Blog Yazılarım

SmartHomeProgrammer.com
Pratikprogramci.com
Kurumsaljava.com
Mikrodevre.com

4. Sürüm

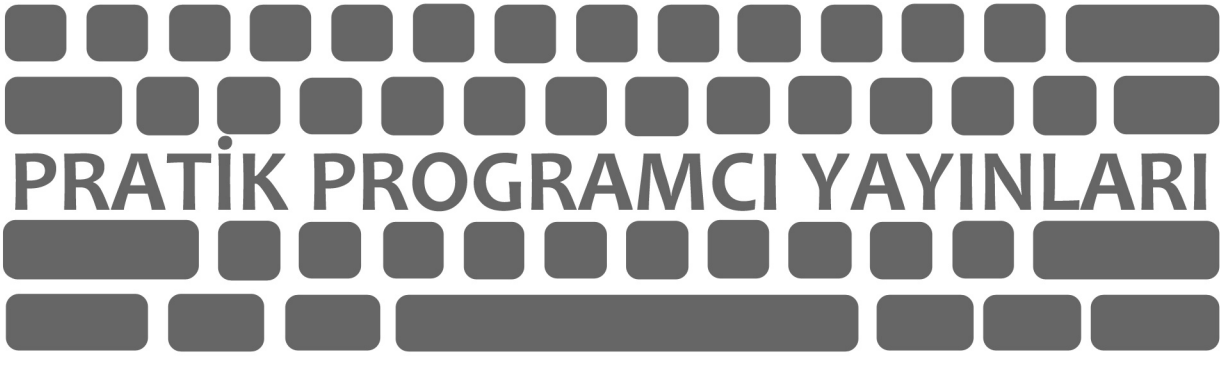
Özcan Acar

Blog Yazılarım

Özcan Acar'ın yazılım hakkındaki düşüncelerini yansıtan blog yazılarından alıntılar.

Özcan Acar

PRATİK PROGRAMCI YAYINLARI



www.pratikprogramci.com

İçindekiler

Önsöz

Merhaba. Ben Özcan Acar. Vakit buldukça KurumsalJava.com, Mikrodevre.com, SmartHomeProgrammer.com, Ozcanacar.com ve PratikProgramci.com adreslerinde yazılım, elektronik ve akıllı ev sistemleri hakkındaki düşünce ve tecrübelerimi blog olarak yazıyorum. Bu kitapta yüze yakın seçtiğim blog yazılarımı bir araya getirdim ve size bir kitap formatında sunmak istiyorum. Vakit buldukça okur ve düşüncelerinizi benimle paylaşırsanız sevinirim. Bana

acar@agilementor.com

adresinden ulaşabilirsiniz.

Saygılarımla

Özcan Acar

Facebook | Twitter | FriendFeed | LinkedIn | Google+

Kitabın Revizyonları

Bu sayfada her yeni sürümle kitaba eklenen yazı başlıkları yer almaktadır.

Şubat 2016 - Dördüncü Sürüm

- Yazılımda Şemsiye Modeli
- Kıymeti Bilinmeyen Eskimiş Bilginin Kıymeti
- Daha İyi Bir Programcı Olmak İçin Sınırlar Nasıl Zorlanmalı?
- Karadelikler, Soyutluk ve Yazılım
- Paralel Evrenlerin Programcıları
- Raspberry PI Router Olarak Nasıl Kullanılır?
- Neden Frontend ve Backend Programcısı Tarihe Karışıyor
- Yazılımcıların Performansı Nasıl Ölçülür?
- Nasıl Usta Programcı Olunurmuş
- Programcıların Besleyip, Büyüttükleri Canavar
- Programcının Evrimi
- Merkezi Versiyon Yönetim Sistemlerinde Sürüm Almak İçin İş Akışı Nasıl Şekillendirilir?
- 2015 Yılına Geldik, Hala Spring'le Birlikte İnterface sınıf mı Kullanmak Zorundayız?

Nisan 2015 - Üçüncü Sürüm

- Bir Mahrumiyetin Sonu
- İşin İnceliklerini Sadece Ustası Bilir
- Java Şampiyonluğu Nedir ve Nasıl Olunur?
- Neden Kafam Bu Kadar Rahat?
- Java Dilinde Neden Göstergeçler (Pointer) Yok?
- 20 Yaş, 30 Yaş, 35 Yaş ve Bugünki Programcı Ben
- Kod Kalitesi Denince Akla Gelenler
- Kod Redaktörlügü
- Yazılımda Otopilot
- Mevcut Bir Uygulama Koduna Nasıl Adapte Olunur?
- Neden Spring, JPA ve Diğer Çatılar ÖğrenilmeMEli
- Versiyon ve Sürüm Numaraları Nasıl Oluşturulur?
- Hangi Programlama Dilini Öğrenmeliyim?s
- Kokan Kod – 1. Bölüm
- Akıllı Ev Projem
- Akıllı Evlerde Kablo Kullanımı Neden Önemli?
- Java String Nesnelerinin Hafıza Kullanımı Nasıl Azaltılır?
- Çalışan Bir Java Uygulamasında Bytekod Nasıl Değiştirilir?
- Teknik Borç Nedir ve Nasıl Ödenir?
- Başkalarının Kodu Okunarak Daha İyi Programcı Olunabilir mi?
- Çok Gezen mi Bilir, Çok Okuyan mı?
- Yazılımda Geviş Getirme Taktiği Nasıl Uygulanır?

- Yazılımcılığın Ne Olduğunu Anlamamış Bilgisayar Mühendisi
- Birim Testlerinde Beklentilerimi Daha Net Nasıl İfade Edebilirim?
- Başlama ve Bitirme Kriterleri

Agustos 2014 - İkinci Sürüm

- 500 Beygir Gücünün Hazin Sonu
- Dağın Ayağına Gelmesini Bekleyen Birisi
- En Basit Çözümü Oluşturma Yetisi Nasıl Kazanılır?
- Sözde Lean!
- Programcının Hayatını Kolaylaştıran 18 Alışkanlık
- Müşteri Gereksinimlerini Anladığımızdan Nasıl Emin Olabiliriz?
- 5 Adımda Daha Kaliteli Yazılım Testleri
- Ne Zaman Başımız Göğe Erer
- Kitap Okumanın Önemi
- Agile Türleri
- Yazılımcıların Gerçek Ekmek Teknesi
- Kod A.Ş. – Kod Anonim Şirketi
- Sorumluluk Sahibi Olmak

Aralık 2013 - Birinci Sürüm

- Programcılar Yazar Olsalardı Keşke!
- Acı Çekmeden Üstad Olunmaz
- Çevikliğin Böylesi
- Battı Balık Yan Gider
- Kod Kata ve Pratik Yapmanın Önemi
- Yazılım Maketleri
- Koddan Korkan Programcı
- Alışkanlıkların Gücü
- Kim Senior Programcıdır?
- TeletAPI'nin API'si
- Alet İşler, El Övünür
- Cahilliğime Verin
- Açık Sözlü Programcı
- Programcılık Çıtası Yükseliyor
- Nasıl Arkadaş Kazanır ve Diğer İnsanları Etkilersiniz
- Copy/Paste Programcı
- Test Edebilme Uğruna Her Şey Mübahtır!
- Deneme Yanılmanın Bedeli
- Veresiye Satan Yazılımcı
- Uzman ve Usta Yazılımcı Arasındaki Fark
- Göz Boyamaca
- Detayları Görebilmek Ya Da Görememek
- Yazılımda Çeviklik İflas mı Etti?
- Benim Çeviklik Tanımım

- Kişisel Gelişim
- Ultra Lüks
- Benlik Güden Programcılar
- Melek Programcılar ve Şeytan Yatırımcılar
- Mantığın Köleleri
- Meğer Neymişiz!
- Programcıyım, Yönetici Değil!
- Programcılık Sanat mı, Zanaat mı?
- Ekibin Dikkati ve Saygısı Nasıl Kazanılır?
- Eşli Programlama, Code Review, Code Retreat ve Adada Yaşayan Programcılar
- İnşaat ve Yazılım Mühendisleri Arasındaki Fark
- Kim Daha İyi Programcı?
- Neden Mikrodevre.com?
- Açık Kaynağa Destek
- Neden Her Programcının Bir Blog Sayfası Olmalı?
- Organizasyonel Değişim
- Böyle Girişimcilik Olmaz!
- Yeni Bir Programlama Dilini Öğrenmenin En Kolay Yolu
- Matrix'de Yaşayan Programcılar
- Ucuz Etin Yahnisi
- Ödünç İnternet
- Java Hotspot, Assembler Kod, Hafıza Bariyerleri ve Volatile Analizi
- Çöplerin Efendisi
- Neden Fonksiyonel Programlamayı Öğrenmek Zorundayız
- Bir Java'cının Gözünden Ruby
- BizimAlem.com – Bir Sistemin Tasarlanış Hikayesi

Programcılar Yazar Olsalardı Keşke!

http://www.kurumsaljava.com/2012/02/28/programcilar-yazar-olsalardi-keske/

Birçok programcının büyük bir büroda çalıştığını düşünelim ve böyle bir ortama girdiğimizi. Programcı olduklarını bilmiyoruz, ne yaptıklarını da. Bu çalışanların ne yaptığını düşünürdük? Büyük ekranlarda devamlı birşeyler yazdıklarını gözlemleyip yazar olduklarını düşünürdük belki. Birşeyler yazana yazar denir. Bu bir roman, bir ders kitabı ya da bir yemek tarifi olabilir ya da bir kod parçası.

Dışardan bakıldığında biz programcılar böyle algılanıyoruz. Bizler de birer yazarız ama bu ünvanı taşımayı hak etmiyoruz. Neden mi? Açıklamaya çalışayım.

Bir yazar kitaplarını okurları için yazar. Baştan savma yazmaz; nasılsa okur ne demek istediğimi anlar demez; bir kitap olsun da, içinde ne olduğu önemli değil demez; roman yazıyorsa roman kahramanlarına abx, xyz gibi anlaşılması zor isimler vermez; o isimleri özenle seçer, onlara kişilik, ruh ve beden verir; kısaca yazar işini severek yapar, bu uğurda yıllarını verir.

Şimdi bir yazarı bir programcı ile kıyaslayalım. İkisi de birşeyler yazıyor. Yazarın yazdıklarını okuyucuları okuyor, programcının yazdıklarını çalışma arkadaşları. Hangi grup daha mutlu okucuyu kitlesi? Ben ikinci guruba dahil olmakla beraber çok mutsuzum. Okuduklarım bana haz vermiyor. Hadi ondan da vazgeçtim. Ne yazdıklarını anlayabilsem…… Bahaneler de hep aynı: “yeterli zamanim yoktu”, “patron programı hemen bitirmemizi söyledi”…. Bir kitap yazarı bu bahanelerle bir kitap yazsa bu kitap ne kadar başarılı olabilir?

Haklı olarak “Bir romanın bir yazarı olabilir. Programlar zaman içinde onlarca ya da yüzlerce programcının elinden geçiyor” dediğinizi duyuyor gibiyim. Bu okunamaz kodun oluşması için bir gerekçe değil. Her programcının benimsemesi gereken birkaç prensip ve pratik ile bu sorunun önüne geçilebilir. Bunların neler olduğuna yazımın ilerleyen bölümünde değineceğim.

Elli yıl oldu neredeyse; programcılar program yazıyor, bilgisayarlar bu programları çalıştırmak için kullanılıyor. Elli yıl oldu ama biz programcılar hala yazarliğa terfi edemedik. Yazdıklarımız okunmuyor. Kendimiz bile bir süre sonra yazdıklarımızı okuyamıyoruz. Bu mu işimize verdiğimiz kıymet?

Elli satırlık bir metodu anlamak beni zorluyor; yüz satırlık bir metot beni çileden çıkartıyor. Sekiz yüz ya da daha fazla satırlık bir metot hakkında ne düşündüğümü anlayabiliyorsunuz sanırım…. Dışarda böyle metotlar bulmak çok kolay. Yazılım dünyası sanki böyle metotlardan oluşmuş…. Yazar olarak ne kadar kötüyüz…. Bu ünvanı hak edebilecek en son meslek gurubu biziz… biz programcılar.

Programcıların yazar olduğu bir evren düşlüyorum, gerçek yazarlar kadar yaptıkları iş taktir edilen, yazdıkları kodun herkes tarafından (buna ev kadınları da dahil) okunabildiği, kod yazarken değişken ve metot isimlerini roman kahramanı olarak görüp, isimlerini özenle seçtikleri, yazdıkları kodun bir hikaye anlattığı programcıların yaşadığı bir evren…

Böyle bir evren var ama bize yüzlerce ışık yılı uzaklıkta, ışık hızı ile hareket edebilsek bile

yüzlerce sene sonra erişebileceğimiz bir evren. İyi birer programcı olmaktan yüzlerce ışık yılı ötedeyiz! Şimdi bir düşünce deneyi yapalım ve bu evreni keşfedelim. Düşünce deneyimizde bizim evrenimizden bahsettigimiz evrene seyahat edecek bir programcımız var. Programcımız ışık hızının çok üstünde bir hızla bu evrene gidecek, oradaki programcılarla sohbet edip geri dönecek. Döndü bile; ne kadar hızlı gidip geldi degil mi? Bize neler getirdi ona bir bakalım. Bundan sonrasını bu seyahati gerçekleştiren programcının ağzından okuyacaksınız.

Yolculuğum uzun sürmedi, birkaç saniye... Uzay aracımla programcıların yaşadığı gezegene indim. Beni çok hoş karşıladılar. Bizlerden bir farkları yok. Beni çalışma ofislerine götürdüler. Bir ekip, evrenler arası ışık hızıyla hareket edebilen uzay araçlarının yazılımı ile uğraşıyordu. Koda bakıp bakmak istemediğimi sordular. İlk önce çekindim; programcı olarak böyle roket mühendisliğinden ne anlarım ki... kodu okusamda anlamam imkansız diye düşündüm. Sen yine de bir bak dediler. Baktım. Bir daha baktım. Birkaç satır okudum. O da ne! Ne kadar kolay okunuyor, okuduğum herşeyi anlıyorum. İnanılacak gibi değil. Bizim programlama dilleri gibi if/else/while yapılar kullanıyorlar, ama yazdıklari kod roman gibi okunuyor. Çok temiz ve sade. Metotlar çok kısa tutulmuş, verilen isimler çok anlamlı. İnsan ne okuduğunu hemen anlıyor. Çok şaşırdım! Bizim dünyamızda durum çok farklı. Zamanımızın büyük bir bölümünü başkalarının yazdığı kodu anlamak için saatler boyu debugging oturumlarında harcıyoruz. Nasıl program yazdıklarını soruyorum. İçlerinden birisi şunları söylüyor:

- İlk önce bir test yazıyoruz. Ortada hiç birşey yok, sadece test kodu var.
- Daha sonra kodu yazmaya başlıyoruz. Testler bizi yönlendiriyor. Herhangi birşeye dikkat etmeden testlerin çalışır hale gelmesi için kodu geliştiriyoruz. İlk sürümde metotlar uzun, isimleri anlaşılmaz olabiliyor, ama ilk sürüm de bunlar önemli değil. Önemli olan kodu çalışır hale gelmesi.
- Kod çalışır hale geldikten sonra tekrar tekrar gözden geçirip, herşey anlaşılır hale gelene kadar değişiklik (refactoring) yapıyoruz. Elimizde otomatik çalışan testler olduğu için refactoring bizim için hiç sorun değil.
- Yaptığımız her değişikliğin ardından testleri koşturarak oluşan yan etkileri tespit etmeye çalışıyoruz. Testler yan etkileri hemen bulmamızı sağlıyor. Bu bizim güvenimizi ve cesaretimizi artırıyor. Çok ufak adımlarla refactoring yapıyoruz. Kodu böylece yavaş yavaş yoğuruyoruz. -Programlarımızı *KISS* (Keep It Stupid Simple = Kısa tut) prensibine göre geliştiriyoruz. Her zaman en basit çözümü tercih ediyoruz. Basit olmayan çözümler kodun okunurluğunu olumsuz etkiliyor. Aramızda en iyi algoritmaları kim yazdı yarışı yapmıyoruz, en basitini ve okunuru kim yazdı yarışı yapıyoruz. Okunmayanar kod gözden geçirme oturumlarında diskalifiye oluyor
- Kod tekrarlamasını önlemek için *DRY* (Dont Repeat Yourself = Kendini tekrarlama) prensibine sadık kalıyoruz. Kod tekrarları programın geliştirilmesinin önünde büyük bir engel.
- Belirli etaplarda birden fazla çalışma arkadaşımızla kodu beraber gözden geçiriyoruz (code review). Burada amacımız kodu daha da okunur hale getirmek. Çalışan testlerimiz olduğu için kodu değiştirmek bizim için çocuk oyuncağı. Kodu bu şekilde yoğurmak bir zaman sonra çok zevkli bir uğraşı haline geliyor. En geç kod gözden geçirme oturumlarımızda uzun metotları ortadan kaldırma fırsatı buluyoruz. -Yazdığımız metotlar en fazla dört satırdan oluşuyor. Her sınıfın ve metodun sadece bir sorumluğu (SRP) var. Sınıf, metot ve değişken

isimlerini çok özenle seçiyoruz. Kesinlikle bir kod parçasını açıklamak için koda yorum eklemiyoruz. Kod yoruma ihtiyaç duyuyorsa o zaman ya seçtiğimiz isimler yetersiz ya da kodun birden fazla metoda bölünmesi gerekiyor. Kodu okunur hale getirmek için bir sebep daha.
- Genel olarak kod yazarken SOLID prensiplerine dikkat ediyoruz. SOLID testler kadar önemli.
- Testler bizim için yazdığımız kod kadar önemli. Test kodunu da zaman zaman gözden geçirip daha okunur hale getirmeye çalışıyoruz. Testlere üvey evlat muamelesi yapmıyoruz.

Bunların yanında *kırık cam* metaforu bize devamlı *izci kuralını* hatırlatıyor. Kırık cam ve izci kuralı nedir diye soruyorum. Programcı arkadaş şu şekilde açıklıyor:

Tamir edilmeyen bir cam mahalle sakinlerinde terkedilmişlik hissi uyandırır. Bu kırık cam binaya sahip çıkılmadığının göstergesi olarak algılanır. Kısa bir zaman sonra başka pencere camları kırılır ve bina sakinleri ya da bölgede oturan diğer şahışlar düzensizliğin artmasını çabuklaştırırlar. Binaya sahipleri tarafından sahip çıkılmadığı taktirde bina büyük hasar alır. Artan hasar nedeniyle bina sahipleri binayı tekrar düzene sokma şefkini yitirebilirler. Program kodları kırık cam misali düzensizlikler giderilmediği taktirde zamanla bakımı zor hale gelirler. Programcı izci kuralını uygulayarak koda sahip çıkar.

Her izcinin uyduğu bir kural vardır:

```
Kamp yaptığın yeri bulduğundan daha iyi bir şekilde bırak!
```

Biz programcı olarak geride bıraktığımız kodun kalitesi, işe başladığımızda bulduğumuz kodun kalitesinden daha iyidir. İzci kuralını uygulayan programcı kırık cam prensibinden dolayı oluşan düzensizlikleri gidermiş olur. İzci kuralının uygulanması durumunda kırık cam hiç oluşmaz ya da mevcut kırık camlar tek tek tamir edilmiş olur.

Bu programcılarla sohbetimiz saatler boyu sürdü. Birşeyi çok iyi anladım ki o da bu programcıların işlerini çok severek yaptıkları. Aramızda ne kadar büyük farklılıklar varmış! Bu programcıların benimsedikleri ve günlük iş hayatlarında uyguladıkları *prensipler*, *pratikler* ve bir *değer sistemi* varmış meğer. Vaybe! Keşke burada kalabilsem ve bu muazzam yazarlarla çalışabilsem. Onlarla çalışmak ne kadar zevkli olurdu kimbilir. Yaptıkları işten çok büyük haz aldıkları ve ortaya çıkan kodla gurur duydukları yüz ifadelerinden ne kadar da belli oluyor. Benim dünyamda programcılar devamlı endişe içinde. Kafalarındaki sorular hep aynı: "programı zamanında yetiştirebilecek miyim?", "anlamadığım şeyi nasıl değiştireyim ben şimdi", "bu metot bu kadar uzun olmak zorunda mı ya!", "neden bu mesleği seçtim? herkesin pisliğini ben mi temizlemek zorundayım?", "bir an önce yönetici olupta bu pislikten kurtulayım bari, zaten bir ömür programcılık yapan adama kim kıymet verir"..... Biz programcılar kendi dünyamızda çok mutsusuz. Bu değişmeli. Zaman ne çabukta geçmiş. Geri dönmem gerekiyor artık.

Uzay aracıma binmek ve dünyama geri dönmek üzere yola çıkıyoruz. Uzay aracıma doğru giderken tekrar yaşadıklarımı gözden geçiriyorum. Programcılık ve yazarlık arasındaki parallelliği ve farklılığı daha değişik bir gözle görüyorum. Programcılık yazarlık demek, bir yazardan farklı birşey yapmıyoruz, ama bir yazardan çok farklı bir şekilde yazıyoruz. Bunun değişmesi gerektiğini düşünüyorum. Bu mesajı mutlaka kendi dünyamdaki programcılarla paylaşmalıyım.

Böyle düşüncelere dalmışken, uzay aracıma geliyoruz. Beni bu evrenin programcıları uğurluyorlar. Bana veda ederken elime bir adet "Clean Code" isimli kitabın nüshasını tutuşturuyorlar. Aralarından birisi "biz bu kitabı sizin dünyanızda keşfettik, bizim için çok faydalı oldu, siz de mutlaka bir göz atın" diyor. Bu kitabı tanıyorum: Robert C. Martin yazmış. Okuma fırsatı bulamamıştım. Şimdi mutlaka okuyacağım. Elveda diyorum ve uzay gemime biniyorum. Birkaç saniye geçiyor... Tekrar kendi dünyamdayım. Artık bir yazar olmaya karar veriyorum, programcı bir yazar....

Siz de bir yazar mısınız?

