğal sınırları da görebiliriz.

Yüksek seviyeli diller. Pek tabi ki, yazılım üretkenliği, güvenirliği ve basitliği için en güçlü çaba, programlama için yüksek seviyeli dillerin gittikçe artan şekilde kullanımıdır. Yüksek seviyeli dillerin, güvenilirlik, basitlik ve anlaşılırlıktaki kazanımlarla birlikte üretkenlikte en az 5 katlık bir ilerleme sağladığı, pek çok gözlemci tarafından teslim edilmiştir.

Yüksek seviyeli bir dil ne sağlar? Bir programı, arızi olan karmaşıklıklardan kurtarır. Soyut bir program, kavramsal yapılardan oluşur: işlemler, veri tipleri, programdaki adımlar ve haberleşme. Somut olan makina programı ise bitler, kütükler, dallar, kanallar, diskler vb. şeylerle ilgilidir. Yüksek seviyeli diller, bir soyut programda olması istenilen yapıları içerip, daha aşağı seviyeli olanlarından kaçındığı oranda, programlarda zaten hiçbir zaman doğal olmamış bir karmaşıklık seviyesini de ortadan kaldırırlar.

Bir yüksek seviyeli dilin yapabileceği, en çok, soyut bir programda, programcının hayal edebileceği yapıların hepsini sunmaktır. Veri yapıları, veri tipleri ve işlemler hakkındaki düşünce seviyemizin gittikçe yükselmekte olduğu kesindir, fakat bu yükselmenin hızı devamlı düşmektedir. Ve dil geliştirme yaklaşımları gittikçe, kullanıcıların sahip olduğu karmaşıklık seviyesine yaklaşmaktadır.

Dahası, bir noktada, yüksek seviyeli bir dilin sahip olduğu detayın yarattığı kullanım zorluğu, böylesi sadece ehlinin anlayabileceği yapıları nadiren kullanan kullanıcının zihni yükünü azaltmak yerine arttıracaktır.

Zaman paylaşımı. Zaman paylaşımı, programcıların üretkenliğine ve ürünlerinin kalitesine ciddi bir ilerleme getirmiştir ama bu ilerleme yüksek seviyeli dillerin kazandırdığı kadar büyük olmamıştır.

...

(Bu kısımın çevirisi sonra tamamlanacaktır.)

Birleşik programlama ortamları. Unix ve Interlisp, yaygın bir kullanım alanı bulan ilk entegre programlama ortamları olarak, üretkenliği çok küçük oranlarda ilerletmiş görünmektedirler. Neden?

Bunlar, entegre kütüphaneler,birleşik dosya formatları, programlar arasında haberleşme kanalları ve filtreler sağlayarak, tek tek pek çok programı bir arada kullanmaktan kaynaklanan arizi zorlukları hedef edinmiştir. Sonuç olarak da, prensipte daima birbirini çağırıp, besleyip ve kullanabilecek olan kavramsal yapılar, pratikte de bunları kolayca yapabilir hale geldiler.

Bu yenilik de, bütün araçların geliştirilmesini tetikledi, çünkü her yeni araç standart formatları kullanan herhangi bir programa uygulanabiliyordu.

Bu başarılardan dolayı bu ortamlar günümüz yazılım mühendisliği araştırmalarının konusudur. Bu araçların vaad ettikleri şeylere ve kısıtlarına bir sonraki kısımda göz atacağız.

Gümüş İçin Umutlar

Şimdi de potensiyel gümüş kurşun olarak ileri sürülen teknik gelişmelere bakalım. Bunlar hangi problemleri ele aldılar – asli problemleri mi yoksa geride kalan arızi zorlukları mı? Bunlar devrim ya da iyileştirici nitelikle ilerlemeler mi sundular?

