Günümüzde Yazılım Mühendisliği ve Yeni Açılımları

Doğan Aydın
Tahir Emre Kalaycı
Ahmet Cumhur Kınacı

İleri Yazılım Mühendisliği
Sunum Raporu
08.05.2007

Işığı Görebilmek

Bir yazılım geliştirim sürecinde ilk olarak kullanılan model waterfall modeli olmuştur. Aslında bu model başlı başına uydurulmuş bir model değildi ve diğer mühendislik disiplinlerinde var olan bir sürecin yazılım mühendisliğine aktırılmış haliydi. Belki de daha önceden deneyimlerle olgunlaşmış bir modelinin yazılım üretiminin bir süreç modeli haline getirilmesi fikrinin yeni ortaya atıldığı bir zamanda kullanılmaya başlanılması iyi bir fikirdi. Daha sonra yapılan bir çok projede waterfall modelinin yetersiz kaldığı ve yazılım süreci için pek de uygun bir seçim olmadığı ortaya çıktı.

Waterfall'ın temel olarak iki hatalı yaklaşımı vardı. Birincisi waterfall modeli tasarımı mükemmel ve kolay kullanılabilir, gerçekleştirimi sağlama ve test aşamasında hataları fark edebilmenin esnek ve kolay olduğu varsayımını benimsiyordu veya kurulan model böyle bir sistem için tasarlanmıştı. Fakat yazılım sürecinde hatlar bu kadar keskin değildi. Tasarım bir çok projede mükemmellikten uzak olduğu gibi test aşamasında bir çok hata ya anlaşılamıyor veya çok geç kalındığından tasarımda gerçekleştirilmesi zor değişiklikleri doğuruyordu. Bu da maliyet ve zaman anlamına geliyordu.

Diğer önemli hatası tüm sistemin proje sonunda bir defada oluşturulmasıydı. Tüm parçalar bir araya getiriliyordu ve testlerle son halini alıyordu.

Bu eksikleri önlemek için model üzerinde iyileştirilmelerin yanında yine modeli reddetmeyen radikal fikirler de ortaya çıktı. Bunlardan bir tanesi "Çöpe atmak için planla; zaten sonunda atacaksın" fikrini savunmuştur. Sürecin kötü olduğunu kabul etmekle birlikte yeni bir model ortaya atmaktansa radikal bir ilaç üretmeye çalışmıştır. Süreç üzerindeki ilk iyileştirmeleri ise 1970 de Winton Royce yapmıştır. Bu iyileştirmeler kısaca maddelenecek olursa;

- Geri beslemeler kullanmak. Böylece proje sonunda değil de süreç esnasında gerekebilecek değişikliklere önceden cevap verebilmeyi sağlamak.
- Geri beslemeleri sadece bir önceki aşama ile kısıtlayarak

maliyet ve zaman kazancı elde etmek söylenebilir.

Yapılan projelerde aynen somon balıklarında olduğu gibi waterfall(şelale)'daki akıntıya karşı ilerleyiş ortaya çıkmış. Süreç boyunca elde edilen deneyim ve fikirler sıçramalara neden olarak tasarım üzerinde tekrar gözden geçirilmelere yol açmış. Çalışmalar tasarım özelliklerine en başından bağlı kalınmasının performansı düşürdüğünü ve tekrar üzerinde çalışılması gerekildiğini göstermiş.

Uzun bir süre yazılım şirketleri waterfall da ısrar etse de ışığı görebilmiş ve waterfall'a alternatif modellere yönelmişlerdir. Şu anda bir çok model waterfall'a tamamen zıt bir politika ile gerçekleşmekte ve genellikle iteratif bir gerçekleştirim yolu tercih etmektedir.

The Mythical Man-Month

Yazılım mühendisliği metodolojilerine savaş açılmaya başlandığı 1975'lerde Brook bir kitap yazdı. Akademik temelli değil de piyasada yıllardır çalışan birisi olarak yazılım geliştirme sürecinin mühendislik odaklı değil insan odaklı bir süreç olduğunu iddia ediyordu. Brook'un bu kitabı yeni metodolojisinin temelini teşkil ettiği gibi diğer metodolojilerin çıkmasına da ön ayak oldu.

Yazılım Geliştirme Projelerinde Temel Unsurlar

Brooks kitabında kendi karşılaştığı ve şu anda etkisini gösteren bir çok problemden bahsetmişti.

The Tar Pit: "Yazılım projeleri insanların yapabileceğinden daha girift ve kompleks bir şeydir." Bu söylenilen şu anda da geçerliliğini koruyan bir yaklaşımdır.

The Mythical Man-Month: "Tüm sebepler bir araya getirilse de proje takvimi doğru hazırlanamaz." Bu da şu anda geçerliliğini koruyan bir hükümdür. Şu anda bile bir çok projede zaman tam anlamıyla tahmin edilememektedir. Belki de bunun sebebi yazılım geliştirmenin müzik ve resim gibi yaratıcı bir eylem olmasıdır.

Brook's Law: "Projenin sonlarında insan gücü eklemek projeyi

geciktirir.” İnsan gücü ile zaman doğru orantılı değildir. Çok fazla insan eklemek iletişim ağını ve alışma sürecini etkilediğinden gecikmelere yol açar.

The Second System Effect: “Hiç dizayna katılmamış bir insan için ikinci dizayn en tehlikeli sistemdir; genel yaklaşım üzerine dizayn etmektir.” Yeni deyimiyle buna “featuritis” denilir.