Yazılımda Şemsiye Modeli

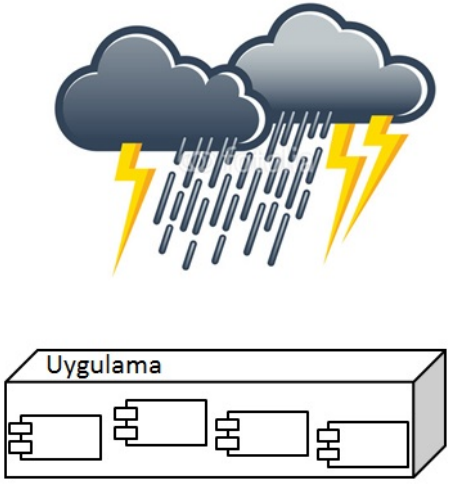
http://www.pratikprogramci.com/2016/02/26/yazilimda-semsiye-modeli/

Yazılımda testlerin gerekliliğini ve hangi testlerin ne zaman uygulandığını açıklamak amacıyla şemsiye modeli ismini verdiğim bir model oluşturdum. Bu yazımda sizlerle bu modeli ve işlevini paylaşmak istiyorum.

Şemsiyeler gerçek hayatta yağmurdan ve yer yer güneşten korunmak için kullanılır. Yazılım süreçleri için oluşturduğum şemsiye modelinde yazılım testleri uygulamayı korumak için açılan şemsiyeleri temsil etmektedir. Şemsiyenin büyüklüğüne ve işlevine göre uygulamayı hatalara karşı korumak ve sağlıklı bir şekilde gelişmesini sağlamak mümkündür.

Şemsiye modelinde değişik büyüklükte şemsiyeler yer almaktadır. Açılan bir şemsiyenin kapsadığı bir alan mevcuttur. Bu alan şemsiyenin koruyabildiği alandır. Değişik test türleri değişik büyüklükte alanları koruyabilirler. Testleri olmayan bir uygulamayı şemsiye modeli ayazda ve yağmurda kalmış olarak tanımlar. Böyle bir uygulamanın akıbeti bellidir.

Ayazda kalmış bir uygulamaya göz atarak başlayalım. Resim 1 de böyle bir uygulama görülmekte.



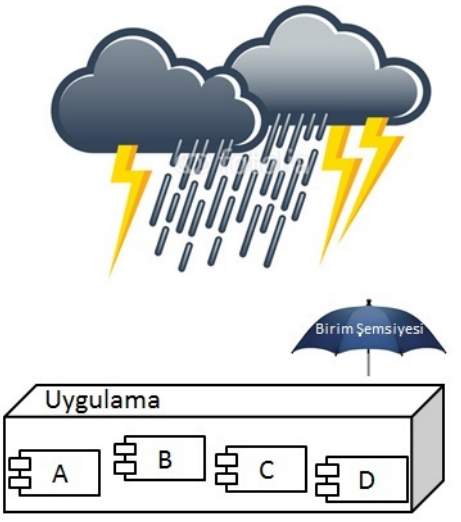
Resim 1

Resim 1 de yer alan uygulamanın yağmurdan korunması yani hatasız olması ve kalabilmesi için hiçbir şemsiye açılmamış yani test oluşturulmamıştır. Böyle bir uygulamanın geliştirilmesi tamamen rus ruleti benzeri bir girişimdir. Testler olmadığı sürece yani bir veya daha fazla koruyucu şemsiye açılmadığı sürece bu uygulama su altında kalmak zorundadır. Projenin belli bir aşamasından sonra ekibin tek işi, su dolan

teknenin batmasını engellemeye çalışmak olacaktır. Teknenin su dolmasını engelleyici bir mekanızma olmadığından, su tahliyesi bir şey ifade etmemektedir. Tekne er ya da geç batacaktır.

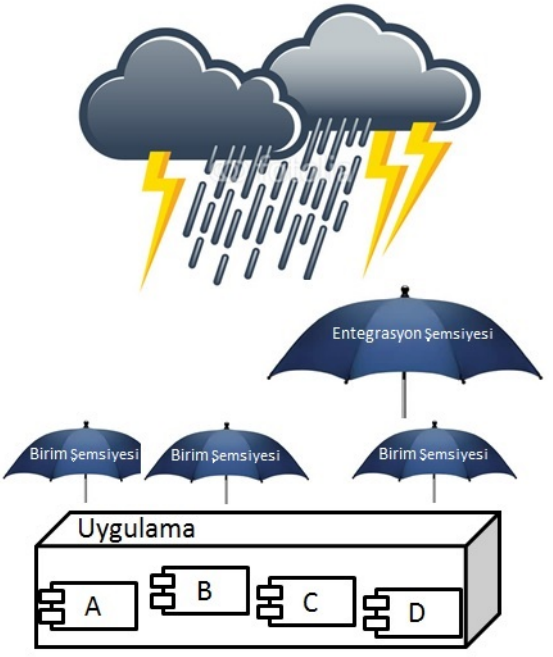
Şimdi her türlü hava koşulunda uygulamayı istenilen hedefe taşımak için atılması gereken adımlardan bahsetmek istiyorum. Şemsiye modelinde değişik büyüklüklerde, uygulamanın belli bir kısmını ya da tümünü yağmurdan koruyacak türde şemsiyeler yani test türleri yer alıyor. Küçük şemsiyelerin kullanımı ve taşınmaları yani test türü olarak oluşturulmaları ve koşturulmaları kolay. Bu tür testler için gerekli yatırım düşük. Buna karşın kapsama alanları kısıtlı yani küçük şemsiyelerle uygulamanın tümümü korumak mümkün değil. Bu küçük şemsiyeleri bir araya getirip, uygulamanın her tarafını koruma altına almak da mümmkün değil, çünkü yapıları itibari ile bir araya getirilip, daha büyük bir şemsiye oluşturma kabiliyetine sahip değiller. Buna karşın şemsiye modelinde tüm uygulamayı ya da büyük bölümlerini koruma altına alabilecek daha büyük şemsiyeler de bulunmakta. Bu tür şemsiyeler çok büyük bir alanı koruyabiliyorlar. Lakin oluşturulmaları ve koşturulmaları zaman alıcı. Bu yüzden maliyetleri yüksek. Şemsiye modeli ile değişik büyüklükte şemsiyeler kombine edilerek, uygulanın tümünü koruma altına almak mümkün. Şimdi bu şemsiyeleri, nasıl kullanıldıklarını, oluşturulma maliyetlerini ve uygulamayı nasıl koruklarını yakından inceleyelim.

Şemsiye modelindeki en küçük şemsiyeler birim testleridir. Birim testleri sadece bir kod birimi, bu bir metot, sınıf ya da modül olabilir, test etmek için kullanılırlar. Kod birimlerinin dışa bağımlılıkları mevcut olabilir. Birim testlerinde bu bağımlılıklar yok sayılır ve mock olarak isimlendirilen ve bağımlılıkları simüle eden kod birimleri ile yer değiştirilmeleri sağlanır. Bu şekilde sadece test edilen kod birimi bünyesinde olup, bitenleri test etmek mümkündür. Nihayi amaç kod biriminin bağımlılıkları olmadan nasıl davranış gösterdiğinin test edebilmesidir. Eğer teste test edilen kod biriminin bağımlılıkları da dahil olursa, test birim testi olmaktan çıkar ve bir entegrasyon testine dönüşür. Bu tür testlerin kullanımı da şemsiye modelinde yer almaktadır. Lakin birim testleri ile sadece işletme mantığı barındıran kod birimlerine konsantre olunur ve birim şemsiyesi açılarak belli bir kod birimi koruma altına alınır ve bağımlılıkları göz ardı edilir. Bu bağımlılıkların dolaylı olarak test edilmediği anlamına gelmektedir.



Resim 2

Resim 2 de birim şemsiyesinin işlevi görülmektedir. Bu örnekte D ismini taşıyan modül birim testleri ile koruma altına alınmıştır ve yağmur görmesi engellenmektedir. D ve diğer modüller arasında interaksiyon olduğunu düşünecek olursak, birim testleri ile bu interaksiyonu test etmediğimiz için D için açmış olduğumuz şemsiye sadece D bünyesindeki işleteme mantığını korur türdendir. Eğer interaksiyon içindeki modülleri tamamen koruma altına alacak büyüklükte bir şemsiyemiz yani testimiz yoksa, bu durumda uygulama entegrasyonun gerçekleştiği noktalarda yine yağmura mağruz kalacaktır, çünkü bu alanlar birim testleri ile test edilemez. Böyle bir şemsiyeyi bir entegrasyon testi yazarak, oluşturabiliriz. Resim 3 de bu yapıyı görmekteyiz.



Resim 3

Entegrasyon testlerinde birbirlerine bağımlı olan modüllerin davranışları test edilir. Örneğin müşteri bilgilerini edinmek için kullanılan CustomerDao ismindeki bir sınıfı test etmek için veri tabanına ihtiyaç duyulmaktadır. Birim testinde veri tabanı erişimi mock olarak implemente edilirken, entegrasyon testinde CustomerDao sınıfının erişebileceği bir veri tabanı oluşturulur ve bu iki modülün entegrasyonu test edilir.

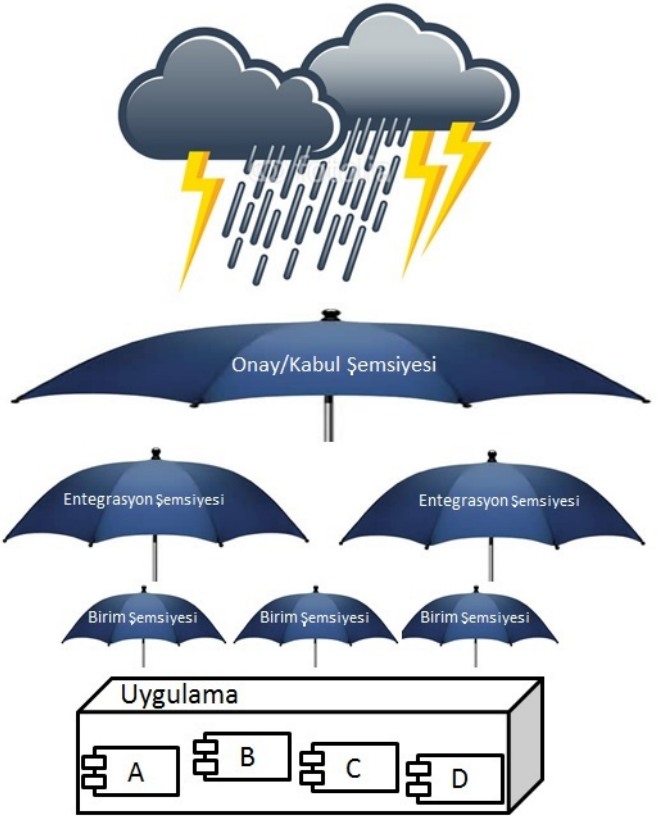
Birim testleri ile CustomerDao sınıfının entegre edilmiş bir sistemindeki işlevini test etmek mümkün değildir. Bu belirsizlik şemsiyenin büyüklüğünü ve kapsama alanını tayin etmektedir. Bu sebepten dolayı birim testleri ile tüm uygulamayı koruma altına almak mümkün değildir, çünkü birim testleri "gerçek olmayan bir ortamda" kabiliyetleri çerçevesinde izole edilmiş kod birimleriyle ilgilidirler. Gerçek bir ortamda bir veya daha fazla modülün birlikteliğini test etmek için entegrasyon testlerine ihtiyaç duyulmaktadır. Entegrasyon testleri gerçeğe yakın bir ortam oluşturubildiklerinden, kapsama alanları daha geniştir ve böylece uygulamanın daha geniş alanlarına hitap edebilirler.

Resim 3 de entegrasyon şemsiyelerinin birim şemsiyelerine nazaran daha büyük ve ağır oldukları görülmektedir. Bu sebepten dolayı taşınmaları daha zordur. Bu entegrasyon testlerinin birim testlerine nazaran daha maliyetli oldukları ve koşturulmalarının daha zaman alıcı olduğu anlamına gelmektedir.

Birim testlerinde de olduğu gibi sadece entegrasyon testleri ile tüm uygulamayı koruma altına almak mümkün değildir, çünkü entegrasyon testleri tüm sistem entegrasyonu

yerine bazı modüllerin entegrasyonuna ağırlık vermektedirler. Tüm sistem entegre edilip, test edilmediği sürece, açılan şemsiyelerin altında yer alamayan kod birimleri mutlaka olacaktır ve daha büyük şemsiyeler açılmadığı sürece, bu kod birimleri ayazda kalacaktır. Bu sistem hatalarının var olacakları anlamına gelmektedir.

Tüm uygulamayı kapsama alanına alabilecek kabiliyete sahip testler onay/kabul testleridir. Resim 4 de onay/kabul şemsiyesinin işlevi görülmektedir.



Resim 4

Onay/kabul testleri uygulamayı bir kara kutu olarak görürler ve onun kullanıcı perspektifinden test edilmesini mümkün kılarlar. Onay/kabul testleri kullanıcı ya da müşteri tarafından tanımlanan onay/kabul kriterlerini (acceptance criteria) baz alırlar ve programcılar tarafından implemente edilirler. Onay/kabul testleri ile bir uygulayı gerçek koşullarda en tepesinden en alt katmanına kadar entegre test etmek mümkündür. Topdown olarak isimlendirilen bu yaklaşım uygulamada kullanıcıların ihtiyaç duyduğu tüm uygulama davranışlarının test edilmesini mümkün kılmaktadır. Çalışır durumda olan onay/kabul testleri uygulamanın beklenen davranışı sergilediğinin kanıtıdır ve bu

sebepten dolayı müşteri ya da kullanıcı tarafından uygulamanın onaylanmasını / kabul görmesini beraberinde getirir.

Bir uygulamanın çalışır durumda olduğunun en iyi ispatı onay/kabul testlerinin varlığı ve çalışır durumda olmalarıdır. Onay/kabul testleri ile uygulamanın sahip olması gereken tüm davranış biçimleri test edilebilir. Lakin resim 4 de de görüldüğü gibi onay/kabul şemsiyenin büyüklüğü maliyetine işaret etmektedir. Bu tür testlerin implementasyonları karmaşık yapılarından dolayı zaman alıcı olabilir. Bu proje maliyetini artırıcı bir durumdur. Bunun yanı sıra onay/kabul testleri tüm sistem entegrasyonunu hedef aldıklarından, koşturulma zamanları buna orantılı olarak uzundur.

Sadece onay/kabul testlerinin uygulanması bize tüm uygulamayı kapsayacak bir koruyucu şemsiyenin açılabileceği izlenimini vermekle birlikte, şemsiyenin büyüklüğünden doğan hantallık, projenin çevikliğini aksatacaktır. Buradaki yazımda da değindiğim gibi çeviklik yazılım testleri ile bir uygulamanın istenilen şekilde yoğrulması için gerekli ortamın oluşturulmasıdır. Uygulamayı yeni müşteri gereksinimleri doğrultusunda yeniden yapılandırabilmek için yapılan değişikliklerin sebep olduğu yan etkilerin lokalizasyonu için geribildirime ihtiyaç duyulmaktadır. Bu geribildirim ne kadar hızlı alınırsa, yeniden yapılandırma o oranda ivme kazanacaktır. Onay/kabul testleri bu tür geribildirim sağlama yetisine sahiplerken, koşturulma zamanları uzun olduğundan dolayı, hızlı bir şekilde geribildirim alıp, gerekli yeniden yapılandırma işlenimi seri halde gerçekleştirmek mümkün değildir. Bunun için daha hızlı geribildirim sağlayan bir mekanizmaya ihtiyac duyulmaktadır. Bu mekanizmanın ismi birim testleridir.

Görüldüğü gibi tüm sistemi koruma altına almak ve bunun yanı sıra çeviklikten ödün vermemek için değişik türdeki testlerin kombine edilmesi gerekmektedir. Bir sonraki yazımda şemsiye modelinin nasıl implemente edilebileceğini bir örnek üzerinde sizlerle paylaşacağım.

Kıymeti Bilinmeyen Eskimiş Bilginin Kıymeti

http://www.pratikprogramci.com/2016/01/22/kiymeti-bilinmeyen-eskimis-bilginin-kiymeti/

Bilginin yarı ömrünün aylar seviyesinde ölçüldüğü bir çağdayız. Her şeyi çok hızlı tüketiyoruz. Bu bilgi için de istisnasız geçerli. Bu durumun en büyük mağdurlarından birisi yazılımcılar. Sürekli yeni bir şeyler öğrenmek zorundalar. Onlarca çatıya (framework) ve onların neredeyse her ay çıkan yeni sürümlerine hükmetmek hangi yazılımcıyı zorlamıyor ki. Zaman zaman bu konuda ipin ucunu kaçırmışlık hissinin oluşması da cabası, öyle değil mi?

Yazılımda aslında bilgi bahsettiğim şekilde eskimiyor. Bu sadece subjektif bir algı. Bu yazımda bahsettiğim gibi değişen ya da gelişen sadece soyutluk seviyesi. Yaptığımız işin birkaç temel prensibi mevcut. Bugün yazılımda geldiğimiz soyutluk seviyesi bu birkaç temel prensibin üzerine inşa edilmiş durumda. Yeni çatıları ve onların sürümlerini kanser hücreleri gibi çoğaltan yazılımcının soyutlama ihtiyacı.

Yazılımcılar herhangi yeni bir çatıyı öğreneceklerse, edindikleri kaynakların mevzubahis çatının en son sürümüne vakfedilmiş olmasına dikkat ederler. Eski sürümlerle ilgili kaynaklara itibar etmez, onları eskimiş olarak sınıflandırırlar. Ama burada ne yazık ki gözden kaçan bir nokta bulunmakta. Bu yazımda bu noktaya değinmek istiyorum. Eskimiş olarak tabir edilen bilginin içinde çok büyük hazineler yatıyor olabilir. Söz konusu yazılım ise, eskimiş bilgi yazılımın temellerini açıklayıcı türden olabilir.

Ne demek istediğimi açıklamak için elektronikten bir örnek vermek istiyorum. Bugünkü modern mikro işlemcilerin temelinde dijital elektronik yatıyor. Dijital elektronik denince benim aklıma gelen TTL (transistor to transistor logic) bazlı AND ve NOR gibi mantık kapıları (gatter). Bu yazımda TTL bazlı mantık kapılarından yaptığım sayaç yer alıyor. Bu tür entegre devreler bundan elli, altmış sene önce yapılmış ve günümüzdeki modern mikro işlemcilerin temelini oluşturuyorlar. Bu mantık kapıları ile deneyler yapmış bir yazılımcıya bir programlama dilindeki &, |, &&, ||, ^ gibi operatörleri anlatmanıza gerek yok. Ne işe yaradıklarını ve nasıl kullanıldıklarını içgüsel olarak bilir, çünkü onları daha önce entegre devre olarak elinde tutmuştur. Bunun nasıl bir devlet olduğunu eline donanıma ait hiçbir parça almadan yazılımcı olma yolunda ilerleyen müstakbel programcılara baktığımızda anlayabiliyoruz.

Bahsetmiş olduğum entegre devrelerle 8, 16 ya da 32 bitlik bir mikro işlemciyi kendiniz oluşturabilirsiniz. Buna ne gerek var diyebilirsiniz. Dijital elektronik ve mikro işlemcilerin temel çalışma prensiplerini kavramak için bunun yapılması şart. Aksi taktirde bu yazımda belirttiğim gibi mikro işlemcilerden bihaberlik söz konusu olucaktır. Mikro işlemcileri ve çalışma prensiplerini tanımayan bir yazılımcının ne kadar iyi kod yazdığı tartışılır.

Günümüzdeki modern yazılımın ve donanımın temelini oluşturan ve transistör bazlı bu entegre devreleri anlatan güncel bir tane kitap bile bulamazsınız. Bu konularda muhtemelen en son kitaplar yetmişlerin sonunda yazıldı. Bu konuda ihtisas yapma istegim benden daha yaşlı kitapları edinmeme sebep oldu.

Bugün hangi yazılımcının eline 1974 de yayımlanmış TTL cookbook kitabını verseniz, “bu ne, bunun 2016 sürümü yok mu” diye size soracaktır. İronik ama ne yazık ki gerçek: yazılımcı her daim bilginin en son sürümünü tüketme eğilimi gösteriyor. Bunun ne kadar fatal error olduğunu bu yazımda aktarmaya çalışacağım.

Bu yazıyı kaleme alma fikri node.js öğrenmek için aldığım bir kitabı okurken oluştu. Kitap 2012 yılında yayımlanmış ve node.js in 0.8 sürümünü baz alıyor. Kitaptaki örnekleri uygulayabilmek için node.js sayfasına giderek, oradaki en güncel sürümü edinmem gerekti. Bir de ne göreyim, en güncel sürüm 5.5 versiyon numarasına sahip. Aradaki farka bakar mısınız! Yerimde kim olsa hemen o kitabı çöpe atıp, node.js 5 sürümünü inceleyen bir kitap alırdı, hatta hatta o kitabı almazdı. Bir ara kendimi güncel node.js kitaplarını araştırırken bulmadım değil. Bunun başlıca sebebi, en yeninin en iyi olduğunu düşünmemiz. Lakin ben node.js öğrenmeye bu kitapla devam ediyorum. Neden mi?

Her çatının temelinde yatan bazı prensipler vardır. Bu Spring çatısında örneğin bağımlılıkların enjekte edilmesi ve bağımlılıkların tersine çevrilmesi (DIP – Dependency Injection Principle) prensibidir. Bugün güncel bir Spring kitabı alın, bahsettiğim bu iki konunun derinlemesine incelenmediğini ya da bir, iki sayfada geçiştirildiğini göreceksiniz. Bunun yerine artık onlarca modülden oluşan bir Spring çatısının yüzlece sayfada ve onlarca örnekte nasıl konfigüre edildiğini okursunuz. Bu büyük bir buzdağının su üstünde kalmış küçücük bir kütlesi ile ilgilenmek ve derinde olup, bitenlerden bihaber olmaktan başka bir şey değildir. Günümüzüm bilimum bilişim kitapları bu buzdağının görünür kısmını incelerler, çünkü soyutlama dediğimiz şey o buz dağının en tepesine çıkma eğiliminden başka bir şey değildir. Oradan suyun altında olup, bitenler hakkında fikir yürütmek imkansızdır. Buzdağının nereye gideceğini tayin eden suyun altındaki kütlesidir, suyun üstünde kalmış şapkası değil.