Ada ve diğer yüksek seviyeli dil ilerlemeleri. Yakın geçmişte en çok övülen gelişmelerden biri, 1980'lerin genel amaçlı yüksek seviyeli dili olan Ada'dır. Ada, sadece dil kavramlarındaki evrimsel gelişmeleri yansıtmakla kalmaz, aynı zamanda modern tasarım ve modularizasyonu öne çıkaran özellikleri de içerir. Belki de Ada felsefesi, Ada dilinden daha ileridedir çünkü o modularizasyonun, soyut veri yapılarının ve hiyerarşik yapılandırmanın felsefesidir. Ada, ihtiyaçların tasarımını belirlediği bir sürecin tabi sonucu olarak, aşırı zengindir. Bu ölümcül bir durum değildir, çünkü daha alt seviyede

iş gören kavramlar öğrenme problemini çözebilirler ve donanımdaki ilerlemeler de bize, derleme maliyetlerini karşılamak üzere daha ucuz MIPS sağlarlar. Yazılım sistemlerinin yapılandırmada ilerleme sağlamak, paramızın satın alabileceği daha çok MIPS için iyi bir yöntemdir. 1960'larda bellek ve hız maliyetlerinden dolayı çok yerilen işletim sistemleri, geçmişdeki donanım dalgalarının MIPS ve ucuz belleğinin bir kısımını kullanmanın en mükemmel yolu olduklarını gösterdiler.

Yine de Ada, yazılım üretkenliğini engelleyen canavarı öldürecek gümüş kurşun olamayacaktır. Herşeyden önce Ada sadece bir diğer yüksek seviyeli dildir ve bu gibi dillerden elde edilen en büyük fayda ise ilk geçişte elde edilmektedir ki bu da, makinanın arızi karmaşıklığını, daha soyut olan ve adım adım komutlarla oluşturulan çözümlere dönüştürmedir. Bir kere bu arızi zorluklar ortadan kaldırıldı mı geriye kalanlar daha küçükleri kalacak ve bunları yok etmekle elde edilecek fayda da kesinlikle daha az olacaktır.

Ada'nın etkinliği değerlendirildiğinde, şu andan itibaren 10 yıllık bir zaman zarfında, Ada'nın çok temelden bir fark yaratmış olacağının görüleceğini tahmin ediyorum, ama bu ne herhangi bir dil özelliğinden dolayı ne de hepsinin bir arada toplanmış olmasından dolayı olacaktır. Ne de yeni Ada ortamları, ilerlemelerin sebebi olduklarını kanıtlayabilecektir. Ada'nın en büyük katkısı , onu kullanmaya geçişin, programcıları, modern yazılım tasarımı tekniklerinde eğitilmesine sağlamasında olacaktır.

Nesne-merkezli programlama. Pek çok sanat öğrencisi nesne-merkezli programlama için, günümüzün teknik modalarından daha çok umut beslemektedir.[2] Ben de onlardan biriyim. Dartmouth'dan Mark Sherman'ın CSNet News'da belirttiği gibi aynı isimle bahsedilen iki farklı kavram ayırt edilmelidir: *soyut veri tipleri ve hiyerarşik tipler*. Soyut veri tipi kavramı, nesnenin tipinin, aslında saklı olması gereken bellekteki yapısıyla değil de bir isim, uygun bir değerler kümesi ve uygun bir işlemler kümesiyle tanımlanmasıdır. Örnekleri, Ada'nın (private tipli) paketleri ile Modula'nın modulleridir.

Simula-67'nin sınıfları gibi hiyerarşik tipler ise, daha sonra sağlanacak alt tipler tarafından detaylandırılabilecek genel arayüzler tanımlamamıza izin verir. Bu iki kavram birbirinden bağımsızdır, saklama olmadan hiyerarşiler ya da hiyerarşiler olmadan saklama olabilir. İki kavram da, yazılım bina etme sanatında gerçek ilerlemeleri temsil etmektedirler.

Fakat yine de her biri, tasarımcıya, tasarımın aslını, hiç bir bilgi katkısı olmayan büyük miktarlarda sözdizimsel malzemeyi kullanmak zorunda olmadan ifade etmesine izin vererek, süreçten bir diğer arızi zorluğu kaldırmaktadır. Hem soyut veri yapıları hem de hiyerarşik yapılar için sonuç, daha yüksek seviyeli bir arızi zorluğun kaldırılması ve tasarımın daha yüksek seviyeli ifadesine izin verilmesidir.

Yine de böyle ilerlemeler, tasarımın ifadesindeki bütün arızi zorlukları ortadan kaldırmadan daha fazlasını yapamaz. Tasarımın zorluğu aslidir ve böyle girişimler bunda hiçbir değişiklik yapamazlar.