Neden Babil kulesi yıkılır?: “Zamanlama felaketleri, fonksiyonel uyumsuzluk ve sistem hataları ortaya çıkar çünkü sağ elin yaptığını sol el bilmez. Takımlar varsayımlarının olduğu yöne sürüklenir.” İletişim insanın becerisinin en zor kısmıdır. Bu da mythical man-month’ın çekirdeğini oluşturur.

Bir adım ileri ve bir adım geri: “Sistemin entropisi tüm hayatı boyunca artar”. Bu “bit rot” adı verilir. Tamirler yapının bozulmasına yol açar ve kargaşayı arttırır.

1975’den Bu yana

IDE ve OO

Waterfall’a karşı açılan savaş esnasında ortaya çıkan iki yaklaşım bu savaşa güç verdi. Bunlardan birisi bilgisayar destekli araçlar ile nesneye yönelik programlama idi.

Bir çok profesyonel kendi disiplinlerini kendi notasyonları ile gerçekleştiriyorlardı. Fakat belki de pre-GUI diye adlandırılacak bir yaklaşımla ilk defa Pascal IDE’si Borland tarafından ortaya atılarak hiç olmazsa grafik tarafında bir ortak ortam sunulmuş oldu. Gerçi bu tip bir yaklaşım daha önce Emacs tarafından gerçekleştirilmişti fakat Pascal IDE PC’ler için gerçekleştirilmişti. Daha sonra visual basic gibi diğer dillerde de uygulanan bu yaklaşım yazılım geliştirmede temel araçlardan biri haline geldi. Böylece bir çok uzmanın kendi geliştirdikleri CASE toollar artık unutulur hale geldi.

Nesneye yönelimli programlama ise yapısal programlamadan daha fazla sürece esneklik kazandırdı. Nesne yönelimli programlamada var olan sınıf, kalıtım, soyut sınıflar gibi yetenekler esnekliği sağladığı gibi

kodların tekrar kullanımını attırdığından çokça tercih edilen bir yapı haline geldi. Popülaritesi o kadar arttı ki devlette destek vermek üzere bir çok OO tabanlı dil yirmi sene içinde ortaya çıktı ve yapısal programlama mantığı neredeyse tamamen kaldırılabilir bir duruma ulaştı.

UML

OO programlamanın 15 senede büyük hız kazanmasıyla yapılan savaş 1997'de sona erdi. İnsanlar en önemli faktörün insan ve aralarındaki iletişim olduğunu anladılar. Başarılı bir projenin sadece sözel bilgi paylaşımı ile değil bilgisayarda gerçekleştirilen işlemlerin iyi dökümante edilmiş olmasıyla ortaya çıktığı görüldü. OO programlamanın getirdiği avantajla profesyoneller Unified Modelling Language adı verilen bir sistemi ortaya attılar. Böylece bir yazılım sistemleri kendi notasyonlarına kavuşmuş oldular ve iletişim büyük oranda güç kazandı.

The Mythical Man-Month Methodology

Brooks kitabında sadece bazı önemli noktalardan değil ayrıca yeni bir metodolojiden de bahsetmekteydi. Bu yaklaşım şu anda var olan ve uygulanan bir çok metodolojinin temelini teşkil etti. Şunu söyleyebiliriz ki daha sonraki tüm metodolojiler bir şekilde bu metodolojiden etkilendi. Bu metodolojinin özü şu idi: "Az fazladır". Bu metodoloji temel olarak beş bileşenden oluşur.

1. **Proje Dizaynı — Conceptual Integrity**
   Brook dizayn entegrasyonunun dizayn etme ve yapılandırmanın en önemli kısmı olduğunu iddia eder. Bunu "İçeriksel integrasyonu yapılmış bir sistem hızlı yapılandırılıp test edilir" diye ifade eder. Brook bu görüşününde;
   - Dizayn bir kişi veya küçük bir grup tarafından yapılmalıdır. Tüm komitenin dizayna çalışması felakete kapı açar.
   - Sistemin alt yapısını kuranlar ile gerçekleştirenler farklı kişiler olmalıdır.

Brooks’un bahsettiği alt yapı sistemin fonksiyonel özelliklerdir. Teknik gerçekleştirim ise kendine ait bir dizayna sahip olmalıdır. Yazılım geliştirmenin ana problemi bu fonksiyonel içerik ile gerçekleştirimin doğru bir şekilde entegre edilip map edilmesidir. Brooks bu konu hakkında fazla ayrıntıya girmez. Fakat OO programlama bu problemi ortadan kaldırmak için iyi bir çözüm olmuştur.

Şu anda yazılım projelerini iki kısıma ayırabiliriz; kullanıcı yazılımları ve programcı yazılımları. Birincisinde tasarımın hem kullanıcı komitesinden hem de profesyonel yazılımcılardan ortak bir ile oluşturulması gerekirken ikincisinde hem tasarlayan hem de kodlayan aynı kişi ve bu da tek kişi olabilir. Özellikle açık kaynak kodlu yazılımlarda gördüğümüz bu durum Brooks’un ikinci kuralına zıt bir yaklaşımdır.

2. **Proje Yapısı :**

Brooks haklı olarak proje takımının bir ölçek olmadığını savunur. Fazla kişi çok iş demek değil belki az iş demektir. Brooks bunun için iki parça çözüm önerisi getirir;

- Büyük grupları küçük gruplara bölmek. Brook kitabını OO dan önce yazdığından bunu gerçekleştirebilmek o zaman çok güçtü şimdi ise her şey bileşen ve sınıflarla ifade edilebildiğinden daha kolay uygulanabilir durumdadır.
- Her bir bileşen ve nesne bir gruba atanmalıdır. Onlar bunu dizayn edip gerçekleştirmelidir. Bu da bu zamanda kolayca yapılabilecek bir durumdadır.