Şimdi node.js örneğine geri dönmek istiyorum. Node.js günümüzün en popüler çatılarından birisi. Kısa zamanda 5.5 sürümüne gelmiş olması bunun en güzel ispatı. Bugün node.js in 5.5 sürümünü anlatan bir kitap ile node.js öğrenmeye başlayan birisi, node.js in var olma nedeni olan temel prensipleri anlayamadan, node.js in ve ihtiva ettiği modüller içinde kaybolup, gidecektir. Sürüm versiyon numarası çatının hacmine işaret etmektedir. Hacim ne kadar büyükse, yazılımcının onun altında kalma ve ezime teklikesi o kadar büyüktür. Yazılımcı bu kütle altında kalmamak için ondan uzak durur. Sadece onu yüzeysel olarak anlar ve kullanır yani buz dağının tepesinden ona hükmetmeye çalışır. Buna karşın çatının ilk sürümlerini inceleyen bir kitap kendini olmayan bu hacme değil, çatının temelinde yatan ana prensiplere adayacaktır, çünkü işin başlangıcından temel prensiplerden başka anlatılabilecek bir mevzu yoktur. Bu şekilde çatının temelinde yatan temel prensipleri kavramak ve doğru yerde kullanmak mümkün olacaktır.

Bu radikal bir fikir gibi görünüyor olabilir. Lakin soyutluk denen canavardan kurtulmak ve motor kapağı altında olup, bitenleri kavramak, daha iyi bir yazılımcı olabilmek için günümüz koşullarında artık zaruri hale geldi. Biraz daha geriden sarmak ve eski sürümlerden öğrenmeye başlamak zaman alıcı bir uğraş gibi görünebilir. Lakin bu yatırımın size yol ve su olarak geri döneceğinden emin olabilirsiniz ;-)

“Neden Spring, JPA ve Diğer Çatılar ÖğrenilmeMEli” başlıklı bir yazım olmuştu. O yazımda da belirttiğim gibi “Teknolojiler gelip, geçicidir, felsefeler ise her zaman daim”.

Daha İyi Bir Programcı Olmak İçin Sınırlar Nasıl Zorlanmalı?

http://www.pratikprogramci.com/2016/01/01/daha-iyi-bir-programci-olmak-icin-sinirlar-nasil-zorlanmali/

Herhangi bir konuda daha iyi olabilmenin yolu, mevcut sınırları ve bariyerleri daha öteye taşımaktan geçer. Bu sınırlar fiziksel ya da manevi türde olabilir. Fiziksel sınırları ileriye taşımanın nasıl bir şey olduğunu ilk pist deneyimimde tekrar tecrübe etme fırsatım oldu. İlk piste çıkışımda, pisti dönme sürem 2 dakika 47 saniye ile benim için büyük bir şok oldu. Dördüncü günün son ayağında bu süreyi 2 dakika 0 saniyeye indirebildim. Dönme sürelerinin iyileşmesiyle birlikte motora ve piste olan hakimiyetimin de arttığını gözlemledim. Sürekli sınırları zorlamak hem acı verici hem de daha iyi olabilmek için verilen savaşın kazanılmasından dolayı zevkliydi.

Fiziksel sınırlarımı bu denli zorlamış olmak ve kendi çapımda bu konuda başarılı olduğumu görmek, aklıma elde ettiğim tecrübe ve bilgileri diğer ilgi alanlarıma nasıl aktarabilirim sorunu getirdi. Bu yüzden bu blog yazısını yazıyorum. En büyük ilgi alanım programcılık. Peki programcılıkta sınırlarımı nasıl zorluyorum ya da zorlamalıyım?

Aynı ekip içinde kısa bir zaman sonra herkesin kod yazma tarzı üç aşağı, beş yukarı aynı seviyeye gelir. Her yiğidin yoğurt yeme tarzı farklılık göstermekle birlikte, programcılar başkalarının kod yazma tarzlarını kopyalayarak, yaptıklarında onlara benzemeye başlarlar. Sonuç itibari ile kullanılan programlama dili programcının program yazarken kendisini ifade etme tarzını büyük oranda belirlediğinden dolayı, aynı ekip içinde yazılan kodlar birbirine benzemeye başlar. Bu şu anlama geliyor: “İstediğim kadar uğraşayım, programcı olarak bir zaman sonra motosiklet örnegindeki 2 dakika 0 saniyelik bariyerin altına inmem mümkün değil, yani programcı yetilerimi geliştiremiyorum, aynı seviyede kalıyorum.”

Bunun başlıca nedeni, kullanılan programlama dilinin programcının düşünüş ve kendini ifade etme tarzını tayin etme gücüne sahip olmasında yatmaktadır. Bir programlama dilinin kod yazmak için sunduğu yapılar bellidir. Kod parçaları bu yapıların kombinasyonlarından oluşur. Durum böyle iken bu sınırlı sayıdaki kombinasyonlar en mükemmel hallerinde programcının er ya da geç en sade çözüme erişmesini, ama daha iyi bir kombinasyon mümkün olmadığından orada takılıp kalmasına sebebiyet verirler. Programcı kullandığı dilin sınırlarına dayanmıştır, daha iyisini yapması mümkün değildir. Bu aynı zamanda onun daha iyi bir programcı olabilmesi için önündeki en büyük engeli teşkil etmektedir.

İki dakikada zor bela dönmeyi başardığım pisti, 1 dakika 36 saniyede dönen pilotlar var. Günlerce kendi dönüş zamanımı düzeltme çabası içindeyken, 1:36 lik bir zamanın nasıl mümkün olabildiği sorusunu sordum kendime. Son pist gününde 2:03 olan en iyi zamanımı 2:00 düşürmek için harcadığım eforu mantıklı bir şekilde açıklamam mümkün değil. Bu perspektiften bakıldığında 1:36 lık bir zaman erişilmez gibi görünüyor.

Peki 1:36 lık bir zaman nasıl mümkündü? Kimine göre o pilotlar slick olarak isimlendirilen mucizevi tekerlere sahiptiler. Başkalarına göre kullandıkları motorlar çok hafifdi. Elli gram tasarruf yapabilmek için motorsikletlerin akülerini bile değiştirmişlerdi. Kullandıkları grenajlar karbondu. Bunun gibi birçok efsane dinledim. Bu spora yeni başlayan birisi olarak bana her türlü hikayeyi satabilirsiniz. Ama bildiğim bir şey var. Eğer teknik olarak 1:36 mümkün ise, oraya varmanın en az bir yolu mevcut. Oraya varmak için motorun üzerine binip, konvansiyonel yöntemlerle tur atmak hiçbir zaman yeterli olmayacak. Sürekli yeni şeylerin denenmesi, sürüş tarzının adapte edilmesi, bilen birilerine sorulması, yeniliklere açık olunması ve biraz da cesur olunması gerekecek. Bu yolda birçok şeyin harmanlanması 1:36 ya giden yolun inşası için zaruri.

1:36 lık zamana erişebilmek için motor sürmenin çok değişik boyutlarına inmek gerekiyor. Bunun programlamada karşılığı birden fazla programlama diline hakimiyet. Hani usta programcılar hep “yeni programlama dilleri öğrenin” tavsiyesini verirler ya… İşte bu öyle laf olsun diye verilmiş bir tavsiye degildir. Bir programcıya verilebilecek en iyi tavsiyedir o.

İnsanların konuştukları diller ve programcıların kullandıkları programlama dilleri ortak bir özelliğe sahipler. Kullanılan dil kullanıcısının düşünüş ve kendisini ifade etme tarzını şekillendiriyor ya da sınırlıyor. George Orwell’in 1984 isimli romanında yöneticiler suni bir dilin kullanılmasını zorunlu kılıyorlar. “Yeni dil” ismini taşıyan bu dil kısıtlı bir kelime hazinesine sahip. Bu suni dilin ana amacı halkın kapsamlı ve sorgulayıcı düşünmesini engellemek. Halk örneğin baskıcı rejime karşı ayaklanamıyor, çünkü kullanılan dilin kelime hazinesinde böyle bir kelime yok.

Her dilin dünyaya bakış açısı, edebiyata yatkınlığı, yeni kelimeler türetebilme kabiliyeti ve fonetiği değişik. Dil onu oluşturan insanların dünyayı nasıl algıladıklarının bir aynası. Birden fazla dil bilen insanların da dünyaya bakış açıları ve kendilerini ifade etme ve düşünme tarzları değişiklik gösterebilmekte. Aynı şey programlama dilleri için de geçerli. Tek bir programlama diline hakim olan bir programcının o dilde kendisini ifade etme şekli sınırlıdır. Bu sınırı ona kullandığı dil koymaktadır. Programcının kendisine programlama dili tarafından vurulan bu zinciri kırabilmesi için kelime hazinesini genişletmesi yani başka programlama dilleri ve konseptleri tanıması gerekmektedir. Sadece bu şekilde programcı problem çözerken dayandığı manevi sınırları aşabilir.

Birçok programlama diline hakim programcılar geniş bir kelime hazinesine sahiptirler. Kullandıkları dilin sunduğu imkanlar yetmediginde, başka programlama dillerinden tanıdıkları konseptleri uygularlar. Bu onların hem daha iyi kod yazmalarını hem de sürekli kendilerini programcı olarak geliştirmelerini mümkün kılmaktadır.

İki dil, iki insan sözü, iki programlama dili, iki programcı olarak da anlamını korumakta. Tek bir programlama dilinde iyi bir programci olmak mümkün. Lakin daha ilerisi için birden fazla programlama diline hakimiyet şart.

Karadelikler, Soyutluk ve Yazılım

http://www.pratikprogramci.com/2015/12/04/karadelikler-soyutluk-ve-yazilim/

Günümüzde yazılım elli sene öncesi gibi if/else/while ile yapılsa da, en büyük farklılığı gelinen soyutluk seviyesi teşkil ediyor. Elli sene önceki gibi yazılımcılar artık mikro işlemcinin üzerinde işlem yapmıyorlar. Artık yazdıkları program parçaları mikro işlemcilerin anlayacağı dilden bile değiller. Bazı programcılar sadece sanal makinelar için (virtual machine) program yazar oldular. Yazılan kod mikro işlemcinin registerlerine düşene kadar beş, altı katmandan geçiyor. Bu yazılımda süre gelen soyutlamanın bir neticesi. Yazılımcılar soyutlayıp, bir üst katmana çıkarak, gidişatı daha kavranabilir ve yönetilebilir hale getirmeye çalışıyorlar. Bu bir nevi evrim. Yazılımda ilerleme sadece bu şekilde mümkün.

Soyutlama yetisinin yazılımdaki yeri şüphesiz çok önemli. Bu olmadan günümüzün karmaşık uygulamalarını makina kodunda yazmamız mümkün olmazdı. Bu bir gereklilik. Lakin bu gerekliliğin karanlık bir tarafı da yok değil. Evrende hiçbir şeyi bedelini ödemeden, sürekli kendi çıkarınıza hizmet edecek şekilde kullanamassınız. Kazancın olduğu yerde, kayıpta olmak zorunda. Bu soyutlama mekanizmaları için de geçerli. Siz soyutladıkça işiniz kolaylaşır, lakin bunun bedeli olarak neyin nasıl çalıştığını, hangi temel prensiplere dayandığını bir zaman sonra kavramanız zorlaşır.

Soyutlama işlemi bir nevi bilginin ziplenmesidir. Mevcut bilgi alınır, üzerine bir kulp takılır, bir başka kulp daha takılır, daha sonra bir kutuya konur, o kutu da başka bir kutuya konur, paketlenir, paketlenir ve daha büyük bir paketin yanına konur, bu paketlerden kocaman bir dağ oluşur. Birisi gelir ve bu dağı zipler, o kadar bilgiyi avuç içine sığacak bir paket içine sıkıştırır. Başka birisi gelerek, bu ziplenmiş pakete yeni bir kulp takar ve soyutlama döngüsü böyle devam eder gider.

Bir karadelik düşünün. Bu karadeliğin bir iğne başı büyüklüğündeki bir parçası güneş sistemimizdeki tüm gezegenlerin kütlesine eşit ağırlıkta olabilir. Karadelik soyutlama işleminin karanlık tarafını gösteren güzel bir metafor. Soyutlama süreci yazılımda sürekli karadelikler doğurur, bilginin anlaşılamayacak şekilde sıkıştırılıp, yeni bilgiye dönüştürüldüğü karadelikler.

İşte günümüzde yazılımcılar bu karadeliklerle savaş halindeler. Her yeni jenerasyon ile karadeliklerin kütleleri artıyor, çünkü soyutlama süreçleri onları devamlı besliyorlar. Bugün bu karadeliklerin iğne ucu kadar parçaları birkaç gezegen kütlesine eşitken, önümüzdeki on yıllarda bu iğne uçları galaksilerin kütlelerine eşit olacaklar. Bilgi arttıkça, soyutlama süreçleri bu bilgileri sıkıştırıp, yoğunlaştıracaklar. Hem bilgi artacak hem yazılımcının hayatındaki karadelikler hem de yazılımcının bilgisizliği.

Artan bilgiyi verimli kullanmanın tek yolu soyutlamadan geçiyor. Bunun bedeli yazılımcı olarak nasıl çalıştıklarını tam olarak anlamadığımız araç, çatı ya da metotları kullanma zorunluluğu. Bu detay bilgi eksikliği oluşan sorunları daha kısa bir zamanda çözmemizin önündeki en büyük engel.

Hem soyutlamayı çalışma verimimizi artırmak için işimize geldigi gibi kullanma isteğine sahip olmak hem de alt tarafta olup bitenleri kavrama sorumluluğunu reddetmek, birbiriyle uyuşmuyor. Kendi içimizde çelişkiye düşmemek adına bu ikisini bağdaştırmamız gerekiyor. Bunu nasıl yapabiliriz?

Gerçek şu ki bilgi sürekli artacak. Bu sebepten dolayı mantıklı bir çerçevede kullanılabilmesi için soyutlanması gerekiyor. Soyutlama süreçlerinin sonucu her zaman yoğunlaştırılmış bilgidir. Bu yoğunluk bir karadeliğin ufacık bir parçasına galaksi boyu yıldız ve gezegenlerin sığması seviyesinde olabilir. Bu yoğunluktaki bir bilgiyi kullanarak, istediğimiz bir neticeyi elde edebiliriz, ama ne olduğunu anlamayabiliriz. Ne olduğunu anlamak için bilginin içine değil, temeline göz atmamız gerekiyor. Bilginin içine girdiğimiz zaman detaylar içinde kaybolabiliriz. Temelini incelediğimiz zaman, onu kavramamız daha kolaylaşacaktır.

Soyutlamanın yan etkilerinden korunmanın, yoğunlaştırılmış bilginin temelinde yatan prensipleri anlamakla mümkün olduğunu düşünüyorum. Temelde yatan prensipleri anladığımız taktirde, bilginin nasıl kullanıldığını değil, aynı zamanda neden kullanıldığını da anlayabiliriz. Neden sorusuna cevap verebildiğimiz taktirde, bilginin doğru yerde ve doğru şekilde kullanılması mümkün hale gelmektedir.

Bu yazım yazılımda temel prensipleri anlatmaya çalışacak bir yazı serisinin başlangıcını teşkil ediyor. Bu yazı serisi “Temelinde Yatan Prensipler” başlığı altında değişik konulara ışık tutmaya çalışacak. Bu serinin ilk yazısı “Mikroservis Mimarilerinin Temelinde Yatan Prensipler” başlığını taşıyor. Bu yazıyı yakında beğeninize sunacağım.

Paralel Evrenlerin Programcıları

http://www.pratikprogramci.com/2015/10/20/paralel-evrenlerin-programcilari/

Bir önceki yazımda frontend ve backend programcılığı arasındaki ayrımın kalkacağından bahsetmiştim. Kendi programcılık kariyerim için bu yazımı bir dönüm noktası olarak görüyorum. Yirmi birinci yüzyılın yazılımcılığında bir dönüm noktasına gelmiş bulunuyoruz. Bu yazımda bunun sebebi açıklamaya çalışağım.

Java'nın ilk günlerinden beri programcı olarak bu dille ekmek paramı kazanıyorum. Son on beş yılda birçok Java çatısı (framework) ile çalıştım. Kendimi daha çok backend tarafında gören bir yazılımcıydım ta ki yukarıda linkini verdiğim yazıyı yazana kadar. Bir backend yazılımcısı değil, kendi programcı evreninde yaşayan bir programcıymışım ve paralel programcı evrenlerinin farkında bile değilmişim. Ne oldu da birden yazılımcı evreninin merkezinde olmadığımı, başka evrenlerinde var olduğunu keşfettim? Anlatmaya çalışayım.

Ağaçlara bakmaktan ormanı göremez hale gelmişiz…

Yıllardır o çatı senin, bu çatı benim diye yazılım yapıyoruz. Bugün bir kurumsal projeye göz atın, içinde en az on beş tane çatı bulabilirsiniz. Artık çatı olmadan yazılım yapılamıyor. Ne yazık ki geldiğimiz durum bu. Çatı yok, proje yok! Bundan daha vahim bir durum olabilir mi? Ne yedügü belirsiz bir ton kodu projene çekiyorsun ve onların üzerine birşeyler inşa etmeye çalışıyorsun. Hasbelkader çalışıyor! Çoğu zaman çalışmıyor, neden çalışmıyor diye kafandaki saçları yoluyorsun. Çatı kullanımı teknik borçlanma türevlerinden birisidir, er ya da geç tasarlanmadığı bir senaryo ile karşılaşıldığında, workaroundlar kaçınılmazdır. Her workaround bir teknik borçtur. Neyse, konumuzdan fazla sapmayalım…

Ben çatılarla büyümüş bir yazılımcı jenerasyonuna dahilim. Kullandığım diller statik veri tipli ve derlenen diller. Kendimi en çok böyle dillerde güvende hissediyorum. Yazdığım kod bana daha bi deterministik geliyor. Neden derleyici varken bir değişkene doğru veri tipinde bir değer atadım mı diye kendim kontrol edeyim ki! Ne gerek var…

Yazdığım kodun derlenmesi için belli bir zamana ihtiyacım var. Bunun üzerine çatıların çalışma şekilleri, konfigürasyonları ve kodumu test etmek için çalışır hale getirirken geçen zaman eklenince, en ufak bir değişikliğin bile derlenen dillerde ne kadar büyük zaman kaybına açabileceğini düşünebilirsiniz. İki binli yılların başında EJB (Enterprise Java Bean) uygulamalarını çalışır hale getirmek için onlarca dakika beklemek zorunda kaldığım günleri unutmadım. Bunları neden yazıyorum?

Çalıştığımız projelerde artık sorgulamadan belli kalıpları takip eder olmuşuz. Yeni bir proje mi var, hemen Spring, Hibernate, o çatı, bu çatı… Bu kalıpların dışına çıkamıyoruz, hep aynı şekilde hareket ediyoruz, çünkü içinde bulunduğumuz evren bu araçlardan oluşuyor. Hep bunlarla çalışmışız, bir mazohist gibi acı çekmeye alışmışız, ama onlardan vazgeçmemiz söz konusu bile olamaz, öyle değil mi?

Tipik bir Spring uygulamasını alın. Basit bir konfigürasyon yapacağım derseniz, elli dereden su getirmek gerekiyor. Peki Spring nasıl oluştu? Spring'in tek varoluş nedeni, Java EJB teknolojisinin çok hantal olmasıydı. Spring çatısını geliştirenler ortaya çıktılar ve "arkadaşlar bırakın şu saçma, sapan EJB, MJB leri, bizde uygulama sunucusu olmadan bile çalışabilen yeni bir çatı var, yazılım yapma hızınızı çok artıracak, bu çatıyı lütfen deneyin" dediler. Bizde sazan gibi atladık, çünkü başka bir alternatif yoktu.

Daha hızlı yazılım yapma iddasi ile ortaya çıkmış olan bu çatıya şimdi bir bakın. Ne hallere gelmiş durumda! Grails bünyesindeki spring/resources.groovy içinde Spring Bean DSL bazlı bir cronjob tanımlaması ihtiyacı duyan bir task için bir saat konfigürasyonla ugraşmak zorunda kaldım. "Aga böyle olmadı" deyip, durdu Grails sunucusu. Kafayı sıyırtacak seviyeye getirdi kısacası. Şu içine düştüğümüz duruma bir bakın hele.

Bize vaad edilen hızlı yazılım geliştirme araç, gereçleri idi. Bir çatının asıl vazifesi, programcısını angaryadan kurtarmaktır. Kullandığımız çatılar angaryadan kurtarmıyor değiller. Ama oradan kazandığımız zamanı uygulamaları çalışır hale getireceğiz, kodu derleyeceğiz, konfigüre edecegiz diye harcıyoruz. Hızlı yazılım geliştirmek için yanlış araçları kullanıyoruz.

Bu cümbüşün içinde birileri gelip "AngularJS, NodeJS gibi teknolojileri denesenize abi, çok daha hızlı uygulama geliştirirsiniz" diyor. Bıyık altından pis, pis gülüyoruz. "O da neymiş yahuuu, biz burada çok ciddi enterprise, menterprise teknolojileri ile uygulama geliştiriyoruz" diyoruz. Yalan mı? Ama o da ne, paralel bir evren mi gelişiyor orada yoksa! Bak, bak hele… bize hiç haber etmemişler. NodeJS de neymiş ki?

Adam bize neden haber etsin ki. Bir defa senin gibi teknoloji içine çakılıp, kalmamış. Tek derdi hızlı bir şekilde müşteri isteğini koda döküp, çalışır hale getirmek. Senin gibi teknoloji fanatiği değil. İki nokta arasındaki mesafenin bir çizgi olduğunu biliyor. Bizim gibi iki noktayı otuz beş köşe üzerinden birleştirmiyor. He mi gelip sana anlattığında, nazarı dikkate alacak mısın? Almayacağının en büyük nedenini söyleyeyim mi? Kullandığın araçlar ve teknolojilerin üstünlüğüne inanmışsın. Başkaları daha ileri teknolojiler oluşturuyor ya da kullanıyor olamaz! Burada sen diye hitap ettiğim kendi şahsımdır, üzerinize alınmayın.

Bir tarafta köklü ve gelenekli proje geliştirme yöntem ve teknolojileri, diğer tarafta hızlı ürün geliştirmek için oluşmaya başlayan teknolojiler. Bir tarafta teknolojiden gözü körleşmiş yazılımcılar, diğer tarafta yazılımın daha kolay yapılabileceğine inanan ve ihtiyaç duydukları araçları oluşturan yazılımcılar. Bir tarafta backend, frontend ayrılmalı diyen yazılımcılar, diğer tarafta aynı teknoloji ile backend ide yazarım diyen pragmatik frontend yazılımcıları. İki ayrı evren…

Ne vahim durumda olduğumuzu gösterir başka bir örnek vermek istiyorum. IBM mainframe üzerinde çalışan Cobol programcılarına neden kolay kolay nesneye yönelik programlama tekniklerinin öğretilemeyeceğine değinelim. Cobol dilinde funksiyonlar bile yok. Cobol programcısının network ya da desktop tecrübesi yoktur ya da çok azdır. Cobol programları structured programming teknikleri ile yazılır. Bir Cobol programcısı

işletme mantığı dendiğinge gözünde birçok şeyi canlandırabilirken, kalıtım, o, bu demeye başlandığında, söylenenler bir kulagından girer, diğer kulağından çıkar, çünkü görüş alanında son otuz yıldır kullandığı teknikler, yöntemler ve teknolojiler vardır, OOP yoktur.