Eğer hala programlama dillimizde olan gereksiz tip tanımları, bir programı tasarlamada var olan işin onda dokuzunu oluşturuyorsa nesne-merkezli programlamayla on katlık bir ilerleme elde edilebilir. Ama bunda şüphem var.

Yapay Zeka. Pek çok kişi yapay zekadaki ilerlemelerin, yazılım üretiminde ve kalitesinde on katlık bir ilerleme sağlayacak bir gelişme çıkaracağını umuyor. Ben ise ummuyorum. Neden böyle olduğunu görmek için "yapay zeka"dan neyin kastedildiğini anlamamız lazım.

D. L. Parnas, yapay zeka konsundaki terminoloji karmaşasını giderdi: [4]

Günümüzde YZ'nin tamamen farklı iki tanımı yaygın kullanımdadır. YZ-1: Geçmişte sadece insan zekasını uygulayarak çözülebilen programların çözümü için bilgisayarların kullanılması. YZ-2: Deneysel ya da kural bazlı programlama olarak bilinen belli özel programlama tekniklerinin kullanılması. Bu yaklaşımda, uzmanların problem çözerken hangi başlıca deneysel yaklaşımları ya da kuralları kullandıkları incelenir... Program da insanların problemi çözdükleri gibi çözecek şekilde tasarlanır.

İlk tanımın kaygan bir anlamını vardır. Günümüzde bir şey YZ-1'in tanımı içerisine girebilir ama programın nasıl çalıştığını ve problemi anladığımızda onu artık YZ olarak düşünmeyiz. Malesef sadece bu alana has hiçbir teknoloji düşünemiyorum. İşin çoğu probleme özgüdür ve onu taşımak için bir miktar soyutlama ve yaratıcılık gereklidir.

Yukarıdaki değerlendirmeye tamamen katılıyorum. Konuşma tanımadaki teknikler, resim tanımlamada kullanılanlarla çok az ortak noktaya sahiptir ve ikisi de uzman sistemlerde kullanılanlardan farklıdır. Resim tanımanın programlama pratiğinde nasıl bir fark yaratacağını görmede zorlanmaktayım. Aynı problem konuşma tanımada için de geçerlidir. Yazılım geliştirmede zor olan şey ne istediğine karar vermektir, onu söylemek değil. İfade etmeyi kolaylaştıran hiçbir şey önemsiz kazanımlardan fazla birşey veremez.

Uzman sistemler teknolojisi, YZ-2, kendisi için ayrı bir yeri haketmektedir.

Uzman sistemler. Yapayy zekanın en ileri ve en çok uygulanan kısmı, uzman sistemleri geliştirmedeki teknolojidir. Pek çok yazılım bilimcisi, bu teknolojiyi yazılım geliştirme ortamlarına uygulamada zorluk çekmektedir. [3,5] Kavram nedir ve ihtimaller nelerdir?

...

(Bu kısımın çevirisi sonra tamamlanacaktır.)

Otomatik programlama. 40 yıldır insanlar "otomatik programlama", ya da problem tarifindeki bir ifadeden problemi çözmek için program üretilmesini beklemekte ve hakkında yazmaktadır. Bazıları bugünlerde, bu teknolojinin bir sonraki devrimi yaratacağını umarmış gibi yazmaktalar. [5]

...

(Bu kısımın çevirisi sonra tamamlanacaktır.)

İhtiyaçların detaylandırılması ve hızlı prototip geliştirme. Bir yazılım sistemi geliştirmenin en zor olan kısmı, ne geliştireceğinize kesin olarak karar vermektir. Kavramsal olan böyle bir işin diğer hiçbir kısmı, detaylı teknik ihtiyaçları, bütün kullanıcılara, makinalara ve diğer yazılım sistemlerine olan arayüzleri de dahil olacak şekilde, belirlemek kadar zor olamaz. Diğer hiçbir kısmı, yanlış yapıldığında, üretilen sisteme bu kadar zarar vermez. Diğer hiçbir kısmın daha sonradan düzeltilmesi, bu kadar zor olmaz.

...

(Geri kalanın çevirisi sonra tamamlanacaktır.)