Brooks’un modelinde iki tip takım bulunur.

Producer	The person who gets things done: builds the teams, divides the work, manages the schedule.
Director	The system architect, the person who creates the overall design.
Of course, all kinds of administrative help should be part of this team. Here are the main members of the **component team**:	
Chief programmer	Does most of the heavy lifting: designs, codes, tests, and documents. Research Brooks cites contends that good professional programmers are ten times more productive then poor ones, so it is quite realistic to have one main programmer do most of the programming work.
The programming assistant	Serves as both a sounding board and built-in redundancy. Can also assume some of the less arduous programming tasks. The chief programmer's apprentice.
Program clerk	Handles all the clerical chores of checking code in and out, creating system builds, and whatever mundane technical administrative tasks are involved in developing a lot of code.
Toolsmith	Constructs, updates, and maintains special tools needed by the team.
Tester	Develops system tests and test data.
Editor	Helps put the documentation into good shape.
Language lawyer	Develops and finds the neat programming tricks to get things done. The hacker's hacker.
Administrator	Handles money, people, space, and machines, and interfaces with the rest of the organization's administration.

3. **Proje Gerçekleştirimi: Plan to Trow One Away**

   Brooks kitabında proje spesifikasyonunun başta yapıldığını fakat proje ilerledikçe bunun her zaman için yetersiz kaldığını söyler. Çünkü geliştirdikçe sistem gereksinimleri değişir ve programcı problem kümesini daha iyi anlayabilir hale gelir. Bu yüzden savunduğu görüşü "önce planlama yap ama üzerine gitme vaz geç ve yeniden dizayn et" ti.

   Brook yeniden yazılmış versiyonunda bu fikrinden biraz vazgeçtiğini bunun waterfall'ı destekleyen ama yeni bir çözüm getirmeyen bir fikir olduğunu söyledi. Çünkü IDE'ler, application server'lar, yüksek seviyeli diller ve tekrar kullanımlı nesneler sayesinde yazılımın iteratif ve hızlı bir şekilde büyüme kaydedebileceği örnekleri görmüştü. İteratif ve hızlı bu yöntemler sayesinde kullanıcı geri bildirimleri problem kümesinin ve sistemin

öğrenilmesi daha projenin ilk aşamalarında sağlanabiliyordu. Bu da sistemi hızlı bir şekilde tekrar dizayn etmeyi sağlıyor ve kullanıcı ile etkileşimi güçlendirdiğinden daha iyi yazılımlar üretilebiliyordu.

Bu konu çerçevesinde bahsettiği birkaç metottan bahseder. Bunlar Simulatorlar ve prototiplerdir. Gerçek zamanlı reaktif sistemler genellikle çok komplex ve donanım ve yazılımın bütünleştirildiği projelerdir. Bir çok genel amaçlı ve özelleşmiş araçlar bu tip sistemlerin modellenmesi ve simülasyonunda kullanılır. Bu tip bir yöntemle sistemin prototipi çıkartılarak sistem test edilir ve problem seti anlaşılır. Daha sonra bu prototipler üzerinden gerçekleştirime geçilir.

Diğer bahsedilen bir yöntem ise “Microsoft’s Build Every Night” yaklaşımıdır. Bu artırımlı yaklaşımın bir versiyonu olup yazılım modüllerinin her gece yeni fonksiyonlar ekleyerek kontrol edilmesini ve kullanıcı testleri için yeni çalışan bir versiyonun elde edilmesini esas alır. Brooks bunu ilk duyduğunda bunun yanlış ve uğraştırıcı bir yol olarak görmüştür. Fakat erken versiyonlarla kullanıcı komitesinden erken geri bildirimler alınmış ve “nightly builds” test süreci için ileriki aşamalarda iyi bir yöntem sunmuştur. Ancak bu yaklaşım yazılım versiyonlarının iyi bir şekilde yönetimini gerektirir ki bunun için iyi araçlar gerektirir. Bunlara örnek olarak Concurrent Versions System (CVS) ve Unix’in “Make” aracını verebiliriz.

4. **Proje İletişimi - The Documentary Hypothesis**

   Brooks altı tane temel dökümantasyondan bahseder ve bunlar kim, ne, ne zaman, nerede ve ne kadar sorularına cevap verirler;

What:	**objectives**	Defines the needs to be met, goals, objectives, and priorities of the system.
What:	**product specification**	Begins as a proposal and ends up as documentation.
When:	**schedule**	
How much:	**budget**	
Where:	**space allocation**	
Who:	**organization chart**	The surgical teams.

iletişim bir projenin başarısında kilit rol oynar ve Brooks bu iletişimin sıkça ve her yönde olması gerektiğini savunur. İletişim kullanıcı ile geliştirici, yöneticiler ile çalışanlar ve tüm takım arasında akmalıdır. Brook bunun için bir proje çalışma kitabının hazırlanması gerektiğini böylece iletişimin ve doküman arşivinin sağlanacağından bahseder. Lotus Notes bu konudaki en iyi bilinen ticari yazılımlardan birisidir. Fakat internet ve web'in gelişmesiyle bir çok kullanışlı araçta çıkmıştır. Wiki'ler ve Web-tabanlı içerik yönetim sistemleri bunlara birer örnek olabilir. Ayrıca bu dokümanların hazırlanmasında UML'in önemini göz ardı etmemek gerektir.