Kurumsal projelerde çalışan biz yazılımcılar da aynı duruma düşmüş durumdayız. Birileri bize gelip, olup, bitenlerden bahsetmeye çalışıyor, lakin bunlar bir kulağımızda giriyor, diğer kulağımızdan çıkıyor, çünkü gidişatı kavrayamıyoruz, sadece kullandığımız çatılara odaklanmış durumdayız. Ağaçlara bakmaktan, ormanı göremiyoruz. Sadece bir evrenin olabileceğine inanıyoruz. Paralel evrenlerin farkında değiliz. Ama içinde yaşadığımız evren yok olma tehlikesi altında.

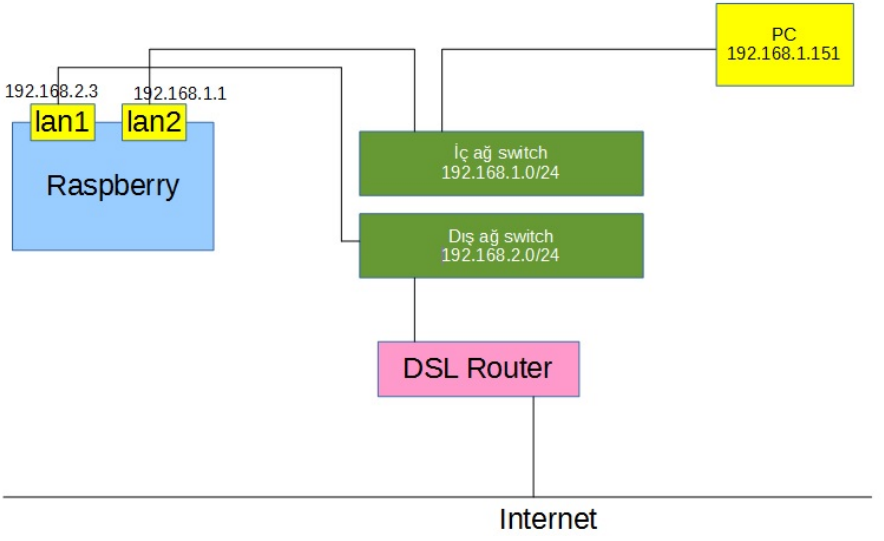
Biz kendi evrenimizde frontend, backend programcılığının ayrımını tartışa duralım, fullstack developer bize çok radikal bir fikir gibi gelsin, her şeyi çatıların üzerine inşa etmeye devam edelim, oluşmakta olan paralel evrende adamlar hızlı yazılım geliştirme filozofisini frontend den alıp, backende taşımak üzereler. Bizim 5 günde yaptığımız işi 2 saat içinde yapacak seviyeye geldiklerinde, bizim evrendeki işsizlik oranı çok yükselecek gibi.

Maksadım içinize korku salmak değildi. Sadece yorumlamaya çalışmaya çalıştığım gidişati sizlerle paylaşmaya çalışıyorum. Daha önce kendisine gönderilen iş ilanlarında Javascript ibaresini gördüğünde burun kıvıran bir yazılımcı olarak, kafamızı kaldırıp, etrafımızda olup, bitenleri takip etmemiz, onlara kulak vermemiz gerektiği düşüncesindeyim. NodeJS, AngularJS, BackboneJS ve ReactJS yolculuğun nereye gittiğine işaret ediyorlar. Daha şimdiden bizim bildiğimiz backend teknolojilerine gerek kalmadan Javascript ile katmanlı mimariler oluşturmak mümkün. Javascript artık öğrenilmesi mecburi bir dil haline geldi. Javascript oluşmakta olan paralel evrenlerin sadece bir tanesinde kullanılan bir dil. Daha böyle birçok paralel evren oluşmadığı ne malum.

Raspberry PI Router Olarak Nasıl Kullanılır?

http://smarthomeprogrammer.com/tr/2015/11/01/raspberry-pi-router-olarak-nasil-kullanilir/

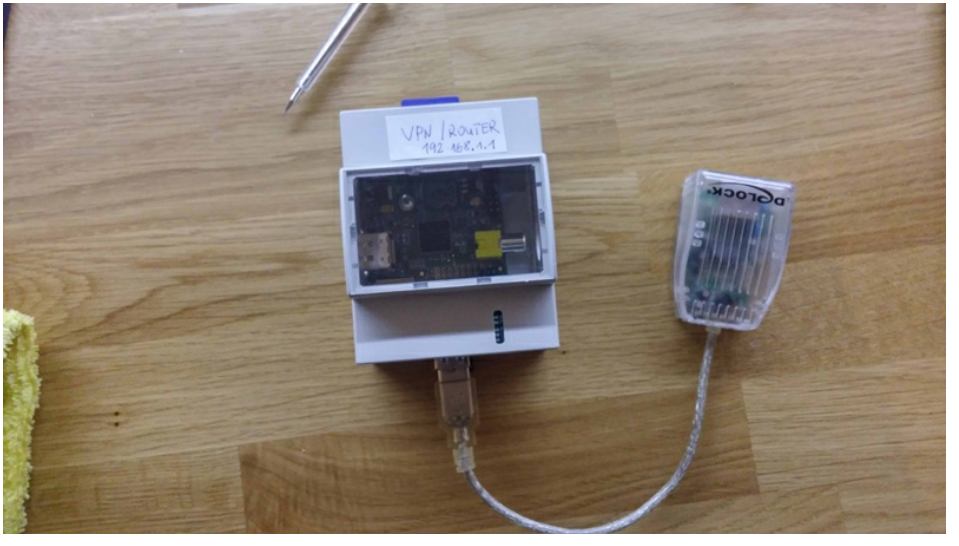
Bir resim bin kelimeye bedeldir deyimiyle ne yapmak istedigimi bir resim ile aciklayarak, baslamak istiyorum. Asagida evimdeki ag yapisini görmektesiniz.



Resimdeki Raspberry mini bilgisayari göz ardi edecek olursak, evimde birbirinden bagimsiz iki bilgisayar agi bulunmakta. Bu aglari ic ag ve dis ag olarak isimlendiriyorum. Ic ag üzerinden ev otomasyonu bilesenleri birbirleriyle iletisim icindeler. Dis agi internete erismek icin kullaniyorum. Bu iki agin birbirlerinden fiziksel olarak ayrilmalari benim icin mühim, cünkü disardan kimsenin ic aga eriserek, ev otomasyonuna müdehale etmesini istemiyorum. Bu durumda iki ag birbirlerinden izole bir sekilde hayatlarini sürdürüyorlar.

Ev disinda iken VPN üzerinden ev otomasyonu uygulamasini erisebilmek icin ic aga giris saglamam gerekiyor. Bu amacla iki agi birbirine baglayacak ve router vazifesi görecek bir bilesene ihtiyacim var. Bu görevi Raspberry üstleniyor. Bu yazimda kullandigim Raspberry nin router olarak nasil konfigüre edildigini sizlerle paylasmak istiyorum.

Bir bilgisayarin iki ag arasinda router olarak görev yapabilmesi icin iki agdada ayagi olmasi gerekiyor. Bilindigi üzere Raspberry sadece bir LAN girisine sahip. Ikinci bir LAN girisi olusturmak icin Delock USB LAN adapter modülünü kullandim. Bu adapter Raspian tarafindan otomatik olarak taniniyor ve sisteme ekleniyor.



Kullandigi LAN girisleri su sekilde:

```
pi@raspberrypi ~ $ ifconfig
eth0      Link encap:Ethernet  HWaddr b8:27:eb:b8:f7:b0
          inet addr:192.168.2.3  Bcast:192.168.2.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1024520 errors:0 dropped:12 overruns:0 frame:0
          TX packets:517855 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1483958463 (1.3 GiB)  TX bytes:35039591 (33.4 MiB)

eth1      Link encap:Ethernet  HWaddr 00:13:3b:12:0e:c6
          inet addr:192.168.1.1  Bcast:192.168.1.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:520568 errors:0 dropped:16 overruns:0 frame:0
          TX packets:1020428 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:26963211 (25.7 MiB)  TX bytes:1498010285 (1.3 GiB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:47 errors:0 dropped:0 overruns:0 frame:0
          TX packets:47 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:4068 (3.9 KiB)  TX bytes:4068 (3.9 KiB)
```

Raspberryi router haline getirebilmek icin önce gerekli kurumlari gerceklestiriyorum:

```
sudo apt-get update && sudo apt-get install ca-certificates
```

Ikici LAN girisini su sekilde /etc/network/interfaces dosyasinda olusturdum:

```
pi@raspberrypi ~ $ cat /etc/network/interfaces
auto lo
iface lo inet loopback

auto eth0
allow-hotplug eth0
iface eth0 inet manual

#USB LAN eth1
auto eth1
iface eth1 inet static
address 192.168.1.1
netmask 255.255.255.0
network 192.168.1.0
broadcast 192.168.1.255
gateway 192.168.2.1
pi@raspberrypi ~ $
```

Yeni ag girisinin aktif hale gelmesi icin yapilmasi gereken islem su sekilde:

```
sudo /etc/init.d/networking restart
```

Benim istegim, ic aga baglanan cihazlarin dis ag üzerinden internete baglanabilmesi yönünde. Örnegin bu yaziyi yazdigim bilgisayar Raspberryi router olarak kullanarak, internete erisebiliyor. Ag ayarlari su sekilde:

```
C:\Users\acar>ipconfig
Windows-IP-Konfiguration
Ethernet-Adapter LAN-Verbindung:

   Verbindungsspezifisches DNS-Suffix: local
   Verbindungslokale IPv6-Adresse  . : fe80::e08a:4c9d:28d5:2b1c%11
   IPv4-Adresse  . . . . . . . . . . : 192.168.1.151
   Subnetzmaske  . . . . . . . . . . : 255.255.255.0
   Standardgateway . . . . . . . . . : 192.168.1.1
```

Görüldügü gibi kullandigim bilgisayarin IP adresi 192.168.1.151 ve default gateway olarak 192.168.1.1 tanimli. 192.168.1.1 Raspberry nin IP adresi. Raspberry bir bacagi ile ic agda oldugu icin bu agin da bir parcasi. Ic aga baglanip, internete erismek isteyen bilgisayarlarin bir DHCP sunucusundan IP adresleri almalari gerekiyor. Ic ag icin DHCP servisini de Raspberry üstleniyor. Bu yüzden bir sonraki adimda Raspberry üzerinde bir DHCP sunucusu olusturmamiz gerekiyor. Sunucuyu söyle kuruyoruz:

```
sudo apt-get install isc-dhcp-server
```

Simdi /etc/dhcp/dhcpd.conf icinde asagidaki ayarlari yapiyorum:

```
# If this DHCP server is the official DHCP server for the local
# network, the authoritative directive should be uncommented.
authoritative;

subnet 192.168.1.0 netmask 255.255.255.0 {
range 192.168.1.150 192.168.1.250;
option broadcast-address 192.168.1.255;
option routers 192.168.1.1;
default-lease-time 600;
max-lease-time 7200;
option domain-name "local";
option domain-name-servers 8.8.8.8, 8.8.4.4;
}
```

DHCP sunucusu aga baglanan yeni bilgisayarlara IP adresi atama görevine sahip. Bu IP adreslerinin hangi aga ait oldugu bilgisini subnet ile tanimliyorum. Range ile hangi IP adreslerinin bilgisayarlara atanabilecegi netlestiriliyor. Buna göre yeni bilgsayarlar 150 ile 250 arasinda bir IP adresi alabilirler.

Bu ayarlarin ardindan DHCP sunucusunu restart ediyorum:

```
sudo /etc/init.d/isc-dhcp-server restart
```

Eger hata olmazsa ekranda asagidaki satirlarin görünmesi kuvvetle muhtemel:

```
[ ok ] Stopping ISC DHCP server: dhcpd.
[ ok ] Starting ISC DHCP server: dhcpd.
```

Simdi sirada iki ag arasinda IP forwarding özelliginin aktive edilmesi var. Sadece bu sekilde Raspberry iki ag bacagi arasinda paketlerin gecisine izin vermektedir.

```
sudo echo 1 > /proc/sys/net/ipv4/ip_forward
```

Bu tanimlamanin kalici olmasi icin /etc/sysctl.conf dosyasinda asagidaki satirin yer almasi

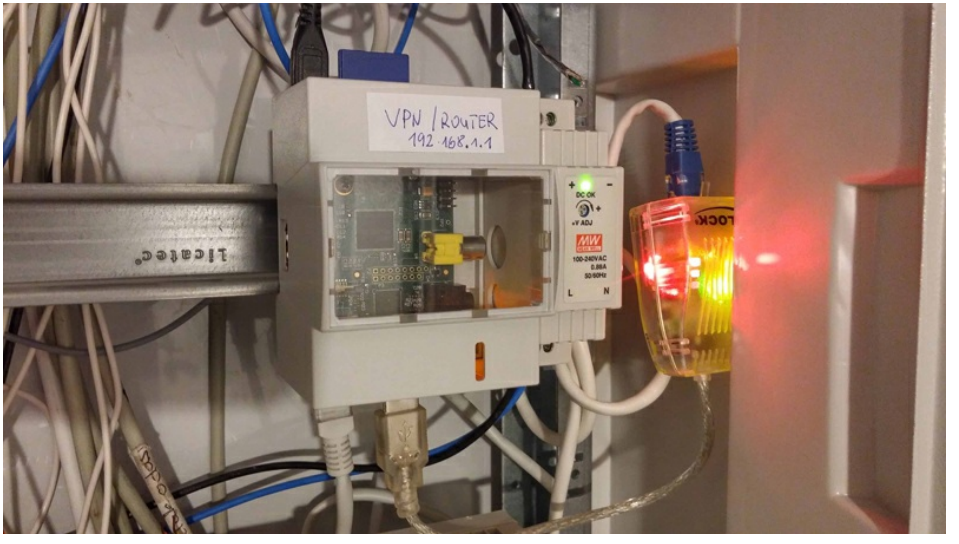
gerekiyor:

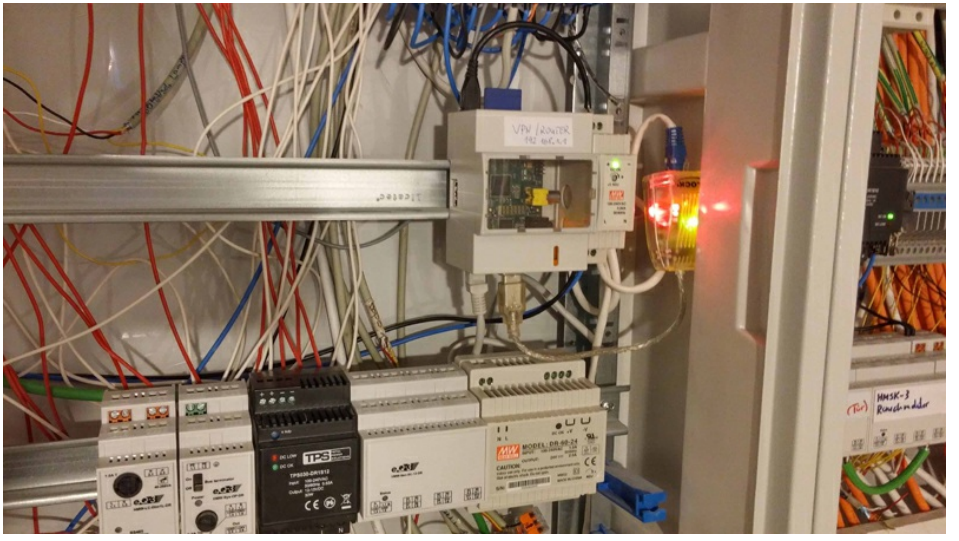
```
# Uncomment the next line to enable packet forwarding for IPv4
net.ipv4.ip_forward=1
```

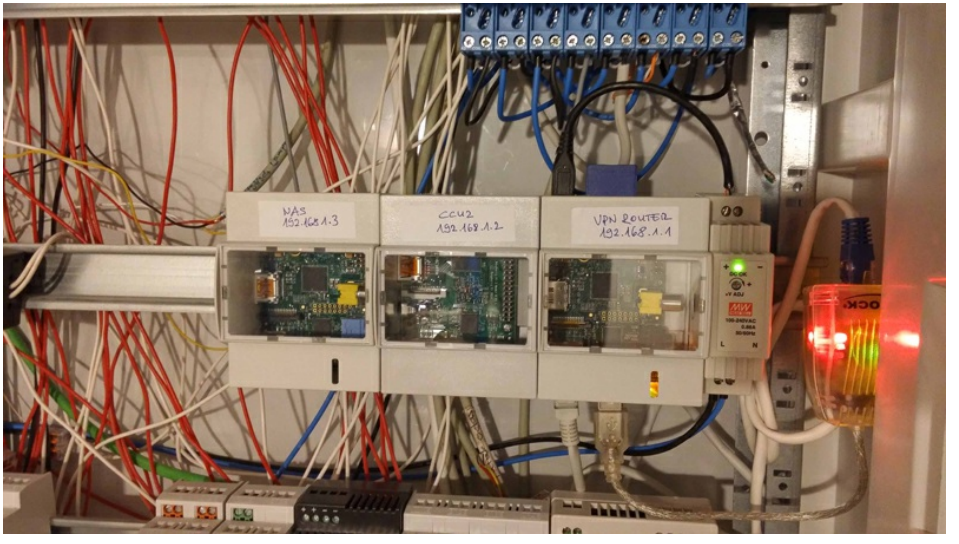
Ic agda bulunan bilgisayarlarin internete cikabilmeleri icin NAT (Network Address Translation) özelliginin aktivlestirilmesi gerekiyor.

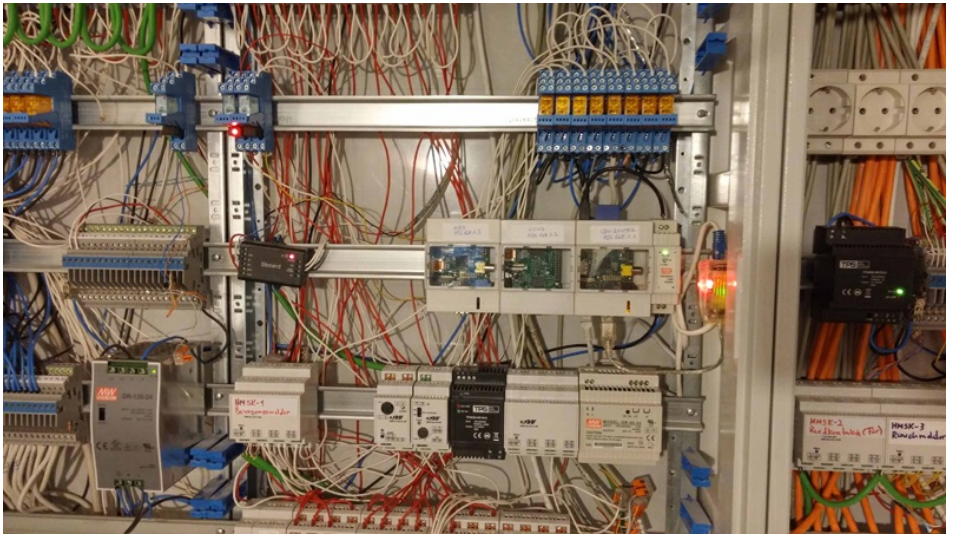
```
iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
```

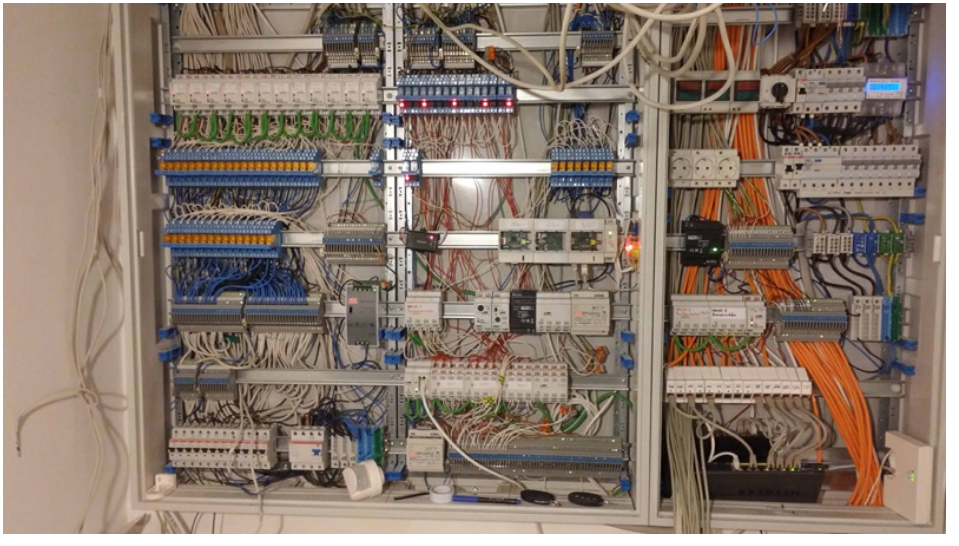
Bu islemlerin ardindan ic ag switch bilesenine baglanan her yeni bilgisayar Raspberry den bir IP adrese alarak, internete erisebilecektir. Raspberry mini bilgisayarlari router yani sira NAS ve ev otomasyonunu yöneten sunucu olarak da kullaniyorum. Bir sonraki yazimda VPN ile disardan ic aga erisimin nasil yapilabilecegini aktarmaya calisacagim.