5. **Proje Organizasyonu – Plan the Organization for Change**

   Jonathan Swift "There is nothing in this world constant but inconstancy." Yani "Dünyada sabit kalmayan tek şey değişkenliktir." der. Bundan yola çıkarak Brooks aynen sistemde ileriki aşamalarda değişime gidildiği gibi proje organizasyonunda da değişikliğe gidilmesi tezini savunur. Bir çok kitap var olan yönetim düşüncesinin sadece yazılım sürecini etkileyen bir şey değil ayrıca başarıyı etkileyen bir faktör olduğunu göstermiştir.

   Yine de yazılım geliştirmede kullanılan bir metod bir çok esnekliğe sahip olmalıdır. OO geliştirim ve surgical team yaklaşımı bu dizayn ve organizasyonel esnekliği sağlar.

Temel Tartışma: Kavramsal Bütünlük ve Mimar

Başarılı bir programlama ürünü kullanıcılarına uygulamanın tutarlı akli modelini sunmalıdır. Bu tutarlılık uygulamanın geliştirilmesinde olduğu kadar kullanıcı arayüzünde eylemler ve parametrelerde de geçerli olmalıdır. Kullanıcı tarafından gözlenen en önemli kavramsal bütünlük, uygulamanın kullanım kolaylığıdır.

Tek veya iki beyin tarafından tasarlanan zarif yazılım ürünleri vardır. Ancak çoğu yazılım bu şekilde bir gerçekleştirmeyi destekleyememektedir. Zamansal kısıtlar yüzünden ikiden fazla insanı bir araya toplayarak bir çok adam/ay gerektiren yazılımların da üretilmesi gerekmektedir. Karmaşık ve rekabete girecek yazılım ürünleri zaman çizelgesini zorlayıcı geliştirme aşamalarına sahiptir.

Bir yazılım ürünü geliştirirken sorulan (ve değişmeyen) en önemli soru tasarım ve geliştirme çalışmalarını nasıl düzenlemeliyiz ki, kavramsal bütünlük üründe mevcut olsun sorusudur. "The MM-M" tarafından vurgulanan merkez soru bu sorudur. Bu kapsamda büyük programlama projelerini yönetmenin, küçük programlama projelerini yönetmekten farklı olduğu vurgulanmıştır. Günümüzde de bu soru geçerliliğini yitirmemiştir.

Mimar[1]**:**Kitabın 4 ve 7 bölümlerinde en önemli eylem olarak bir kişinin projede mimar olarak çalışmasının gerekliliği gösterilmiştir. Mimar, ürünün genel akli modelini biçimlendirme ve kullanıcılara nasıl kullanılacağını anlatma görevine sahip bireydir. Bu çalışma ürün özelliklerinin nasıl kullanılacağı ve çalıştırılacağı ile ilgili detaylı belirtimleri de içermektedir. Mimar ayrıca kullanıcının geliştiricilerle arasındaki bağlantıyı da sağlamakla ve kullanıcının yazılımdan beklentilerinin karşılanması için geliştiricileri yönlendirmek ve bilgilendirmekle yükümlüdür. İşlevsellik, başarım, maliyet, büyüklük ve zaman arasındaki dengeyi sağlama görevi mimarın en önemli görevidir.

Mimar'ın görevini kolaylaştırmak için mimarinin kullanıcının algıladığı şekliyle ürün tanımının gerçekleştirimden ayrılması gerekmektedir. Ayrıca mimari ve gerçekleştirimin ayrılması tasarım görevleri arasında düzgün bir

1 http://en.wikipedia.org/wiki/Software_architect

sınır oluşmasına neden olmaktadır. Büyük ürünler için sistemi alt sistemlere ayırarak, her alt sistem için bir mimar atanması gerekmektedir. Bütün bu mimarları görevlendirecek bir ana mimarda bu alt sistemlerin sistem halinde bütünleştirilmesi işleminden sorumlu olacaktır.

Kavramsal bütünlük ürünün kalitesi açısından günümüzde de geçerliliğini koruyan bir kavramdır. Bir sistem mimarının olması kavramsal bütünlük açısından, genel resmin takip edilebilmesi açısından ilk adımdır. Bu ilkeler sadece yazılım ürünleri için değil, diğer sistem geliştirmeleri için de geçerlidir.

"Featuritis" ve Frekans tahminleme

Çok geniş kullanıcı kümeleri için yazılım geliştirmenin en büyük handikabı, her kullanıcıyı tatmin etmek için iş yapılmasının gerekliliğidir. Eskiye nazaran artık günümüzde çoğu işi yapmak için kişiye özel uygulamalardan ziyade genel uygulamalar kullanılmaktadır. Bu genel uygulamaların herkese uygun olması gerekliliği, bazen bu uygulamaların özellik zengini olmasına neden olmaktadır. Bu şekilde aracın çok şişman bir araç olduğu görülebilmektedir.

Bu aşamada mimarların ve geliştiricilerin kafasını kurcalayan en önemli sorunların başında hangi kullanıcıları öncelikli olarak tatmin etmeliyiz geliyor. Buna bir çözüm olarak kullanıcıların hata takip ve özellik isteklerini bir sistem aracılığıyla alıp, bir sonraki sürümde olmasını istedikleri özellikleri oylamalarını sağlayıp, en çok oylanan özellikleri gerçekleştirmek gösterilebilir. Elbette geliştirme takımının önceliğinin yüksek olduğuna inandığı hatalar ve özellikler kullanıcılara başvurulmadan düzeltilmelidir. Örnek olarak günümüzde önemli bir yer teşkil eden ve başarılarını kanıtlamış olan serbest ve açık kaynak yazılım ürünlerinin geliştirme ve yönetim aşamaları incelenebilir. Genel olarak kullanıcılardan ürün sürümleri arasında alınan geri bildirimler ve yeni özellik istekleri bir takip sisteminde toplanarak bir sonraki sürümü çıkarma döneminin başlangıcında tartışılmaya açılarak ya sadece oy hakkı olanların oylarıyla veya kullanıcıların oylarını kullanabildikleri bir sistem aracılığıyla karar alınmaktadır. Çoğu serbest yazılım projesinin "Benevolent Dictator for Life

(BDFL)"[2] adı verilen öncü bir geliştirici veya yöneticisi bulunmaktadır. Bu lider bazı kararlar almakta ve uygulanmasını sağlayabilmektedir.

Kullanıcıların kendilerine özgü yazılımlarla çalışabilmeleri için yaratılmış olan eklenti mantığını örnek olarak gösterebiliriz. Eclipse[3] geliştirme ortamı örnek olarak verilebilir. Kullanıcı standart bir Java ve gerekli temel bileşenlerden oluşan yazılımı indirdikten sonra, diğer geliştiriciler veya kullanıcıların ihtiyaçları doğrultusunda ürettikleri eklentileri indirerek Eclipse'e yükleyerek kendisine özgü bir Eclipse geliştirme ortamı kullanabilmesi bu tip eklentiye dayalı yazılım kullanım örneğidir. Ayrıca GNU/Linux[4] modül ve yazılımlarını seçerek kendi dağıtımınızı oluşturmanız buna bir örnek olarak verilebilir. Bu noktada bu iki yazılımın geliştirme yaklaşımı olan "Free/Libre/Open-Source Software (FLOSS)"[5] geliştirme yaklaşımının özellikle incelenmesi, bu tip yazılımların ne şekilde geliştirilip, kararların nasıl alındığı ve böylece yazılımlara karşı neden başarı kazandığını anlamamız açısından gereklidir.

Agile Ekosistem

Dijkstra'nın manifestosu yazılım geliştiriminde katı bir disiplini ön görüyordu. Fakat son zamanlarda bu görüşe tamamen zıt bir görüş ortaya atıldı yani disiplin olabildiğince azaltılmalıydı. Bu da bir manifesto ile yayınlandı ve adı "*Manifesto for Agile Software Development*" olarak nitelendirilen bu manifesto;

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- **Individuals and Interactions** over processes and tools.
- **Working Software** over comprehensive documentation.
- **Customer Collaboration** over contract negotiation.
- **Responding to Change** over following a plan.

That is, while there is value in the items on the right, we value the items on the left more.

çevik yaklaşımın anahtar katkısı 4M insan odaklı yaklaşımı açması ve

2 http://en.wikipedia.org/wiki/BDFL
3 http://www.eclipse.org/
4 http://en.wikipedia.org/wiki/Linux
5 http://en.wikipedia.org/wiki/FOSS

genişletmesiydi. Ayrıca, yazılım geliştirme sürecindeki insan faktörünün anlaşılması ve üzerine gidilmesinin ve nesne yönelimli geliştirmenin üzerinde vurgu yapmışlardır. Highsmith insan ve onun ilişkilerinin üzerine oturmuş bu yönteme vurgu yapmak için metot yerine "ekosistem" terimini kullanmayı tercih etmiştir.

Extreme Programlama (XP)

Extreme programlama agile metotlara ilgiyi arttıran bir metot olmuştur. Buna bir etki de isminin akıllıca seçilmesi olabilir. XP 4M yöntemiyle bir çok paralellikler gösterirken daha yeni yeni yaygınlaşmaya başlamıştır. XP'deki bazı önemli öğeler aşağıdaki gibidir;

- Use only 3-10 programmers.
- Arrange for one or several customers to be on sight providing ongoing expertise.
- Programmers work in pairs, two per workstation, following agreed upon coding standards.
- Development is done in three-week iterations, with delivered tested code at the end of each iteration, and a working system build delivered to customers at the end of every two to five iterations.
- The unit of specification is the "user story." For each iteration, programmers estimate how much time each user story will take, and the customer then sets priorities.

The team is led by a coach.

- The programmers need to constantly simplify the code. Programmers rotate pair assignments every couple days so that everyone is familiar with all the code.

Crystal

Alistair Cockburn bir metodoloji ailesi olarak Crystal'i kullanır. Bu yöntemin anahtar falsefesi "projeye tipine metodu ayarla" dır. Crystal'in iki boyutu vardır; renk ve sayı. Siyah renkli kristaller büyük ve çok insanla girişilen projeleri gösterir ve bunlar ağır metodları gerektirir. Kristallerdeki yüksek sayılar projenin riskini ifade eder ve ciddiyet ve ihtimam gerektirir. Cockburn bu konu hakkında şunları söyler;

...the core Crystal philosophy is that software development is usefully viewed as a co-operative game of invention and communication, with a primary goal of delivering useful, working software and a secondary goal of setting up for the next game.

Bu mantığa göre farklı projelerde kişilerin farklı psikolojiye göre hareket etmeleri ön görülmektedir. Bu da bu yaklaşımı oldukça zor kılar.