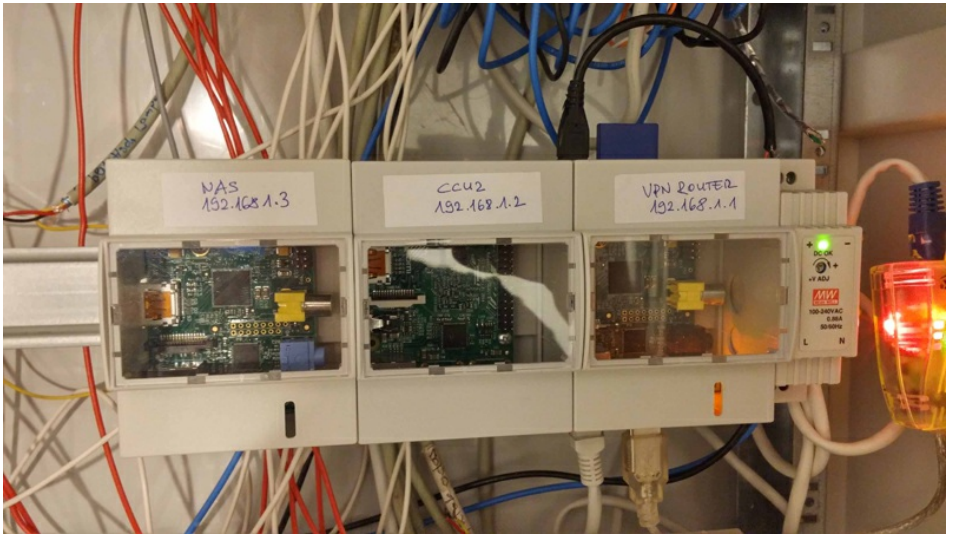





NAS
192.168.1.3
CCU2
192.168.1.2
VPN ROUTER
192.168.1.1






NAS
192.168.1.3
CCU2
192.168.1.2
VPN ROUTER
192.168.1.1

Neden Frontend ve Backend Programcısı Tarihe Karışıyor

http://www.pratikprogramci.com/2015/10/15/neden-frontend-ve-backend-programcisi-tarihe-karisiyor/

Öncelike backend/frontend nedir, backend/frontend programcısı ne yapar sorularına cevap vermeye çalışarak başlamak istiyorum.

Bir tiyatro oyununu düşünelim. Sahnede gördüklerimiz uygulamanın ön yüzü yani frontend, sahne arkasında olup bitenlerin hepsi arka taraf yani backend dir. Katmanlı mimarilerde her uygulamanın bir arka tarafı, bir ya da daha fazla ön tarafı olur. Uygulamanın görsel tarafında çalışan programcılara frontend, sahne arkasında olup, bitenlerden sorumlu olan programcılara backend programcısı denir.

Uygulamanın görsel kısmını geliştirmek için bu konuda uzmanlaşmış frontend yazılımcıları ile çalışmanız gerekiyor. Bu yüzden her ekip içinde frontend/backend programcısı diye ayrım yapılır. Bunun başlıca sebebi, her iki katmanda ayrı teknoloji ve çatıların (framework) kullanılması ve değişik türde programcı yeteneklerine ihtiyaç duyulmasıdır. Backend karmaşık işletme mantığına ev sahipliği yaparken, frontend daha ziyade son kullanıcıların uygulama ile olan interaksiyonlarını mümkün kılan katmandır.

Yapı itibari ile backend daha karmaşık (complex) gibi görünse de, bu karmaşa uygulamanın görsel katmanında da mevcut. Aradaki tek farklılık, kısa bir zaman öncesine kadar frontend programcılarının bu karmaşa ile baş edebilmek için gerekli araç, gereç ve tekniğe sahip olmamalarıydı. Ama bu konuda bir dönüm noktasına gelmiş bulunuyoruz. Oyunun kuralları değişiyor. Bunun hem frontend hem de backend programcıları farkında mı acaba? Bu sorunun cevabını bulmaya çalışalım.

Backend programcıları ne yapar? Backend programcıları kendilerini yazılımcı ve yazılım mimari olarak görürler. Uygulamanın tüm mimarisinden yazılımcı olarak onlar sorumludurlar ya da bundan sorumlu olmak isterler. Frontend kısmını pek kaleye almadan, hayatlarını sürdürüp giderler. Tipik bir backend programcısına Javascript kodu yazdıramazsınız. O sadece backend bilir, orada kalır, kendini orada rahat hisseder. Buna karşın bir backend programcısının kelime hazinesinde tasarım şablonları (design patterns), tasarım prensipleri, bağımlılıkların enjekte edilmesi (dependency injection), katmanlı mimari, test güdümlü yazılım, esnek bağ, nesneye yönelik programlama, refactoring, tekrar kullanılabilirlik ve temiz kod gibi terimler bulabilirsiniz. İdeal bir backend programcısı çevik yazılım metotları yardımı ile bakımı ve geliştirilmesi kolay bir backend oluşturabilir. Uygulama ne kadar karmaşık olursa olsun, backend programcısı bu karmaşa ile baş edebilecek yetileri sahiptir (ya da olmalıdır, aksi taktirde backend programcısı ünvanını hak etmez). Ya frontend programcıları nasıl çalışıyor?

Bu sorunun cevabını vermeden önce şunu belirtmek isterim. Maksadım frontend programcılarını küçümsemek değil. Yanlış anlaşılmak istemem. Onlar da benim gibi yazılımcılar. Ben de birçok projede frontend programcısı olarak çalıştım. Neticede aynı

meslekten olup, uzmanlık alanları değişik olan programcılar bir araya gelerek, müşterinin gereksinimlerini tatmin etmesi beklenen bir uygulama ortaya koyuyorlar. Lakin bir backend programcısı olarak son 15 yılda öğrendiğim ve uyguladığım yöntem ve tekniklerin yeni yeni frontend programcıları tarafından keşfedildiklerine şahit oluyorum. Bugün frontend yazılımcılığına baktığımızda, MVC (Model View Controller) çatılarının 'state of the art' olduklarını görmekteyiz. Bundan önce ne vardı? Bilimum modern yazılım teknikleri göz ardı edilerek oluşturulmuş, prosedürlerden oluşan Javascript ambarları! Aynı şey PHP ya da benzeri çatılarla geliştirilmiş uygulamalar için de geçerli. Hangi çatı olursa, olsun bir controller yapısını açıp, içine baktığınızda, tüm işletme mantığının orada olduğunu görebilirsiniz. JSP sayfalarına gömülmüş JDBC kodları görmüşlüğüm bile mevcut. Frontend programcılığı günümüzde çok iptidai yöntemlerle yapılıyor. Geniş çaplı bir yazılım metodolojisinin uygulandığıı görmek mümkün değil. Sadece MVC tasarım şablonunu kullanarak, modern bir frontend yazılımı yapıyorum demek doğru olmaz. Yazılımın bilinmesi ve uygulanması gereken birçok boyutu bulunmaktadır. Backend programcıları bunun çok iyi bilincindeler, ama uygulayıp uygulamadıklari tartışılır.

Uygulamanın görsel katmanının frontend programcıları tarafından sadece bir katman olarak değil, orada katmanlı bir mimarinin uygulanabileceği bir saha olarak görülmesi zaruri. Aksi taktirde frontend katmanında oluşan karmaşa ile baş etmeleri imkansız. Bunu nasıl yapabilirler?

Klasik katmanlı bir mimaride görsel, servis ve veri katmanı olarak en az üç katman mevcuttur. Backend genellikle servis ve veri katmanını altında bulunduran bir REST, Webservis ya da EJB katmanıyla kullanıma açılır. Bu yapı oluşturulurken birçok yazılım tekniği, prensibi ve metodolojisinden faydalanılır. Bunların hepsinin frontend programcılığında da öne çıkan öğeler olması gerekmektedir. Backend programcısı yazılım yapmak için gereken hangi bilgiye sahipse, frontend programcısının da aynı bilgiye sahip olması gerekmektedir. Backend yazılım tekniklerini frontend bünyesinde uygulamaya çalışan AngularJS gibi çatılar oluşmaya başladı. Peki frontend programcıları bu geçişe hazırlar mı?

Şimdi bunun tam tersini irdeleyelim. Eğer yakın zamanda frontend yazılımcılığını backend yazılımcılığında uygulanan teknik ve yöntemlerle yapmayı mümkün kılan seviyeye gelecek isek, frontend programcılarına ihtiyaç var mı? Bu durumda ihtiyaç ortadan kalktı gibi görünüyor. Peki bu boşluğu kim doldurmak zorunda? Bu boşluğu backend programcısı doldurmak zorunda! Peki backend programcısı bu geçişe hazır mı?

Yıllarca Javascript ve türevlerine sıcak bakmayan bir backend yazılımcısı olarak, bir backend programramcısını frontend programcısına dönüştüremeyeceğinizin garantisini verebilirim. Backend programcısı frontend bünyesinde uygulanan programcılığı programcılık olarak görmez. Bu onun kibirli olduğunun göstergesi değildir. Bu standartlarda frontend programcılığı backend programcılığının 15 sene gerisinden gelmektedir. Backend programcının eline frontend yazılımı yapabilmesi için gerekli araç, gereçleri vermeniz gerekiyor. Bu araçlar oluşmaya başladı. Örnek mi?

Günümüzde kullanılan web tarayıcıları EcmaScript 5 spesifikasyonunu destekliyor. EsmaScript Javascript programlama dilini oluşturmak için kullanılan dil spesifikasyonudur. EsmaScript 5 bünyesinde nesneye yönelik programlama konseptleri yok. Bu yüzden JavaScript bünyesinde funksiyonlar ya da değişken referansları yardımı ile sınıf vari bir şeyler tanımlayabiliyorsunuz. Örneğin şöyle:

```
var car = {
    brand: "ford",
    model: "fiesta",

    getCarInfo: function () {
        return this.brand + " - " + this.model;
    }

// ya da

function car (brand, model) {
    this.brand = brand;
    this.model = model;
    this.getCarInfo = getInfo;
}

function getInfo() {
    return this.brand + " - " + this.model;
}
```

EcmaScript 5 bazlı Javascript ne sınıf, ne interface, ne soyutluk ne de kalıtım tanıyor. Prototype değişkeni üzerinden kalıtım oluşturmaya çalışmayı kalıtım olarak görmüyorum. Böyle teknikler kodun daha da kötü okunur hale gelmesini sağlıyor.

Şimdi EcmaScript 6 yı baz alan TypeScript örneğine bir göz atalım:

```
class Tire{
}

class Vehicle {
    private tires:Tire;
}

class Car extends Vehicle {

    private brand:string;
    private model:string;

    constructor(brand:brand, model:model){
        super();
    }

    public getBrand():string {
        return this.brand;
    }

    public getModel():string {
        return this.model;
    }
}

interface Driver{
    isDriving:bool;
    startDriving:void;
    stopDriving:void;
}

class Person implements Driver{

    public isDriving:bool;

    constructor() {
        this.isDriving = false;
    }

    public startDriving():void {
        console.log("now driving....");
        this.isDriving = true;
    }

    public stopDriving():void {
        console.log("stopping the car....");
        this.isDriving = false;
    }
}
```

TypeScript in safkan nesneye yönelik programlama dili olduğunu görüyoruz. Bu altyapıyı kullanarak gösterim (model, view, controller), servis ve veri katmanını sınıflardan yararlanarak oluşturabiliriz. Bunu yapmamızı mümkün kılan çatıların başında AngularJS geliyor. Yazılımda karmaşayı önlemenin tek yolu böl ve yönet tekniğinin uygulanmasıdır.

Burada modüler yapılar ve tek sorumluluk prensibi öne çıkıyor.

AngularJS bir Javascript MVC çatısı. 1.x sürümü EcmaScript 5 destekleyen web tarayıcılarında çalışıyor. AngularJS 2.x TypeScript bazlı olacak. AngularJS bünyesinde MVC konsepti yanı sıra işletme mantığının yer aldığı servisler, bağımlılıkların enjekte edilmesi (dependency injection) ve modüler yapılar mevcut. AngularJS in temelinde yatan asıl arzunun frontend yazılımcılığına backend yazılımcılığından tanıdığımız konseptleri getirmek olduğunu görebiliyoruz.

Bunun yanı sıra işveren de frontend bölümünde modern yazılım tekniklerinin uygulanması gerekliliğinin farkına varmış durumda. Uygulamalar için yapılan yatırımlar uygulamaların geliştirilebilirliklerinin yüksek tutulması sayesinde korunabilir. Artan müşteri gereksinimleri ile büyük bir karmaşa ve kargaşa içinde batıp gitmiş bir frontend katmanında yapılan yatırımın korunabileceği söylenemez, çünkü bu durumda uygulamanın ileri versiyonlarının oluşturulamaması tehlikesi mevcuttur. Bu sebepten dolayı frontend için modern yazılım tekniklerinin uygulanması gerekliliği doğmaktadır.

Benim görüşüm kısa bir zaman sonra backend, frontend programcısı ayrımının ortadan kalkacağı yönünde. Frontend programcıları ileri yazılım tekniklerini uygulamaya başladıkları andan itibaren, backend programcılığı için gerekli yazılımcı yetilerini kazanmaya başlayacaklar. Frontend programcısı için bu bilgi çıtasının yükselmeye başladığı anlamına geliyor. Bu duruma ayak uydurmak zorundalar, aksi taktirde onları büyük bir tehlike bekliyor: backend programcıları onların yerlerini almak zorundalar!

Aynı durum backend programcıları için de geçerli. Frontend için ileri seviye teknikler uygulanmaya başlandığında, işverenler bu tekniklere sahip programcılarla çalışmak isteyecekler. Bu durumda sadece backend programcılığı yapabiliyorum demek yeterli olmayacaktır. Teknoloji olarak birbirine yakınlaşmış backend ve frontend yazılımcılığı için her iki sahada aktif olabilecek yazılımcılara ihtiyaç duyulmaktadır. Tek bir tarafa hakimiyet yeterli olmayacaktır. Gidişat artık terim olarak pek sevmesemde fullstack development yönündedir. Bu frontend programcılığını ortadan kaldıran bir akım olacak. Programcı olarak hayatlarını sürdürmek istiyorlar ise, frontend programcıları arkaya, backend programcıları da öne doğru uzanmak zorundalar. Piyasa bunu istiyor gibi görünüyor.

Yazılım sektöründe yine taşlar yerinden oynamaya, bilinen şeyler eskimeye ve yeniliklere olan gebelik artmaya başladı. Marifet bu gidişata ayak uydurabilmekte. Ben olsam yavaş yavaş Javascipt 6 öğrenmeye başlardım :)

Yazılımcıların Performansı Nasıl Ölçülür?

[http://www.pratikprogramci.com/2015/09/28/yazilimcilarin-performansi-nasil-olculur/] 3/4/2016 8:47:13 PM (http://www.pratikprogramci.com/2015/09/28/yazilimcilarin-performansi-nasil-olculur/)

BTSoru.com da yazılımcıların performansı nasıl ölçülür şeklinde bir soru sorulmuş. Bu konudaki naçizane fikirlerimi bu blog yazımda sizlerle paylaşmak istedim. Bu yazımda performans ile verimliliği eş tutuyorum. İyi bir performans verimli bir çalışma anlamına gelmektedir.

Yazılımcının iş gücü olarak ortaya koyduklarının tümünü performansı olarak tanımlayabiliriz. Performans ölçümünde sorulması gereken bazı sorular şu şekilde olabilir:

- İşini severek yaptığı söylenebilir mi?
- Müşteri gerensinimlerini hangi hızda kavrıyor?
- Programcı hangi zaman diliminde ne kadar iş ortaya koyuyor?
- Yaptığı işin kalitesi nedir?
- Etrafı ile iletişimde ne kadar başarılı?
- Yapılan tahminler (estimates) çerçevesinde işi tamamlayabiliyor mu?
- Yazdığı kodun hata oranı nedir?
- Yazdıgı kodun doğruluk oranı nedir?
- Yazdığı kodun dokümansyon oranı nedir?

Bu soruların cevaplarını aramadan önce, şu sorunun cevabını aramamızın daha doğru olacağını düşünüyorum: “iki programcı kendilerine verilen bir iş için aynı çözümü mü üretirler?” Bu sorunun cevabının hayır olduğunu biliyoruz. Her yiğidin yoğurt yiyiş tarzı değişik olduğu gibi, her programcının da kod yazış ve çözüm üretiş şekli farklı olacaktır. Peki bu durumda programcının performansını ölçmek için kaç satır kod yazdığı ya da işi hangi zaman diliminde bitirdiği önemini yitirmiyor mu? Her progragramcının seçtiği yöntem farklı olacağı için perfomans ölçümünü aynı terazi ile tartmamız imkansız hale geliyor. Bu yüzden yukarıda verdiğim soruların cevapları performans ölçümleri için ya yetersizdir ya da subjektif cevapların oluşmasını teşvik ederler. Örneğin müdürün sepmatik bulmasından dolayı iyi bir programcı olduğunu düşündüğü programcı benim için iyi olmayabilir. Yazılımda peformans ölçümlerinin sübjektif bir doğası olduğunu kabul etmemiz gerekiyor. Peki ama programcıların performansını nasıl ölçebiliriz? Zor iş doğrusu!

Bir ekibin içinde yer alan bir programcıya “bu ekibin en iyi programcısı kimdir” diye sorun, size hemen kim olduğunu ya da olmadığını söyleyecektir. İyi ama objektif yöntemlerle performansını ölçemediğimiz bir programcı hakkında nasıl olur da böyle bir kanaat ortaya atılabilir? Diyorum ya, burada yine subjektif algılar ortaya çıkıyor. Eğer objektif performans ölçümü mümkün degil ise, o taktirde subjektif performans ölçümlerini kullanmak zorundayız. Subjektif performans ölçümü nasıl yapılabilir, şimdi bu sorunun cevabını aramaya çalışalım.

Bir ekip içinde kimin iyi yazılımcı olduğunu ekip içinde herkes bilir demiştim. Bu gerçekten de böyle. Bu kanı subjektif algılarla oluşuyor. Bir progamcı hakkında bu algıların oluşabilmesi için programcının iki yetenek gurubunda faal olması gerekiyor. Bunlar:

- Somut teknik yetenekler (hard skills)
- Sosyal yetenekler (soft skills)

İlki programcının nasıl kod yazdığı, yazılım süreçlerine ne derecede hakim olduğu, müşteri gereksinmlerini kavrama ve koda dönüştürme kabiliyetinin gelişmişliği ile, ikincisi duygusal zekası, ekip çalışmasına yatkınlığı, bilgisine bilgi katma azmı, çalışma arkadasları ile iletişimi, hayata bakış açısı, hobileri ve programcı kimliği ile ilgilidir. Şimdi bu yeteneklerin programcıyı nasıl öne çıkardığını ve subjektif performans ölçümünde kullanılabilir veriler sağladığını yakından inceleyelim.

(**İyi programcı == iyi kod yazan ve en basit çözümü oluşturan programcı**) denklemi her zaman geçerlidir. Buradaki yazımda basit çözüm oluşturma yetisinin nasıl geliştirilebileceği konusuna değinmiştim. Geriye iyi kod yazma teknikleri kalıyor. Bu konu başlı başına bir bilim dalı. Bu yüzden uzun süre ihtisas gerektirir. Bu konuda sayılabilecek yüzlerce konu başlığı mevcut. Aklıma gelen bazı konu başlıklarını sizlerle paylaşmak isterim. Şöyle ki: temiz kod (clean kod) akımı, tasarım prensipleri, tasarım şablonları, çevik yazılım süreçleri, test güdümlü yazılım (TDD/ATDD = Test Driven Development, Acceptance Driven Development), kod kataları, versiyon ve sürüm yönetimi, versiyon kontrol yönetimi, modüler yapılar, yapılandırma araçları (build tools), performans ölçümleri, dijital elektronik, mikro denetleyiciler, IoT (Internet of Things) işletim sistemleri, donanım, alana has diller (DSL = Domain Specific Languages), UML, XML/XSL, sürekli entegrasyon (continuous integration), derleyici (compiler) mimarileri, webservice/microservice/SOA/rest/bulut mimarileri, büyük veri (big data), index ve arama teknolojileri, mobil programlama, katmanlı mimariler, yeniden yapılandırma (refactoring) teknikleri, nesneye yönelik programlama (OOP = Object Oriented Programming), fonksiyonel programlama, aspekt tabanlı programlama (AOP = Aspect Oriented Programming), alan bazlı programlama (DDD = Domain Driven Development), davranış güdümlü programlama (BDD = Behavior Driven Development), yazılım ustalığı (software craftsmanship)… Daha sayabileceğim birçok konu mevcut, lakin yazılıma yeni başlayanların gözünü korkutmak istemiyorum :)

Bu vermiş olduğum her bir konu başlığı için aylarca ya da yıllarca ihtisas yapmak gerekebilir. Yazılım bir ömür aralıksız sürebilecek bir ihtisas alanı gibi görünüyor. Nitekim iyi programcı olarak gördüğümüz meslektaşlarımız bu konuların hepsine hakim olmasalar bile, her konuda bilgi sahibidirler ya da çok kısa bir zaman diliminde otodidakt (kendi kendine bir konuyu öğrenebilme yetisi) olmalarından dolayı istedikleri konuyu öğrenebililer. Bu bilgilere sahip olunması, bilgi dahilinde kodun şekillendirilmesi ve uygulanan metot ve yöntemlerin tümü sonuç itibari ile somut teknik yetenekler olarak tanımladığım programcı yetenekleri gurubunu oluşturmaktadır. Bu yetenekleri gelişmiş bir programcının yazdığı koda bakarak dahi hangi seviyede olduğunu görmek mümkündür. Tabi kod yazmak madalyonun bir yüzü. Gelelim diğer yüzüne.

Yazılım bir giriş/çıkış (IO = Input/Output) transformasyonunda başka bir şey değildir. Kullanıcı veri girer, uygulama bu verileri başka verilere dönüştürür. Aynı şekilde programcı için de IO kuramı geçerlidir. Programcı yapacağı iş konusunda veri alır ve bunu koda dönüştürür. Bu dönüşüm sürecinin en mühim safhası, programcının kendisi yönünde gerçekleşen veri akışını nasıl algıladığıdır. Burada soft skills olarak tanımladığımız sosyal yetiler devreye giriyor. Sosyal yetiler olmadan bir programcının ne kadar mükemmel kod yazdığı hiçbir önem taşımamaktadır, çünkü madalyonun diğer kısmı kendisinde eksiktir.

Sosyal yetilerin subjektif performans ölçümlerini nasıl etkilediğini birkaç örnek üzerinde aktarmak istiyorum. Zaman zaman ekip toplantıları olur. Bu toplantılarda fikir alışverişine dahil olunuş seviyesi iletişim ve söz sahibi olma yetisinin ne kadar gelişmiş olduğuna işaret eder. Fikir sahibi olanlar ve toplantı içeriğine yön verebilenler öne çıkar ve göze batmaya başlarlar. Bir başka örnek vereyim. Modüler yazılım sistemleri arayüzlerden (interface, facade, rest api) oluşur/oluşmalıdır. Bu programcıları birlikte çalışmaya zorlayan bir durumdur. Ekip işi denilen durum buradan doğar. Ekip işine yatkınlık bir sosyal yetidir. Bu konudaki beceriklik subjektif performans ölçümünü sağlayıcı niteliktedir.