Açık Kaynak Yöntemler:

Diğer agile metodların tersine, özgür yazılımlar ve açık kaynak kodlu geliştirim her hangi bir metodoloji profesyonellerine sahip değildirler. Eric S. Raymond *“The Cathedral and the Bazaar”[1]* adı verilen bir manifesto yayınlayarak açık kaynak kodlu yazılımların nasıl bu kadar başarılı olduğunu açıklamıştır. Bu tip projeler insan odaklı ve 4M karakteristiklerinin bir çoğunu içinde barındırmaktadır. Açık kaynaklı projelerinde bütçe olmaz. Kaynak kısıtlı bir oyun mantığı yoktur. Buna karşın çok başarılı ücretsiz ve açık kaynak kodlu projeler vardır; örneğin GNU, Linux, Apache, Mozilla. Bunlar 4M’nin değerini ve geçerliliği doğrular.

Model Tabanlı Yazılım Geliştirme

Bilgisayar kullanımı başladığından beri soyutlamanın seviyesini arttırmak için araştırmalar yapılmaktadır. Programlama dilleri sürekli olarak soyutlamanın seviyesinin arttırılması şeklinde gelişmiştir. "Model driven development (MDD)" bu sürecin doğal bir sonucudur. Soyutlama seviyesinin arttırılmasıyla MDD karmaşık ama rutin olan programlama görevlerini (sistem kalıcılığı, birlikte işlerlik, dağıtıklık, vb.) otomatikleştirmektedir.

MDD için öncelikli motivasyon üretkenliği arttırmaktır. Böylece firmaların yazılım geliştirme çabalarından getirileri arttırılacaktır. Bu yarar iki yönlü olmaktadır:

- Gerçekleştirilen işlevselliğin arttırılmasıyla geliştiricilerin kısa süreli üretkenliğini arttırır.
- Ürünün eskimişlik oranını düşürerek geliştiricilerin uzun süreli üretkenliğini arttırır.

Kısa süreli üretkenlik ne kadar işlevselliğin gerçekleştirildiğine bağlıdır. Ne kadar çok işlevsellik eklenirse üretkenlik o kadar artmış demektir. Çoğu MDD aracı bu üretkenliği desteklemektedir.

Uzun süreli üretkenlik yazılım ürünlerinin uzun ömürlülüğüne bağlıdır. Bir ürün ne kadar uzun süreli işlevselliğini korursa, yatırımın geri dönüşü o kadar değerli olacaktır. Bu nedenle ikinci ve stratejik olarak daha önemli olan özellik değişime duyarlılıktır. Dört temel değişim biçimi özellikle önemlidir:

1. Personel: Geliştiriciler geliştirme ile ilgili önemli bilgiyi akıllarında tuttukça firmaların bu geliştiriciyi kaybederek bilgiyi kaybetme riski vardır.
2. Gereksinimler: Gereksinimlerin değişmesi en önemli sorunlardan biridir. Bu her zaman olmaktadır.
3. Geliştirme ortamları: Ürünler tek bir ortamda üretilmeye çalışılsa da bu ortamın değişmesi önemli etkiler yaratmaktadır.
4. Konuşlandırma ortamları: Geliştiriciler daha iyi konuşlandırma ortamlarını öğrendikçe eskisini değiştirmek isteyeceklerdir. Bu değişimin mümkün ve kolay olması gerekmektedir.

Model tabanlı yazılım geliştirme için OMG tarafından geliştirilen MDA yaklaşımı tanımlanmıştır. Bu yaklaşımda UML tabanlı modellerin farklı platformlar ve soyutlama seviyeleri için dönüşümleri için yapılması gerekenler anlatılmaktadır. MDA'nın en önemli amacı tasarımı mimariden ayırmaktır. MDA MDD için üç aşamalı bir ana çatıdır:

1. Herhangi bir platformdan bağımsız olarak geliştirilecek olan yazılım sisteminin modeli oluşturulur. Bu model “Platform Independent Model” (Platform Bağımsız Model) adını almaktadır.
2. PIM bir veya daha fazla “Platform Specific Model” (Platforma özgü model)'e dönüştürülür. Bu dönüşüm işlemi bazı belli dönüşüm eşlemelerine göre gerçekleştirilir. PSM sistemi platforma özgü yapılarla tanımlar.
3. PSM'ler koda dönüştürülür.

Tam olarak model tabanlı yazılım geliştirme sayılmasa da yazılım geliştirme sürecini otomatikleştiren araçlar geliştirilmiştir. Örnek olarak UML modellerinden kaynak kodun oluşturulması verilebilir. Ayrıca bir uygulamayı temel bileşenleriyle (yetkilendirme, veritabanı, kalıcılık, web tabanlı arayüz, vb.) otomatik olarak UML araçlarından alınan modellerden

oluşturan kullanışlı araçlar da mevcuttur. Bu araçların en çok bilinen iki tanesi AndroMDA[6]ve OpenMDX[7]'tir.

Gerçek anlamda MDA sayılan işlem model dönüşümleridir. Model dönüşümü girdi olarak verilen bir modelin başka bir modele dönüştürülmesi işlemidir. Girdi ve çıktı modellerinin uyumlu olduğu üst modeller bu dönüşümün sağlanabilmesi için uygun olmalıdır. Model dönüşümleri için de çok farklı model dönüşüm dilleri tanıtılmıştır. Bunlara örnek olarak QVT, ALT, VIATRA, GReAT, Tefkat, Kermeta ve MT verilebilir[8][2].

"WIMP (Windows, Icons, Menus, Pointing)" ve Sonrası:

Kitabın yazıldığı tarih olan 1995 yılında bu yükseliş yeni başlamış olsa da artık günümüzde bu yükseliş tamamlanmış, çoğu yazılım bu yaklaşıma uygun özelliklerle geliştirilmektedir. Çoğu programlama dilinin ve işletim sisteminin bu şekilde yazılımlar üretmek için kullanılabilen kütüphaneleri bulunmaktadır. Artık üç boyutlu masa üstlerinden ve daha farklı bilgisayarla etkileşim arayüzlerinden bahsedilmeye başlanmıştır. Bu noktada artık birer kural olarak kabul edilmiş olan WIMP yaklaşımından ziyade yeni yaklaşımları incelemek gerekmektedir.

Bu yeni yaklaşımlara WIMP Sonrası ("Post WIMP") arayüzleri adı verilmektedir [3]. Bu kavram içerisinde iki boyutlu standart arayüz elemanlarına bağlı olmayan etkileşim teknikleri kapsamaktadır. En çok karşılaşılan örnek olarak kalem tabanlı harekete ("gesture") dayalı PDA ve benzeri araçlarda kullanılan etkileşim gösterilebilir. Bir diğer örnek olarak oyunlarda kullanılan direksiyon, oyun çubuğu gibi oyunlara özgü aletler verilebilir.

3B modelleme için 3B bileşenler sıkça kullanılmaya başlanmıştır. 3B sahnelerde artık 2B bileşenler yerine bu 3B bileşenler daha çok şey ifade etmeye başlamıştır. Bu bileşenleri VRML ve X3D gibi yeni 3B diller sağlamaktadır.

Ses tanıma kitapta her ne kadar kullanımı artacağı bahsedilen bir

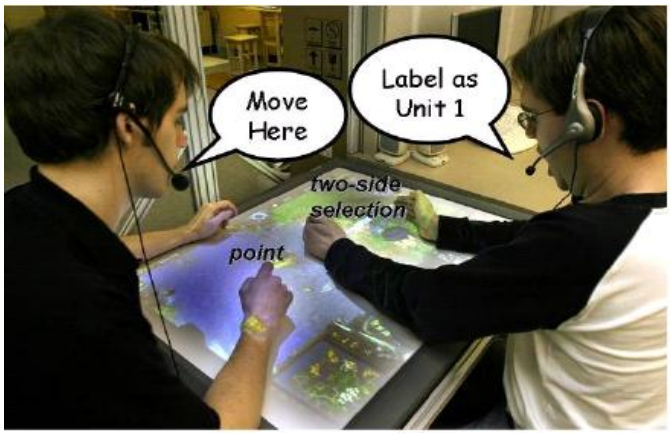
6 http://www.andromda.org/
7 http://www.openmdx.org/
8 http://en.wikipedia.org/wiki/Model_Transformation_Language

yöntem olsa da henüz sıklıkla kullanılabilecek bir olgunluğa ve kolaylığa erişmemiştir. Henüz ses tanımanın zorlukları tam olarak aşılamamıştır.
Bir başka yeni etkileşim aracı olarak Haptic'lerden bahsedilmelidir. Yakın zamanda alınabilir bir maliyete ulaşan hapticler henüz son kullanıcıya ulaşamamıştır. Haptic'ler ile amaçlanan kullanıcıların dokunma ve konum hislerini etkileşim içerisinde kullanmaktır.

Gelecekte bu alanın gelişmeye devam edeceği çok açıktır. Olası örnekler olarak giyilebilir bilgisayarlar, beyaz tahta veya duvar boyutunda görüntüleyiciler ve dokunmaya dayalı algılayıcılar, arttırılmış gerçeklik sağlayan ve kafaya takılabilen gözlük ve başlıklar, taslağa dayalı etkileşim cihazları, iki elinde kullanıldığı etkileşim araçları, akıllı tahtalar[9] sayılabilir.

GSI DEMO : Sayısal Masalar Üzerinden Tek Kullanıcılı Uygulamaların Jest ve Sesle Etkileşimli Olarak Çok Kullanıcılı Hale Getirilmesi

Bir çok ticari uygulama tek kullanıcı için ve klavye/fare ve monitör ile kullanılmak üzere tasarlanmakta. Buna karşın insanlar birlikte bir masa etrafında ortaklaşa çalışabilmesine yönelik uygulama ve araçlar mevcut değil. Bu proje ile amaçlanan bu tarz bir alt yapı oluşturmak. Ses, el hareketleri ve mimiklerle etkileşim sağlanabilmektedir. Uygulamaya için geliştirilen örnek donanımın kullanımı aşağıda gösterilmiştir.



Ubiquitous Computing :

Ubiquitous Computing amacı fazlalık oluşturmadan ve gerekli olmadığı durumlarda kullanıcıya görünmez olarak tüm fiziksel ortamlarda

9 http://mblog.lib.umich.edu/%7Erdivecha/archives/2006/02/the_world_of_sm.html

bilgisayar erişimini sağlamaktır. Sanal gerçeklikten farklı olarak günlük fiziksel dünya ile bilgi entegrasyonunu sağlar. Şimdiki kişiye özel sayısal yardımcı araçlardan farklı olarak ubiquitious computing tümüyle birbirine bağla cihazlardan oluşan, ucuz kablosuz ağlara sahip bir dünya hedefler. Bu yüzden bir PDA taşınmasına gerek olmaz çünkü bilgiye her yerden istenilen zamanda ulaşılabilir.