Genel olarak sosyal yetilerin programcı için bir süzgec vazifesi gördüğünü söyleyebiliriz. Programcı bu süzgeç sayesinde ekiple birlikte neyi nasıl yapması gerektiğine karar verir ve bu kararı somut teknik yetenekleri ile hayata geçirir. Madalyonun ön yüzünü sosyal yetiler, arka yüzünü somut teknik yetiler oluşturur. Programcıyı programcı yapan bu ikisinin birleşimidir. Birisinin eksikliği karşımızdaki şahsın programcı olmak için yeterli donanıma sahip olmadığı anlamına gelir. Ama bu eksiklik onun hiçbir zaman programcı olamayacagı anlamına gelmez. Şöyle ki…

Bahsetmiş olduğum somut ve sosyal yetenekler pratik yapılarak geliştirilebilecek cinstendir. Hiçbir şahıs programcı olarak doğmadığı gibi, bu yeteneklere de emek sarfetmeden sahip olamıyor. Bu yazımda pratik yapmanın önemine değindim. İnsanlar arası ilişkiler için gerekli sosyal yeteneklerin geliştirilmesi için de değişik türde pratikler yapılabilir. Burada önemli olan niyet ve sorumluluk bilincidir. İyi bir programcı olmak çok çaba sarfetmeyi gerektirir ve bireyin bu sorumluluğu alıp, alamadığı doğrudan subjektif performans ölçümlerine yansır.

Programcılık sadece kod yazmaktan ibaret değil. Aslına bakarsanız kod yazma programcılığın çok küçük bir bölümünü oluşturuyor. Oraya gelene kadar programcının diğer yeteneklerini kullanması gerekiyor. Bu yetenekler olmadan doğrudan kod yazma safhasına gelinemeyeceği aşikar. Bu durumda geriye programcının kendisini işine olan saygısı ve sorumluluk bilincine istinaden bahsetmiş olduğum yetenekleri geliştirmeye adaması kalıyor. Bu sadece yapılan işe duyulan tutku ile sağlanabilecek bir durum.

Nasıl Usta Programcı Olunurmuş

http://www.pratikprogramci.com/2015/09/09/nasil-usta-programci-olunurmus/

Genç yazılımcı okurlarımdan gelen soruların başında nasıl usta yazılımcı olurum sorusu geliyor. Nasıl usta yazılımcı olunurmuş, kendimce açıklamaya çalışayım.

Usta bir yazılımcı olmanın tek bir yolu var: çok okumak, çok pratik yapmak, daha çok okumak ve daha çok pratik yapmak. Yazılımcı ustalaşmaya doğru yürüdüğünde, bunu hisseder, o zamana kadar oku ve pratik yap!

Ustalaşma süreci kendini tamamen yazılıma vermek anlamına gelmez. Bu süreci birçok kanaldan beslemek yerine, neden tek bir kanal açık tutulsun ki! Nitekim bir konuda ustalaşmak için insanın kendisini yan dallarda da geliştirmesi gerekiyor. Sadece bu şekilde birçok şey sentezlenerek, ustalık için gerekli kıvam yakalanabilir. Yazılımcı örneğin yan dal olarak elektronik ya da müzik ile uğraşmalı. Buradan aldığı impulslar daha iyi bir programcı olması için gerekli ilhamı yakalamasını sağlayacaktır. Zaten amaç daha iyi bir programcı olmak olmalı, usta yazılımcı diye bir şey yok! Gelinen her kademenin bir üst kademesi var, bunu çarptığımız duvarlarda zaten hissetmiyor muyuz? O yüzden nihayi usta statüsüne erişmek mümkün degil, sadece daha da iyi olma potansiyeli mevcut.

Peki usta olmak gerekli mi ki? Gözlemlediğim bir şey var: işinde çok iyi olanlar, genelde yaptıkları işi çok seven insanlar oluyor. İnsan yaptığı iş için zaman içinde heyecanını kaybedebilir. Bu durumda da o işte ustalaşma fikri absürdleşiyor, çünkü zevk almadığın bir işte usta olmuşsun, sana ne faydası var! Ekmek teknesi diyorsan, hayata o pencereden bakmaya devam et. Bu durumda zaten ustalaşmıyorsun, uzmanlaşıyorsun.

Ustalaşmanın mayası tutkudur. Ustalaşmak istiyorum diye yola düşmeden önce bu tutkuyu keşfetmek gerekir. Onu keşfetmek için kalbinizi dinleyin. En çok neyi yapmayı seviyorsanız, tutku onun içinde gizlidir. Onu keşfettikten sonra hedefleriniz sizi bulur.

O yüzden içinizden gelen sesi dinleyin ve zevk aldığınız işlere yönelin, bu programlama olmak zorunda değil.

Programcıların Besleyip, Büyüttükleri Canavar

http://www.pratikprogramci.com/2015/09/03/programcilarin-besleyip-buyuttukleri-canavar/

Sizce bir yazılım projesinde en çok kimin sözü geçer? Programcıların mı yoksa proje yöneticisinin mi? Kimin izin verdiği kadar iş yapılır? Kim ne derse, o olur? Kim en çok yazılımcıların gözünü korkutur? Kim "istemem" deyip, masaya yumruğunu vurduğunda, herkesin yüreği ağzına gelir? Kim olabilir ki bu demeyin. Her projede böyle bir diktatör ya da bir canavar var. Tüm proje ekibi el ele vererek, onu ekibe dahil olması için çalışırlar. Bu şahsiyeti şimdi yakından tanıyalım.

Ekip oluşturulur, başına bir Scrum master ya da proje yöneticisi konur, sırtlar sıvazlanır ve projeye başlanır. Bahsettiğim canavar o gün doğar ve ekibin bir parçası haline gelir. Sonraki evrelerde onun ağırlığı hissedilir, onun istediği, izin verdiği şekilde hareket edilir, ondan çekinilir, hatta korkulur.

Projenin ilk birkaç haftası lay, lay, lomlar eşliğinde geride bırakılır ve aniden bir şeyin farkına varılır: o da ne, hiç hesapta olmayan birisi toplantı masasında oturuyor! Kim davet etti ki bunu diye sorulmaz, çünkü o sessizce ekibin bir parçası haline gelmiştir. İlk haftalarda pek sesini çıkarmaz o şahsiyet. Yanaşmadır, fazla ses çıkarmadan hayatını sürdürmeye, yani büyümeye çalışır. Büyüdüğünde ekibe ne zulümler yapacağını planlar, ama bundan kimseye bahsetmez. Maksadı çaktırmadan büyümektir. Henüz küçük olduğundan kimse onu kale almaz.

Proje ilerledikçe bir kenarda yanaşma gibi duran canavarcık büyür ve yavaş yavaş ağırlığını ortaya koymaya başlar. Artık proje bünyesinde ağırlığı hissedilir hale gelmiştir. Yazılımcılar onun istediği şekilde hareket etmeye başlarlar. O kırbacı eline alır ve herkesi istediği şekilde terbiye etmeye başlar. Yavaş yavaş kafa koparma operasyonlarına da başlamıştır. İstediğini projeden attırır, çünkü o kadar büyümüştür ki artık, ona kimse yan gözle bile bakmaya cesaret edemez. Onu adam etmek isteyen herkes ya işinden olur ya kendiliğinden çeker gider, çünkü adam olma şansı çok düşüktür. Çok asabi olduğundan, ona yaklaşmak mümkün değildir. Artık ofiste herkes ondan korkusundan fısıldaşarak konuşmaya başlanır. Herkes onun zulmünden korkar. Fazla mesailerin yegane sebebi odur. Herkes ondan nefret eder, ama onunla yaşamak zorundadırlar, çünkü projenin canavarı o dur.

Yoo! Bilim kurgu yazıları yazmaya başlamadım. Ben gerçeklerden bahsediyorum, kendi ellerimizle büyüttüğümüz o canavardan. Şu kadar yıldır yazılımla uğraşıyorum ve o canavarın olmadığı proje görmedim. İşin komik tarafı, programcıların o canavarı kendi elleriyle beslemeleri ve büyüdüğünde ona ellerini kaptırmaktan korkmalarıdır, çünkü onu elleriyle beslemeye devam etmek zorundadırlar. O böylece büyümeye devam eder ve o kadar büyür ki tüm ofisi kaplar. O canvarın ismi yazılım ürünü, software, program vs. dir.

Oluşturduğu şeyden korkan tek meslek grubu sanırım programcılar. En iyi niyetlerle projeye başlar programcılar, ama bir zaman sonra bakarlar ki karşılarında onları sürekli

ısıran, tartaklayan bir canavar oluşmuş. Bunun böyle olduğuna “bu işin ustasıyım, çeviğim, acayip bir ürün ortaya koyacağım” diyen programcılarda bile sahit oldum. En çevik programcı olsanız bile, bir zaman sonra nedense oluşan program içinden çıkılamaz hale geliyor. İstediğiniz kadar design pattern, design principle, integration pattern ya da test driven development kullanın, sonuç hep aynı olacaktır, çünkü yapısı itibari ile çevik olmayan bir şeyi yoğurmaya çalışıyorsunuz. Sizin değil, onun çevik olması gerekiyor ki sizi çevikleştirebilsin.

Projenizdeki bahsettiğim canavarı bir aslan olarak düşünün. Onunla karşı karşıya gelindiğinde, kimin kaybedeceği malum. Peki karşınızda bir aslan yerine 10 tane kedi olsaydı, durumu nasıl değerlendirirdiniz? Kedilerden korkmayacağınız gibi, onlarla baş etmeniz daha kolay olacaktır. Önemli olan yazılım ürününün aslana dönüşmesini engellemek ve kedi olarak kalmasını sağlayabilmektir. Bunu nasıl yapabiliriz?

Agaç yaşken eğilir atasözünü hatırlayalım. Kodu hamur olarak düşünürsek, yoğrulabilmesi için yaş olması gerekir. Hamura şu kattığımız sürece, hamuru istediğimiz şekle sokabiliriz. Aynı şey kod parçaları için de geçerlidir. Testleri olan bir kod parçasını istediğimiz şekilde yeniden yapılandırabiliriz. Yazılım ürünü hamur, testler ise hamuru yoğurduğumuz sudur. Kuruyan hamuru yoğurmak mümkün değildir, aynı şekilde testleri olmayan bir uygulamayı müşteri gereksinimleri doğrultusunda yeniden yapılandırmak imkansızdır. Bu yüzden programcılar bir zaman sonra uygulamayı değiştirmeye cesaret edemezler. Ellerinde kaskatı olmuş bir hamur parçası vardır. O zamanla büyür ve bahsettiğim canavar haline gelir.

Testler müstakbel canavarı ehlileştirmek ya da oluşmasını engellemek için kullanılabilecek yöntemlerin başında gelmektedir, lakin tek başlarına canavarın oluşmasını engelleyemezler. Bu canavarın üzerine birkaç koldan gitmemiz gerekir ki oluşması önlenebilsin. Hangi yöntemleri kullanabiliriz?

Böl ve yönet taktiğinden bahsettim. Bir aslan yerine on tane kedi ile karşı karşıya kalmayı tercih ederim. Aslanın benim yaptıklarımdan beslenmesini önlemek için oluşmasını engellemem gerekir. Yazılımda bunun karşılığı modüler yapılardır. Koca bir gökdelen inşa etmek yerine, birçok villadan oluşan bir site inşaatı gibi düşünebiliriz yazılımı.

Canavarın oluşmasını engelleyecek diğer bir yöntem ise, ona verilen zihnin ya da aklın (intelligence) sınırlandırılmasıdır. Bunu KISS yöntemiyle sağlayabiliriz. Çözüm ne kadar basit ise, kod o oranda daha az karmaşık olacaktır. Karmaşık olan kod bahsettiğim canavarı güçlendirir, onun zekasını ve bize hükmetme kabiliyetini artırır. Bu canavarın kodun içindeki karmaşadan beslendiğini söylemek yanlış olmaz. Bunun önüne geçmek için mümkün olan en basit çözümü seçmemiz gerekmektedir.

Şimdi kısa bir özet yapalım. Canavarın oluşmasını ya da büyümesini engellemek için yapmamız gerekenler:

- Bol bol test yazmak.
- Gökdelen tasarlamak yerine, uygulamayı parçalara yani modüllere bölmek ve tek katlı konutlar yapmak. Uygulama bu modüllerin birleşiminden bir araya gelen bir ürün

olmalı.
- Mümkün olan en basit çözümü uygulamak.

Bu yazdıklarım ne yazık ki canavarın oluşmasını engelleyen bir reçete değil. Bu listeye bir kalem daha eklememiz gerekiyor:

- Kodun ne durumda olduğunu bize gösteren araçlar kullanmak

Orta ve büyük çaplı projelerde ne kadar iyi niyetli olursanız olun, yazdığınız testlerin kapsama alanı (code coverage) %100 de olsa, test güdümlü yazılım bile yapsanız, kod karmaşa (complexity) seviyesi kaşla, göz arasında artabilir. Bu proje canavarını hemen hortlatır. Bunun önüne geçmek için kodun ne durumda olduğunu gösteren metriklerin sürekli göz önünde bulundurulması gerekmektedir. Bunlardan bazıları:

- Cyclomatic complexity değeri (örneğin iç içe geçmiş kaç if dalı bulunmakta? Bu değer ne kadar yüksekse, kod o oranda problemlidir.)
- Paketler arası döngü (Kodun bağımlılık oranını artırır.)
- Modüller arası döngü (Modüllerin tek başlarına iş yapmalarını ve tekrar kullanımı engeller)
- Sınıf başına düşen kod satır sayısı (ne kadar düşükse, kod o kadar az sorumluluğa sahiptir.)
- Kod tekrar adedi (Ne kadar yüksekse, kod o oranda kırılgandır.)
- Kalıtım derinliği (Kalıtım hirarşisi derin kodun bakımı ve geliştirilmesi zordur.)

Bahsetmiş olduğum bu metrikler kodun hangi statiksel yapıda olduğunu gösterir niteliktedirler ve sürekli göz önünde bulundurulmaları gerekmektedir. Bunu Sonar gibi bir araç yardımı ile yapabiliriz. Sonar FindBugs, PMD, Checkstyle, JaCoCo gibi araçların bir araya geldiği bir kod statik analiz aracıdır. Gerrit ve Jenkins gibi araçlarla birlikte kullanıldığında, kodun ne durumda olduğu bilgisi ekip içinde sürekli paylaşılan bir bilgi haline gelecektir.

Ne yazık ki kod yapısı itibari ile büyümeye eğilim gösteren bir şey. Her yeni müşteri isteği ile bu büyüme süreci desteklenmektedir. Bu gidişata dur demenin tek yolu, kod büyümeden parçalarına bölmek ve dikey değil, yatay büyümesini sağlamaktır. Dikey büyüyen kod zamanla bahsettiğim canavara dönüşecektir. Kod yatay büyüyorsa, o zaman bahsettiğim kediler ortaya çıkar ki onlarla baş etmek daha kolaydır.

Kod ne yazık ki büyümek zorunda, çünkü her yeni müsteri istediğinin hayata geçirilmesi gerekiyor. Kodun yatay büyümesini mümkün kılmak için SRP, OCP ve DIP gibi tasarım prensiplerinden faydalanabiliriz. Eğer test yazılımı ihmal edilmezse, yatay büyüme desteklenirse ve gidişat Sonar ile göz önünde tutulursa, ortaya çıkan şeyin çevik bir yazılım ürünü olduğunu söyleyebiliriz, çünkü her yeni müşteri isteği ile yeniden yapılandırılması mümkündür.

Benim bu noktada aklıma bir soru geldi, sizinle paylaşayım. Yazılım ürününü çevik yapan çevik yazılımcı mıdır yoksa yazılımcıyı çevik olmaya zorlayan çevik bir yazılım ürünü müdür? Bu konuda biraz düşünelim…

Ben yazılımcıyı çevikleştiren durumun yazılım ürününün çevik olmasından kaynaklandığını düşünüyorum. Yazılımcılar ne kadar çevik olurlarsa, olsunlar, üzerinde çalıştıkları yazılım ürünü yoğrulmaya izin vermiyorsa, yani çevik olma özelliklerine sahip değilse, bir zaman sonra çevik yazılımcılar gidişata uyak uydurarak, canavarın hakimiyetine boyun eğerler. Buna karşın bahsettiğim çevik özelliklere sahip bir yazılım ürünü programcısını çevik olmaya zorlayacaktır, çünkü programcının çevik olabileceği ya da olmak zorunda olduğu bir ortam mevcuttur. Önemli olan işte böyle bir ortamı oluşturabilmek. Geliştirilen uygulama da ekibin bir parçasıdır. Bu tüzel kişinin nasıl bir karakterde olacağını geri kalan yazılımcılar belirler.

Canavarla karşılaşmamanız dileğiyle..

Programcının Evrimi

http://www.pratikprogramci.com/2015/06/25/programcinin-evrimi/

Son zamanlarda kendimi “programcı olarak hedefim nedir, nereye doğru yol alıyorum, kendimi nasıl geliştirmeliyim, neler öğrenmeliyim” gibi soruların cevaplarını ararken yakalıyorum. Yaşlanıyorum ve bu beni mesleki evrimsel gelişimimin nasıl olması gerektiği konularına itmeye başladı. Bu konular hakkında kafa yorarken, bu blog yazısını yazma fikri doğdu ve düşüncelerimi sizlerle paylaşmak istedim.

Bir programcının evrimi nasıl olmalı? Cevabını aradığım soru bu. Bu ve buna benzer konularda daha önce bir takım yazılar kaleme almışım:

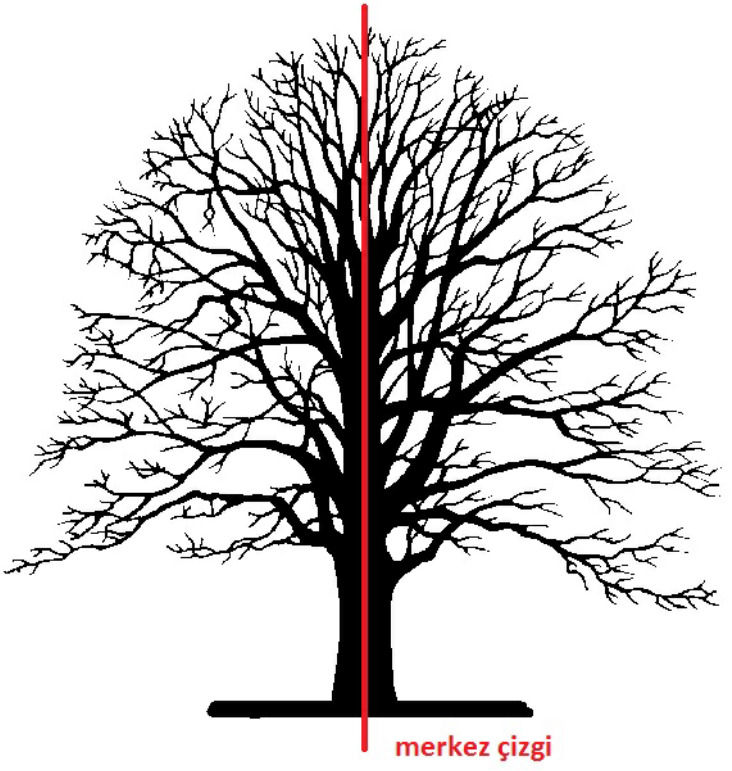
- 20 Yaş, 30 Yaş, 35 Yaş ve Bugünki Programcı Ben
- Hangi Programlama Dilini Öğrenmeliyim?
- Başkalarının Kodu Okunarak Daha İyi Programcı Olunabilir mi?
- Çok Gezen mi Bilir, Çok Okuyan mı?
- Acı Çekmeden Üstad Olunmaz
- Kişisel Gelişim
- Kod Kata ve Pratik Yapmanın Önemi

Kısaca özetlemek gerekirse, bir programcı yaptığı işin hakkını verebilmek için bir öğrenme makinası olmak zorunda. Yeniliklere adapte olabilmek için sürekli öğrenmek gerekiyor. Benim aklıma burada ilk olarak “neyi öğrenmeliyim” sorusu geliyor. Öğrenilmeyi bekleyen o kadar çok konu var ki, saymakla bitmez. Karşımıza çıkan her şeyi öğrenmek zorunda mıyız? Bu sorunun cevabının evet olduğunu düşünecek olursak, öğrenme sürecini nasıl şekillendirmeliyiz? Eğer her şeyi öğrenmek zorunda değilsek, hangi konulara odaklanmalıyız? Soru üstüne soru. İşin içinden çıkmak kolay değil. Bu sorular ve sağladığımız cevaplar programcı olarak evrimimizin nasıl gerçekleşmesi gerektiği konusuna ışık tutuyorlar. Bu sorularla yüzleşmek ve cevaplarının peşine düşmek zorundayız. Aksi taktirde ortada ne programcı kalır, ne de onun evrimi.

İsterseniz doğadaki canlıları kapsıyan evrimsel sürecin nasıl işlediğine bir göz atalım. Belki oradaki tespitlerimiz kendi mesleki evrimimizin nasıl olması gerektiğine ışık tutabilirler.

Doğadaki haliyle evrim, iyileştirme ve bulunulan ortama daha iyi adapte olma çabasıdır. Bu yaşadığımız gezegende milyonlarca yıllardır cereyan eden bir süreçtir. Her türlü canlı evrimsel gelişime tabidir. Evrimi gerekli kılan üzerinde yaşadığımız gezegenin dinamikleridir. Örneğin dünyanın kuzey kutbunda yaşayan canlılar oradaki soğuklara dayanıklı iken, benzer türdeki canlıların ekvator yakınlarında sıcaklarla yaşamayı becerdiklerini gözlemleyebilmekteyiz. Bir deney kapsamında bu canlıların yerleri değiştirilse, çok kısa bir zamanda telef olurlardı, çünkü sahip oldukları yetenekler karşılaştıkları yeni durumla baş edici seviyede değildirler. Yeni yeteneklerin gelişmesi ve bulunan ortam için gerekli adaptasyon zaman gerektirmektedir. Evrim zamanla işleyen bir süreçtir ama hayatta kalabilmek için zaruridir.

Kuzey kutbunda çalışan bir programcıyı ekvatorda çalışmaya zorlasaydık durum nasıl olurdu? Programcının sıcaklardan şikayet edeceği aşikar, ama telef olmayacağını da biliyoruz. Neden? Çünkü insan yüksek derecede bulunduğu ortama adapte olabilme yetisine sahiptir. Bu insanlar için evrimsel gelişimin bir getirisi. Ama bu konuyu derinleştirmeden, doğadaki diğer canlıların evrimsel gelişimlerine bir göz atalım. Aşagıda gezegenimizde vücut bulan yaşamı temsil eden evrim ağacını görmektesiniz.



Evrim ağacının en alt bölümünde tek hücreli amipler yer almaktadır. Ağaç yükseldikçe tek hücreli canlılardan türeyen diğer canlıları görmekteyiz. Ağacın tepesinde yaşadığı ortamın bilincinde ve zekasıyla karar alabilen canlıları görmeteyiz. Bu kesimde insanlar da yer almaktadır. Dallara ve bu dallardaki yapraklara doğru gidildiğinde, bulunduğu ortama yüksek derecede adapte olmuş ve uzmanlaşmış canlıları görmekteyiz. Buna çölde yaşayan kertenkele ya da eksi elli derecede yumurtadan yavrusunun çıkmasını bekleyen, kuluçkaya yatmış pengueni örnek verebiliriz. Bu canlılar bulundukları ortama adapte olmuş ve orada yaşayabilmek için gerekli beceri ve biyolojik yapıyı geliştirmişlerdir. Yaşadıkları ortamların bir anda değişmesiyle birlikte bu tür uzmanlasmış canlılar da yok olmaktadırlar, çünkü meydana gelen değişikliğe karşı kolayabilecek donanımda değildirler. Burada sadece insanlar büyük bir istisna oluşturmaktadırlar.

Alanında uzmanlaşmış yani bulunduğu ortama adapte olmuş bir canlıyı bir insan ile kıyasladığımızda, en büyük farklılığın uzmanlık seviyelerinde yattığını görmekteyiz. İnsan doğası itibari ile hiçbir şeyi iyi derecede yapamayan bir canlıdır. Bir yunus balığından daha iyi yüzemez, bir panterden daha iyi koşamaz, bir pireden daha iyi zıplayamaz, ama iyi olduğu bir alan var: çok hızlı öğrenir ve adapte olur. Gerekirse yüzmeye başlar ve çok iyi bir yüzücü olma yetilerini geliştirir. Uçamaz ama, uçak icat edip, en hızlı uçan canlıdan bile hızlı uçar. Kuzey kutbundan alınıp, ekvatora konduğunda, bunu dert etmez, hemen adapte olur. Bir yolunu bulur ve ayakta kalır.

İnsanların evrimleri sürecinde uzmanlıktan uzaklaşıp, her türlü durum ve ortama adapte olabilen canlılar olmayı tercih ettiklerini görmekteyiz. İnsanların hayatta kalmalarını sağlayan, bir konuda uzman olmamaları yani generalist olmalarıdır. Sahip oldukları zeka onların gerekli yetileri kısa sürede geliştirmelerini ve bulundukları ortama adapte olmalarını sağlamaktadır. Görüldüğü gibi uzmanlık ortam değiştiğinde yok olmayı beraberinde getirmektedir. İnsanlar hayatta kalabilmek için uzmanlığı terk etmişlerdir. Dünya ismini taşıyan bu gezegenin insanların kontrolünde olmasının tek sebebi, uzmanlık nedir bilmeyen bir canlı türünün süregelen evrimidir.

Yukarıda yer alan ağaca tekrar bir göz atalım. Orada bir merkez çizgi görmektesiniz. Merkez çizgisine olan uzaklık uzmanlaşma derecesini ifade etmektedir. Merkez çizgiye en uzakta olan canlılar bulundukları ortamda ayakta kalabilmek için uzmanlaşmış canlılardır. En ufak bir ortamsal değişiklikle bu canlılar yok olurlar. İnsanlar ağacın en tepesinde ve merkez çizgisinin üzerinde bulunmaktadırlar. Bu onların en gelişmiş canlılar olmaları yani sıra uzmanlık alanları olmayan, lakin her ortama ayak uydurabilen canlılar oldukları anlamına gelmektedir. Peki bunların bir programcının evrimi ile ne ilgisi olabilir. Açıklamaya çalışayım.

Gerçek hayatta da olduğu gibi uzmanlaşma belli şartlar altında yok olma tehlikesini beraberinde getiriyor. Yazılımın bir dalında uzmanlaşmış bir programcının akibeti uzman olduğu alanın yaşam süresiyle doğru orantılıdır. Bu alanın yok olması ile birlikte uzman yazılımcı da yok olur. Bu yok olma başka bir alana kayma ya da tamamen yazılım dışında kalma anlamına gelebilir. Uzman yazılımcının burada büyük bir riziko ile yaşadığını görmekteyiz. Yazılımcının bunun önüne geçmek için uzmanlığı terk edip, ustalığı seçmesi gerekmektedir.

Sadece bir programlama dilini kullanmak bir uzmanlaşma belirtisidir. “Hangi Programlama Dilini Öğrenmeliyim?” başlıklı yazımda aktarmaya çalıştım. Tek bir dile olan hakimiyet, o dilin revaçtan düşmesi ile programcının zora girmesi anlamına gelecektir. Bu yüzden dil ögrenim süreci tek bir dile değil, birçok dile yönelik olmalıdır. Birçok dil bilen bir programcı belki kullandığı dillerin hiçbirinde uzman değildir, lakin ortama göre dil seçerek, yok olmaya karşı koyabilir.

Programcı olarak nihayi amacımız merkez çizgisine yakın bir yerlerde konuşlanma olmalıdır. Bir yaprak oldugumuz taktirde, gelen güz rüzgarları ile evrim ağacından koparılacağımız şüphesizdir. Ana dallara ne kadar yakın olursak (merkez çigzi), daldan dala atlayarak, rüzgarlardan korunmamız mümkün olacaktır. Önemli olan daldan dala

atlayabilecek konumda olabilmemizdir.

Programcı öğrenme makinasıdır demiştim. Uzmanlaşma fikrini bir kenara bırakmamız, bilgi edinme çabalarımızı durdurabiliriz anlamına gelmemektedir. Aksine, usta bir yazılımcı olmak için daha çok alanda faal olmamız gerekmektedir. Bu sadece yazılım konularıyla sınırlı değil. Programcının müzik, elektronik ya da sportif faaliyetleri programcı yeteneklerini pekiştirmede faydalı olacaktır. Bir şiir kitabındaki tek bir mısranın programcının dünya görüşünü değiştirme gücüne sahip olabildiğini düşündüğümüzde, birçok çicekten bal almanın mesleki başarı için de ne kadar önemli olduğunun kanıtı olarak kabul edebiliriz.

Merkezi Versiyon Yönetim Sistemlerinde Sürüm Almak İçin İş Akışı Nasıl Şekillendirilir?

http://www.pratikprogramci.com/2015/06/09/merkezi-versiyon-yonetim-sistemlerinde-surum-almak-icin-is-akisi-nasil-sekillendirilir/

Bir önceki yazımda versiyon ve sürüm numaralarının nasıl oluşturulması gerektiği konusuna değinmiştim. Bu yazımda sürüm oluşturma sürecinde iş akışının Subversion, CVS ve ClearCase gibi merkezi versiyon kontrol sistemleri idaresindeki kaynak kodları üzerinde nasıl gerçekleştiğini ve sürüm ve versiyon numaralarının nasıl yönetilmesi gerektiğine değinmek istiyorum.

Subversion ve CSV gibi merkezi versiyon yönetim sistemlerinde programcılar tarafından yapılan değişiklikler main/trunk/head/master olarak bilinen ana kod dalına (branch) eklenir (commit). Sürüm bu daldan oluşturulur. Sürümün oluşturulabilmesi için sürüm yönetici tarafından codefreeze (ana kod dalında yapılan degişikliklerin durdurulması) ilan edilmesi gerekmektedir. Codefreeze ile programcılar belli bir zaman dilimi için üzerinde çalıştıkları uygulama özelliklerini (feature) commit edemezler. Böylece sürüm öncesi ana kod dalı mevcut hatalar giderilerek (bugfix), sürüm alınabilecek olgunluğa getirilir.

Sürüm yöneticisinin sürüm oluşturmak için sırayla yapması gereken işlemleri şu şekilde sıralayabiliriz:

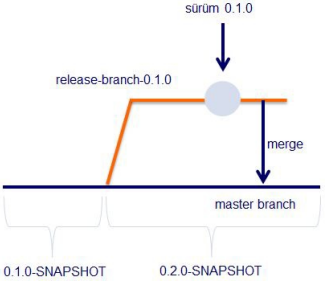
- İterasyon sonuna gelindiğinde sürüm yöneticisi codefreeze ilan eder. Programcılar ne zaman codefreeze ilan edileceğini bilirler ve çalışmalarını bu yönde şekillendirirler. Codefreeze sürecinde programcılar yaptıkları kodsal değişiklikleri ana kod dalına eklemezler. Tespit edilen hataların giderilmesine izin verilir.
- Sürüm yöneticisi sürekli entegrasyon sunucusunda (örnegin Jenkins) projeyi yapılandırır (build). Yapılandırma esnasında birim ve diğer testler koşturulur. Oluşan hatalar programcılar tarafından giderilir.
- Ana kod dalı sürüm oluşturmak için hazır hale gelmiştir. Sürüm ana kod dalından alınmaz. Bu amaçla bir sürüm dalı (release branch) oluşturulması gerekmektedir. Sürüm yöneticisi ana kod dalını baz alarak yeni bir sürüm dalı oluşturur. Oluşturulan sürüm dalı ilerde sürümde tespit edilen hataları gidermek için de kullanılır.
- Sürüm dalının oluşturulmasıyla birlikte codefreeze sonlandırılır. Bu şekilde programcılar ana kod dalı üzerinde çalışmalarına devam edebilirler. Programcıların bloke olmalarını engellemek amacıyla codefreeze aralığının kısa tutulması gerekmektedir.
- Sürüm yöneticisi sahip oldugu sürüm oluşturma araçları yardımı ile sürüm dalını baz alarak sürümü (release) oluşturur. Oluşan sürüm bir jar, war, ear, gz, tar, zip ya da so dosyasıdır. Oluşturulan sürümün başka bir proje bünyesinde kullanılmasını mümkün kılmak için sürüm Nexus gibi bir sürüm deposuna (release repository) eklenir.
- Sürüm dalı üzerinde yapılan tüm değişikliklerin tekrar ana kod dalına dönmesi gerekmektedir. Bu amaçla sürüm yöneticisi ya da bir programcı tarafından iki dalın birleştirildikleri merge işlemi gerçekleştirilir.

Şimdi klasik iş akışının nasıl uygulandığını somut örnekler üzerinde inceleyelim:

Küçük Değişiklikler (Minor Changes)

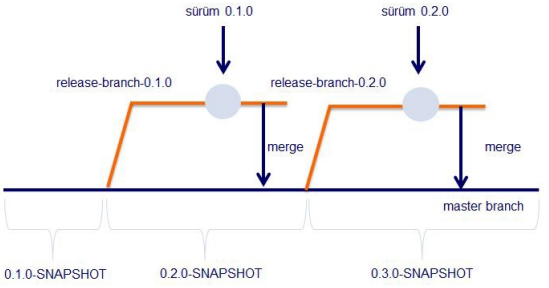
Bu senaryoda programcılar küçük versiyon numarasının değiştirilmesine sebep olan değişiklikleri ana kod dalına (master branch) eklemektedirler.

0.1.0 Numaralı Sürümü Oluşturmak İçin Yapılması Gerekenler



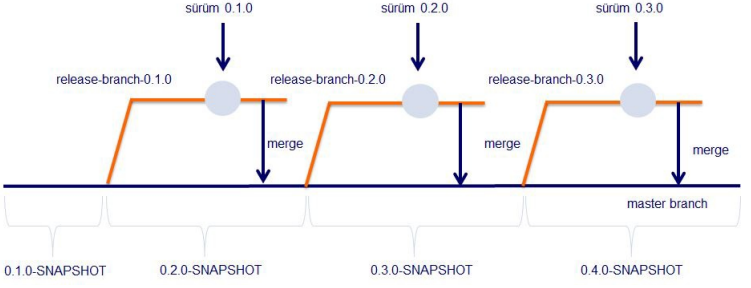
- Ana kod dalında kullanılan versiyon numarası 0.1.0-SNAPSHOT şeklindedir. Burada SNAPSHOT ibaresi sürüme doğru gidilirken kodun değişkliğe ugrayabileceği ve bu versiyon numarasının nihai olmadığı anlamına gelmektedir.
- Sürüm yöneticisi programcılardan kod üzerinde yaptıkları değişiklikler hakkında bilgi edinir. Sürüm yöneticisi yapılan değişikliklerin sadece küçük versiyon numarasını etkileyecek türde olduğunu tespit eder. Bu sürüm numarasının 0.1.0 şeklinde olması gerektiği anlamına gelmektedir.
- Sürüm yöneticisi release-branch-0.1.0 ismini taşıyan ve 0.1.0-SNAPSHOT ana kod dalını (master branch) baz alan yeni bir sürüm dalı oluşturur. Bunun öncesinde uygulamayı yapılandırarak, tüm testlerin çalışır durumda olduğunu kontrol eder.
- Oluşturulan sürüm dalı release-branch-0.1.0 sürüm öncesi ve sürümde meydana gelen hataları gidermek için kullanılacak kod dalıdır.
- Sürüm dalının oluşturulmasıyla codefreeze kaldırılır.
- Sürüm yöneticisi ana kod dalının versiyon numarasını 0.2.0-SNAPSHOT, sürüm dalının versiyon numarasını 0.1.0 olarak değiştirir. Versiyon numaralarının adapte edilmesi kullanılan sürüm araçları tarafında da sağlanabilir.
- Sürüm yöneticisi release-branch-0.1.0 dalına geçerek, 0.1.0 versiyon numarasını taşıyan sürümü oluşturur. Sürüm, sürüm deposuna yerleştirilir.
- Sürüm dalı release-branch-0.1.0 üzerinde yapılan tüm değişiklikler ana kod dalı 0.2.0-SNAPSHOT ile senkronize (merge) edilir.

0.2.0 Numaralı Sürümü Oluşturmak İçin Yapılmasi Gerekenler



- Ana kod dalında kullanılan versiyon numarası 0.2.0-SNAPSHOT şeklindedir.
- Sürüm yöneticisi programcılardan kod üzerinde yaptıkları değişiklikler hakkında bilgi edinir. Sürüm yöneticisi yapılan değişikliklerin sadece küçük versiyon numarasını etkileyecek türde olduğunu tespit eder. Bu sürüm numarasının 0.2.0 şeklinde olması gerektiği anlamına gelmektedir.
- Sürüm yöneticisi release-branch-0.2.0 ismini taşıyan ve 0.2.0-SNAPSHOT ana kod dalını (master branch) baz alan yeni bir sürüm dalı oluşturur. Bunun öncesinde uygulamayı yapılandırarak, tüm testlerin çalışır durumda olduğunu kontrol eder.
- Oluşturulan sürüm dalı release-branch-0.2.0 sürüm öncesi ve sürümde meydana gelen hataları gidermek için kullanılacak kod dalıdır.
- Sürüm dalının oluşturulmasıyla codefreeze kaldırılır.
- Sürüm yöneticisi ana kod dalının versiyon numarasını 0.3.0-SNAPSHOT, sürüm dalının versiyon numarasını 0.2.0 olarak değiştirir.
- Sürüm yöneticisi release-branch-0.2.0 dalına geçerek, 0.2.0 versiyon numarasını taşıyan sürümü oluşturur. Sürüm, sürüm deposuna yerleştirilir.
- Sürüm dalı release-branch-0.2.0 üzerinde yapılan tüm değişiklikler ana kod dalı 0.3.0-SNAPSHOT ile senkronize (merge) edilir.

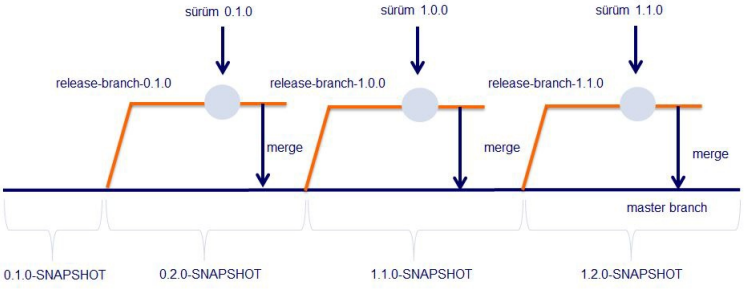
0.3.0 Numaralı Sürümü Oluşturmak İçin Yapılması Gerekenler



- Ana kod dalında kullanılan versiyon numarası 0.3.0-SNAPSHOT şeklindedir.
- Sürüm yöneticisi programcılardan kod üzerinde yaptıkları değişiklikler hakkında bilgi edinir. Sürüm yöneticisi yapılan değişikliklerin sadece küçük versiyon numarasını etkileyecek türde olduğunu tespit eder. Bu sürüm numarasının 0.3.0 şeklinde olması gerektiği anlamına gelmektedir.
- Sürüm yöneticisi release-branch-0.3.0 ismini taşıyan ve 0.3.0-SNAPSHOT ana kod dalını (master branch) baz alan yeni bir sürüm dalı oluşturur. Bunun öncesinde uygulamayı yapılandırarak, tüm testlerin çalışır durumda olduğunu kontrol eder.
- Oluşturulan sürüm dalı release-branch-0.3.0 sürüm öncesi ve sürümde meydana gelen hataları gidermek için kullanılacak kod dalıdır.
- Sürüm dalının oluşturulmasıyla codefreeze kaldırılır.
- Sürüm yöneticisi ana kod dalının versiyon numarasını 0.4.0-SNAPSHOT, sürüm dalının versiyon numarasını 0.3.0 olarak değiştirir.
- Sürüm yöneticisi release-branch-0.3.0 dalına geçerek, 0.3.0 versiyon numarasını taşıyan sürümü oluşturur. Sürüm, sürüm deposuna yerleştirilir.
- Sürüm dalı release-branch-0.3.0 üzerinde yapılan tüm değişiklikler ana kod dalı 0.4.0-SNAPSHOT ile senkronize (merge) edilir.

Büyük ve Küçük Değişiklikler (Major, Minor Changes)

Bu senaryoda programcılar hem küçük hem de büyük versiyon numarasının değiştirilmesine sebep olan değişiklikleri ana kod dalına (master branch) eklemektedirler.



0.1.0 Numaralı Sürümü Oluşturmak İçin Yapılması Gerekenler

- Ana kod dalında kullanılan versiyon numarası 0.1.0-SNAPSHOT şeklindedir.
- Sürüm yöneticisi programcılardan kod üzerinde yaptıkları değişiklikler hakkında bilgi edinir. Sürüm yöneticisi yapılan değişikliklerin sadece küçük versiyon numarasını etkileyecek türde olduğunu tespit eder. Bu sürüm numarasının 0.1.0 şeklinde olması gerektiği anlamına gelmektedir.
- Sürüm yöneticisi release-branch-0.1.0 ismini taşıyan ve 0.1.0-SNAPSHOT ana kod dalını (master branch) baz alan yeni bir sürüm dalı oluşturur. Bunun öncesinde uygulamayı yapılandırarak, tüm testlerin çalışır durumda olduğunu kontrol eder.
- Oluşturulan sürüm dalı release-branch-0.1.0 sürüm öncesi ve sürümde meydana gelen hataları gidermek için kullanılacak kod dalıdır.
- Sürüm dalının oluşturulmasıyla codefreeze kaldırılır.
- Sürüm yöneticisi ana kod dalının versiyon numarasını 0.2.0-SNAPSHOT, sürüm dalının versiyon numarasını 0.1.0 olarak değiştirir.
- Sürüm yöneticisi release-branch-0.1.0 dalına geçerek, 0.1.0 versiyon numarasını taşıyan sürümü oluşturur. Sürüm, sürüm deposuna yerleştirilir.
- Sürüm dalı release-branch-0.1.0 üzerinde yapılan tüm değişiklikler ana kod dalı 0.2.0-SNAPSHOT ile senkronize (merge) edilir.

1.0.0 Numaralı Sürümü Oluşturmak İçin Yapılması Gerekenler

- Ana kod dalında kullanılan versiyon numarası 0.2.0-SNAPSHOT şeklindedir.
- Sürüm yöneticisi programcılardan kod üzerinde yaptıkları değişiklikler hakkında bilgi edinir. Sürüm yöneticisi yapılan değişikliklerin büyük versiyon numarasını etkileyecek türde olduğunu tespit eder. Bu sürüm numarasının 1.0.0 şeklinde olması gerektiği anlamına gelmektedir.
- Sürüm yöneticisi release-branch-1.0.0 ismini taşıyan ve 0.2.0-SNAPSHOT ana kod dalını (master branch) baz alan yeni bir sürüm dalı oluşturur. Bunun öncesinde uygulamayı yapılandırarak, tüm testlerin çalışır durumda olduğunu kontrol eder.
- Oluşturulan sürüm dalı release-branch-1.0.0 sürüm öncesi ve sürümde meydana gelen hataları gidermek için kullanılacak kod dalıdır.
- Sürüm dalının oluşturulmasıyla codefreeze kaldırılır.
- Sürüm yöneticisi ana kod dalının versiyon numarasını 1.1.0-SNAPSHOT, sürüm dalının versiyon numarasını 1.0.0 olarak değiştirir.
- Sürüm yöneticisi release-branch-1.0.0 dalına geçerek, 1.0.0 versiyon numarasını taşıyan sürümü oluşturur. Sürüm, sürüm deposuna yerleştirilir.
- Sürüm dalı release-branch-1.0.0 üzerinde yapılan tüm değişiklikler ana kod dalı 1.1.0-SNAPSHOT ile senkronize (merge) edilir.