Özel yardımcı rolündeki, sesli iletişim kurulan kişisel etmen bilgisayarlardan farklı olarak ubiquitious computing öncelikli olarak arka planda çalışır. Kişisel bilgisayar verilen emirleri yerine getirirken, ubiquitious bilgisayar her şeyi kullanıcı kendi başına yapıyormuş hissi verir. Video ve ses kullanımının fazlalığı ile beraber sesli iletişim olması elektronik arayüzleri kişiler arası arayüzlere çevirir. Gelecekte yüzlerce küçük görüntüler günümüzdeki ofislerde kullanılan Post-it, çalışma kağıtları, duvar mesajları gibi şeylerin yerini alacak ve çalışma ortamıyla entegre bu sayede de kullanıcının günlük etkinliklerini olumlu yönde iyileştirir[4].

Anlamsal Web :

Günümüz İnternet ortamı öncelikli olarak insan kullanımı ve etkileşimi için tasarlanmıştır. Buna karşın Web servislerinin otomatik olarak birlikte çalıştırılmasında artış olmakta özellikle B2B ve elektronik ticaret uygulamalarında. Genellikle bu birlikte çalışma API'ler aracılığıyla HTML sözdizimi ve web sayfası gösterimleri kullanılarak içerik çıkarma temelli olarak çalışır. Eğer bir web sayfası biçimi değiştirilirse kullanılan API'nin de değiştirilip uyumlu hale getirilmesi gerekir. Temel olarak bilgisayar programlarının ve etmenlerin birlikte kararlı ve büyük çapta çalışabilmeleri için bu tür servislerin bilgisayar tarafında yorumlanabilir olması gerekir. Bu şekilde servislerin özellikleri, yapabilecekleri, arayüzleri ve etkileri karışık olmadan makine tarafından anlaşılabilir bir şekilde kodlanmalıdır.

Anlamsal Web'in gerçekleştirimi yapay zeka etkilenimli içerik ifade etme dillerinin, örneğin OIL, 3 DAML+OIL ve DAML-L, geliştirimi ile birlikte devam etmektedir. Bu diller güçlü anlamsal ifade yeteneklerine, karmaşık

sınıflandırma ile mantıksal ilişkilerin gösterimi özelliklerine sahiptir[5].

Web Intelligence :

Web'de Zeka (Web Intelligence), yapay zeka yöntemlerinin web tabanlı sistemlere uyarlanması ile ilgilenir. Yeni nesil web tabanlı ürünler, sistemler, servisler ve etkinliklerde yapay zeka ve ileri bilgi teknolojilerinin kullanılması üzerine çalışmalar yapılan bir bilimsel araştırma geliştirme alanıdır. Temel amacı Web ortamında veri üretimi çok hızlı olmasından dolayı Web ortamının getirdiği yararların tam kullanımı ve arttırımını sağlamak. Bu konuyla ilgili araştırmaları organize etmek ve yönlendirmek üzere "The Web Intelligence Consortium" (WIC) kurulmuştur.

Webde Zeka şu ana konu başlıkları altında incelenebilir :

1. Web Bilgi Erişimi : Arama, ontoloji, semantic web, metin analizi
2. Web Madenciliği ve Çiftçiliği : Web dosyaları veya belgeleri kullanılarak model bulma, belgeler arası bağlantılar.
3. Sosyal Ağ : İnsanların web ortamında oluşturduğu gruplar. Örneğin belli bir konuda uzman kişi bulunması.
4. "Grid" Hesaplama : Büyük ölçekli hesaplama problemlerinin web ortamında çözülmesi
5. Bilgi Ağı ve Yönetimi: Web'deki bilgilerin güncellenmesi ve düzenlenmesi.
6. Çoklu Model Etkileşim : Konuşma, el yazısı gibi farklı yollarla kullanılabilen ve sesli, görsel vb şekilde cevap veren uygulamalar.
7. Her yerde olan Hesaplama : Etkileşim ortamının arkasında web üzerinde bir çok uygulamanın 'görünmez' olarak çalışması.
8. Web'de Zeka Sistemleri Uygulamaları : Diğer konularla bütünleşik olarak çalışacak uygulamalar.

Etmen Tabanlı Yazılım Mühendisliği :

Yazılım etmenleri (software agent) teknolojisi; açık, dinamik değişimlerin çok olduğu ve heterojen ortamların yapısına uygun yazılımların geliştirilmesine yönelik bir teknolojidir. Genel olarak, yazılım etmenleri söz edilen türdeki ortamlarda özerk(autonom) bir biçimde davranma özelliğine sahip birimler olarak tanımlanmaktadır. Açık, dinamik ve heterojen ortamlara verilebilecek en iyi örneklerden birisi İnternet ortamıdır. Yazılım etmenleri, 1990'lı yıllardan itibaren İnternet üzerine dağılmış, değişik biçemlerde tutulan, sürekli değişen ve gelişen bilgilerin kullanıcı amaçları doğrultusunda derlenmesi ve işlenmesini gerektiren Web temelli uygulamalarda başarı ile kullanılmıştır.

Etmen Tabanlı Yazılım Mühendisliği (Agent Oriented Software Engineering) alanının temel amacı, etmen tabanlı yazılım sistemlerinin oluşturulması için gerekli geliştirme araçlarının/ortamlarının tasarımı ve gerçekleştirimini sağlamak ve daha sonra bu araçların/ortamların kullanımına dayanan geliştirme metodolojileri tanımlamaktır.

Kaynaklar:

[1] The Cathedral and the Bazaar
[2] OMG Model Driven Architecture
[3] Post-WIMP User Interfaces
[4] Weiser, M., Computer, Vol.26, Iss.10, Oct 1993 Pages:71-72
[5] SA McIlraith, TCH Zeng - Intelligent Systems, IEEE , 2001