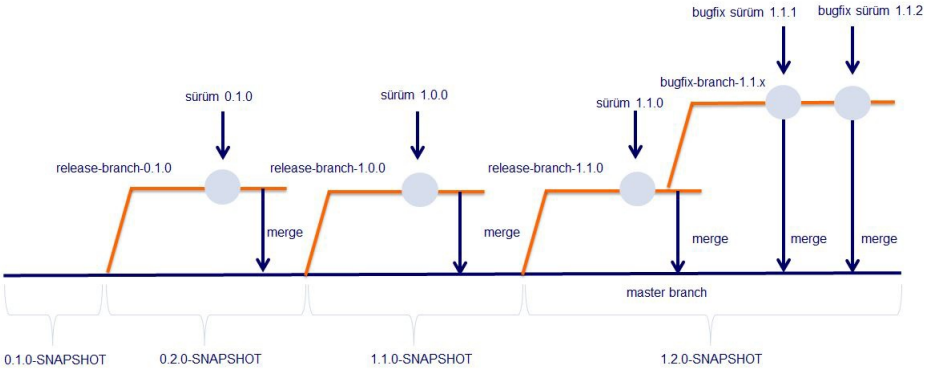
1.1.0 Numaralı Sürümü Oluşturmak İçin Yapılması Gerekenler

- Ana kod dalında kullanılan versiyon numarası 1.1.0-SNAPSHOT şeklindedir.
- Sürüm yöneticisi programcılardan kod üzerinde yaptıkları değişiklikler hakkında bilgi edinir. Sürüm yöneticisi yapılan değişikliklerin küçük versiyon numarasını etkileyecek türde olduğunu tespit eder. Bu sürüm numarasının 1.1.0 şeklinde olması gerektiği anlamına gelmektedir.
- Sürüm yöneticisi release-branch-1.1.0 ismini taşıyan ve 1.1.0-SNAPSHOT ana kod dalını (master branch) baz alan yeni bir sürüm dalı oluşturur. Bunun öncesinde uygulamayı yapılandırarak, tüm testlerin çalışır durumda olduğunu kontrol eder.
- Oluşturulan sürüm dalı release-branch-1.1.0 sürüm öncesi ve sürümde meydana gelen hataları gidermek için kullanılacak kod dalıdır.
- Sürüm dalının oluşturulmasıyla codefreeze kaldırılır.
- Sürüm yöneticisi ana kod dalının versiyon numarasını 1.2.0-SNAPSHOT, sürüm dalının versiyon numarasını 1.1.0 olarak değiştirir.
- Sürüm yöneticisi release-branch-1.1.0 dalına geçerek, 1.1.0 versiyon numarasını taşıyan sürümü oluşturur. Sürüm, sürüm deposuna yerleştirilir.
- Sürüm dalı release-branch-1.1.0 üzerinde yapılan tüm değişiklikler ana kod dalı 1.2.0-SNAPSHOT ile senkronize (merge) edilir.

Eğer API değişiklikleri yapılmazsa, bir sonraki sürümde sürümün taşıyacağı versiyon numarasi 1.3.0, ana kod dalının versiyon numarasi 1.4.0-SNAPSHOT şeklinde olacaktır. API değişikleri yapılması durumunda sürüm numarasının 2.0.0 şeklinde olması gerekmektedir, çünkü yapılan değişiklikler bir önceki sürümle API seviyesinde uyumlu değildirler ve bu durumun versiyon numarası aracılığı ile ifade edilmesi gerekmektedir. 2.0.0 sürüm akabinde ana kod dalının versiyon numarasi 2.1.0-SNAPSHOT olur.

Sürüm Hatalarının Giderilmesi (Bugfix)

Bu senaryoda üçüncü sürüm ile oluşturulan ve 1.1.0 versiyon numarasına sahip sürümde meydana gelen hataların giderildikten sonra yeni bir sürümün nasıl oluşturulduğu incelenmektedir.



Bugfix Sürümü İçin Yapılması Gerekenler

- Sürüm yöneticisi release-branch-1.1.0 sürüm dalını baz alarak bugfix-branch-1.1.x ismini taşıyan yeni bir kod dalı oluşturur. 1.1.0 sürümü bünyesinde yer alan hataların bugfix-branch-1.1.x kod dalında giderilmesi gerekmektedir. release-branch-1.1.0 kod dalı değiştirilemez. Değiştirilmesi durumunda 1.1.0 versiyon numarasına sahip bir sürümün tekrar oluşturulması mümkün değildir.
- bugfix-branch-1.1.x bünyesinde hatalar giderilir.
- bugfix-branch-1.1.x kullanılarak giderilen hataları ihtiva eden 1.1.1 1.1.2 versiyon numaralı bugfix sürümler oluşturulur.
- bugfix-branch-1.1.x kod dalındaki diğer hatalar giderildikçe
- bugfix numarası bir artırılır ve yeni bugfix sürümü oluşturulur.
- bugfix-branch-1.1.x kod dalında yapılan tüm değişiklikler ana kod dalına (master branch) geri aktarılır. Bu değişiklikler release-branch-1.1.0 kod dalına aktarılmaz.

Bir sonraki yazımda versiyon ve sürüm numaralarının Git iş akışında (gitflow workflow) nasıl yönetildiği konusuna değineceğim.

2015 Yılına Geldik, Hala Spring'le Birlikte İnterface sınıf mı Kullanmak Zorundayız?

http://www.pratikprogramci.com/2015/05/19/2015-yilina-geldik-hala-springle-birlikte-interface-sinif-mi-kullanmak-zorundayiz/

Ada, C#, D, Dart, Delphi, Java, Logtalk, Object Pascal, Objective-C, PHP, Racket, Seed7, Swift. Bu dillerin ortak bir yanı var. Hepsinde interface vari sınıf yapılarını bulmak mümkün. İnterface sınıfların ne olduklarını ve nasıl kullanıldıklarını bu yazımda açıklamaya çalıştım. İnterface sınıflar benim için iyi bir tasarımın vazgeçilmez ögeleri. Bu yazımda neden daha aktif kullanılmaları gerektiğinden bahsetmek istiyorum.

Bu yazı bir şahsın "2015 yılına geldik, hala Spring'le birlikte interface sınıf mı kullanmak zorundayız?" söylemine cevaben oluştu. İnterface sınıflardan faydalanmak için birçok neden var. Bunların en başından tartışmasız DIP (Dependency Inversion Principle – Bağımlılıkların Tersine Çevrilmesi Prensibi) gelir. Şimdi aşağıdaki Java kod örneğine bakalım:

```
ArrayList list = new ArrayList();
```

Bu şekilde kod yazan DIP prensibini anlamamış demektir, çünkü oluşturulan bağımlılığın yönü somut bir sınıfa doğru. list değişkeni somut bir sınıf olan java.util.ArrayList sınıfına işaret etmektedir. Denklemin her iki tarafında da somut bir sınıf yer almaktadır. DIP prensibini uygulamak için denklemin iki tarafını da soyutlaştırmamız gerekiyor. İlk adımı şöyle atabiliriz:

```
List list = new ArrayList();
```

Değişkenin veri tipini bir interface olan java.util.List olarak değiştirdim. Bir interface sınıfının kullanıldığı kodun ifade etmeye çalıstığı bir şey bulunmaktadır. Bunun ne olduğunu gerçek hayattan bir örnek vererek, açıklamak istiyorum. Markete alışverişe gidiyorsunuz. Bir zaman sonra ödeme yapmak için kasada beklemeye başlıyorsunuz. Sıra size geldiğinde, kasiyer sadece 10 TL banknotlar ile ödeme yapabilirsiniz diyor. Siz doğal olarak "neden ayrım yapıyorsunuz, her türlü para ile ödeme yapabilmeliyim" diyorsunuz. Haklısınız.

Burada para terimini soyut bir yapı olarak kullandınız. Sizin için 10, 50 ya da 100 TL'lik banknotların hepsi birer para. Para terimi sizin için birçok banknot türüne işaret eden soyut ve genel bir kavram. Cüzdanınızda bulunan paranın ne türde olduğu sizi ilgilendirmiyor. Önemli olan miktarı ve parayı ne için kullanabileceğiniz. Lakin karşınızda sadece 10 TL'lik banknotların geçerli olduğu bir market çıktığında, bunu anlamlı bulmuyorsunuz, çünkü sahip olduğunuz para kavramı her cinsten parayı kapsıyor. Market sadece 10 TL'lik banknotları kabul ederek, ödeme şeklini daraltmış durumda. Neden bir market her türden parayı kabul etmek varken, 10 TL'lik banknotları tercih ediyor olsun? Şimdi marketi bir programcı olarak düşünelim. Programcıların çogu kod yazarken 10

TL'lik banknot kabul eden market gibi davranır. Aşağıdaki koda bakalım:

```
class Market{
    10TLBanknot para = new 10TLBanknot();
}
```

Market sınıfı sadece ve sadece 10TLBanknot veri tipinde olan bir yapı kullanabiliyor. Bu markete gidip sadece 10 TL'lik banknotlarla ödeme yapmaktan farksız bir durum. Para değişkeni somut bir yapıya işaret etmektedir. Somut bir şey elle tutulur, gözle görülür bir yapıdır. 10 TL'lik banknot somut bir yapıdır. 10 TL'lik banknot 10 TL'lik banknot anlamına gelmektedir. Buna karşın para terimi her türden banknot anlamına gelmektedir. 10 TL'lik banknot geçerli dendiğinde, başka hiçbir türden banknot geçmez denmiş olur. Para geçerli dendiğinde, her türlü banknot geçerli denmiş olur. Somut yapılar kullanıldığında hedef gösterilmiş olunur. Soyut yapılar kullanıldığında hedefin ne olduğunu tam olarak görmek mümkün değildir, çünkü cüzdandan para olarak 10 TL'lik banknot da çıkabilir, 100 TL'lik banknot da. Somut kavramlar kullanıldığında seçim yelpazesi daraltılır, soyut kavramlar kullanıldığında seçim yelpazesi genişletilir. Kod yazarken somut kavramlar kullanıldığında, kod doğrudan seçilmiş olan somut yapıya bağlanır ve değiştrilmesi zora girer. Buna karşın soyut yapılar kullanıldığında, kodu soyut yapının değişik türlerini kullanmak suretiyle değiştirmek kolaylaşır. Buna esnek bağ oluşturma ismi verilmektedir.

Şimdi tekrar aşağıda yer alan koda göz atalım. Bu kod ne ifade etmektedir?

```
class Listeler{
    List list = new ArrayList();
}
```

Denklemin sol tarafı soyut olan List sınıfını kullanarak burada soyut listeler kullanılır demekle birlite, denklemin sağ tarafındanki ArrayList "sol taraf istediğini söylesin, burada somut bir liste sınıfı kullanılmaktadır" demektedir. Sağ tarafın ağırlığı daha fazladır ve onun dediği geçerlidir. Bu durumda kod bünyesinde somut bir yapı kullanmaya başlamış oluruz. Somut yapıların kullanılmalarındaki en büyük sıkıntı, zor değiştirilebilir olmalarıdır. ArrayList yerine LinkedList kullanmak isteseydik, yukarıda yer alan sınıfı degiştirip, yeniden derlememiz gerekirdi. Bunun yanı sıra yukarıda yer alan sınıfı nereye gönderirseniz gönderin, yanında ArrayList sınıfını da olmak zorundadır. Bu sebeplerden dolayı Listeler sınıfı DIP ile uyumlu değildir. Bu sınıfı DIP ile uyumlu hale getirebilmek için new ile yeni nesne oluşturma işlemini sınıfın dışına almamız gerekmektedir. Bu işleme bağımlılıkların enjeksiyonu (dependency injection) ismi verilmektedir ve aşağıda yer alan Market sınıfında uygulanmaktadır.

Şimdi aşağidaki sınıfa bir göz atalım:

```
class Market{
    private Para para;

    Market(Para para){
        this.para = para;
    }
}
```

Konstrüktör aracılığı ile herhangi bir Para implementasyonunu Market sınıfına enjekte edebilir hale geldik. Artık bu markette her türlü banknot kullanılabilmektedir. Market sınıfı her türlü Para implementasyonunu kabul eden soyut bir yapıya kavuştu ve böylece test edilebilirligi de artmış oldu.

Tekrar "2015 yılına geldik, hala Spring'le birlikte interface sınıf mı kullanmak zorundayız?" sorusuna geri dönmek istiyorum. Spring çatısında teknik olarak bu zorunluluk ortadan kalktı. Spring somut sınıflardan nesneler oluşturup, başka sınıflara enjekte edebiliyor. Küçük çaplı uygulamalarda belki interface sınıfların kullanımı o kadar da çok mühim olmayabilir. Lakin müşteri gereksinimlerinin sürekli değişiklige ugradığı dinamik bir ortamda interface sınıfların kullanımı uygulamanın geleceği açısından hayati olabilir. Burada uygulamanın yeni müşteri gereksinimleri doğrultusunda yeniden yapılandırılabilmesi ve yoğrulabilmesi önem taşımaktadır. Bunu sağlayabilmenin tek yöntemi uygulama testlerinin oluşturulmasıdır. İnterface sınıfların ağırlıklı kullanıldığı uygulamalar mock, stub implementasyonları ve bağımlılıkların enjekte edilmesi tekniğiyle kolay test edilebilir hale gelmektedirler.

İnterface sınıfları arkasına gizlenmiş kod birimlerini kara kutu gibi görebiliriz. Kendi içlerinde nasıl işledikleri önem taşımamaktadır. Önemli olan nasıl bir kullanım arayüzüne sahip olduklarıdır. Kendi mock ve stub implementasyonlarımız aracılığı ile ayrıca kara kutunun davranış biçimini değiştirerek, uygulamayı istediğimiz şekilde test edebiliriz.

Bir Mahrumiyetin Sonu

http://www.mikrodevre.com/2015/01/18/bir-mahrumiyetin-sonu/

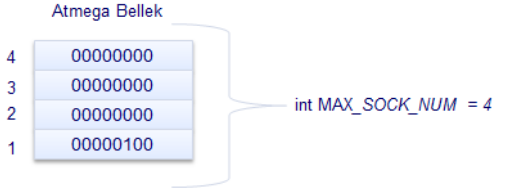
Geçenlerde genç bir yazılımcı arkadaşımızdan bir mail aldım. Şöyle yazmış:

```
int MAX_SOCK_NUM=4;
for(int sock=0; sock < MAX_SOCK_NUM; sock++){
...
}
```

4 için int yapmanın anlamı ne? Bu kod Arduino açık kaynak kodu, sock değişkeni için neden short tipini seçmemişler? Bunları neden önemsemiyorlar?

Yazıma devam etmeden, arkadaşımızın ne demek istediğine açıklama getirmek istiyorum. Int genelde 32 bitlik bir veri tipidir, yani unsigned bir int ile 0 – 4294967295 (4.3 milyar) arasında bir rakamı adresleyebiliriz. Eğer bir biti negatif rakamları da tutmak için ayıracak olursak, int veri tipindeki bir değişkende +- 2.1 milyarlık bir rakamı tutabiliriz.

Yukarıda yer alan kod örneğinde 4 rakamını tutabilmek için int veri tipinde olan ve MAX*SOCK*NUM ismini taşıyan bir değişken oluşturulmuş. Bu kod Arduino gibi bir mikro denetleyici bünyesinde koşturulduğunda, bu değişkenin işgal ettiği bellek miktarı aşağıdaki şekilde olacaktır:



Koddan anlaşıldığı gibi, uygulamanın ihtiyacı olan değişkenin büyüklüğü 4 rakamını temsil edebilecek bir bit kombinasyonudur. Bu bit kombinasyonu bir byte olarak şu şekilde yazabiliriz:

```
00000100
```

Eğer ihtiyacımız tek bir byte ise, bu durumda değişken için gerekli bellek alanı şu şekilde olacaktır:

Byte veri tipi ile bir rakamı temsil etmek mümkün olmadığı için bir yüksek veri tipi olan short veri tipini kullanabiliriz. Short 16 bit uzunluğunda olup, 0-65535 arası bir rakamı temsil edebilmektedir ki bu da bizim işimizi görmektedir. Sonuç itibari ile değişken bünyesinde tutmak istediğimiz değer 4 rakamıdır.

Bu kadar laf salatasına ne gerek var, ha int, ha short, ne fark eder diyebilirsiniz. Arada çok ince bir fark var. Bunu anlatmaya çalışacağım. Bu fark bizim yazılımcı olarak donanıma ne kadar uzak ya da yakın olduğumuzu gösteriyor.

Buradaki yazimda ifade etmeye çalıştım. Biz yazımcılar ne yazık ki donanımdan çok uzaklaştık. Bunun çok çeşitli sebepleri var. Başlıca sebebi donanımcıların son yıllarda baş döndürücü hızda donanımda kat ettikleri yol ve biz programcıların her şeyi soyutlama merakı. Soyutlaya, soyutlaya öyle bir seviyeye geldik ki, artık en altta, yani mikro işlemci bünyesinde olup bitenlerden bihaberiz. Bunun en güzel örneğini yazımın başında verdiğim kod örneği teşkil ediyor. Programcının birisi kısıtlı bellek alanına sahip olan bir mikro denetleyici için short veri tipini kullanması gerekirken, int veri tipini kullanmış ve en az 2 byte genişlikteki bellek alanını kullanılmaz hale getirmiştir.

İsterseniz burada ufak bir analiz yapalım. Programcı muhtemelen 32/64 bitlik mikro işlemciler için kod yazmaktadır. Aynı programcı 8 bitlik bir mikro denetleyici için kod yazmaya başladığında 32/64 bitlik kod yazma alışkanlıklarına devam etmekte ve kod yazdığı donanımın sahip olduğu özellikleri göz ardı etmektedir. Eğer kullandığı donanımın yapısal özelliklerini göz önünde bulundursaydı, gerekli değişkeni şu şekilde tanımlardı:

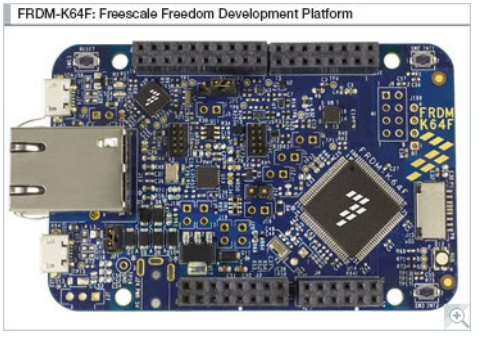
```
short MAX_SOCK_NUM=4;
```

Bu bence donanımdan kopmuşluğun en belirgin ibaresidir. Ben buna en çok Java dünyasında rastlıyorum. Java sanal makinesinin var oluşu (JVM) ve yazılan Java kodunun her platformda koşturulabilir olması, Java programcılarını tamimiyle donanımdan koparmış ve matrixde yaşayan programcılar haline getirmiştir.

“Ben programcıyım, donanımla ilgilenmem” yaklaşımını doğru bulmuyorum. Ne için kod yazdığımızı bilmemiz gerekir. Soyutluk seviyesi yükseldikçe ne yazık ki donanımdan biraz daha uzaklaşıyor ve kaportanın altında ne olup bittiğini bile kestiremez hale geliyoruz. Bu gidişatı değiştirmek o kadar kolay değil, çünkü mikro işlemci tasarımcıları hız sınırına yaklaştıklarını anladıkları için artık hücre bölünme yöntemiyle donanım geliştirmeye başladılar. Çok kısa bir zaman sonra 512 ve 1024 çekirdekli mikro işlemciler için kod yazmak zorunda kalacağız. Cumanın gelişi perşembeden belli değil midir?

Bence burada bir mahrumiyet söz konusu. Donanımdan ve donanımı anlamaktan mahrum kaldık ya da edildik. Bilgisayarın başına oturduğumuz zaman sadece Eclipse ya da başka bir IDEyi görüyoruz, başka bir şey bizi ilgilendirmiyor. Kod yazıyoruz, nasıl çalıştığını anlamıyoruz. Ama kanımca artık bu mahrumiyetin sonu geldi.

Dünyada olup, biten olumsuzlukları göz ardı edebilecek olursak, teknik anlamda öyle muazzam, öyle güzel bir çağda yaşıyoruz ki, bu güzelliği kelimelerle anlatmam mümkün değil. Yazılımcı olarak donanıma yaklaşma ya da geri dönme kapıları bize açılmış durumda. Artık bir yazılımcı açışından donanım ve yazılıma eş değerde hakimiyetin mümkün olacağı çağda yaşıyoruz. Bunun bir örneği aşağıda yer alıyor.



Bu bir mikro denetleyici. 120 MHZ hızında ve 256KB bellek ve 1024KB flash bellek ihtiva ediyor. Biliyorum! Taş devrinde gibi hissettiniz kendinizi. Benim 20 sene önceki 386DX bilgisayarım bile bundan hızlıydı ve daha fazla belleği vardı. Ama mevzu bu değil.

Burada gördükleriniz, donanım ile ilgilenen ve donanım dünyasında giriş yapmak isteyen yazılımcılar için giriş biletleri. Bakın bunlarda var:

Arduino

Raspberry PI

Günümüzde popüler olmaya başlayan bazı mikro denetleyici ve mikro işlemcilerden örnekler vermeye çalıştım. Bu avuç içine sığacak büyüklükteki donanımlar için Java, C ve Assembly gibi dillerde program yazmak mümkün. Bu tür donanım için kod yazan programcılar donanıma çok yakın bir seviyede çalışıyorlar ve işlemcilerin bünyesinde olup, bitenleri daha iyi kavrayabiliyorlar. Bu onların daha iyi programcı olmalarını sağlıyor.

Programcılıkta daha derin bir anlayış seviyesine gelebilmek için donanıma yaklaşmanın, onun nasıl işlediğini anlamanın önemli bir rol oynadığını düşünenlerdenim. Programcılar donanımdan uzaklaşarak değil, ona yaklaşarak 512, 1024 ya da 2048 çekirdekli sistemler için paralel koşturulabilen uygulamalar geliştirebilirler.

Tarihte yeni bir dönemin başındayız. Bizi teknolojik açıdan çok güzel gelişmeler bekliyor. Bu gelişmelerin bir parçası olabilirsiniz.

İşin İnceliklerini Sadece Ustası Bilir

http://www.mikrodevre.com/2015/04/15/isin-inceliklerini-sadece-ustasi-bilir/

Kör, topal, kendi çabalarımla elektroniği öğrenmeye çalışırken, bunun pek verimli olmadığını anladım. Eğer bir sahada temelleriniz varsa, yeni bir şeyler öğrenmek kolay. Lakin konu ile alakanız yoksa, sıfırdan bir şeyler öğrenmeye çalışmanız, her zaman kafanızı duvara vurarak, motivasyon kaybetmenize neden olur. Benim içinde bulunduğum durum bu. Bu durumu değiştirmenin tek yolu, bir bilene sormaktır dedim ve online elektronik dersleri almaya başladım.

Bir işin inceliklerini sadece o işin ustası bilebilir. Yazımın başlığı bu. Bu sonuca düz mantıkla varmak kolay. Bir de yaşayarak anlamak var. Bir örnek vereyim.

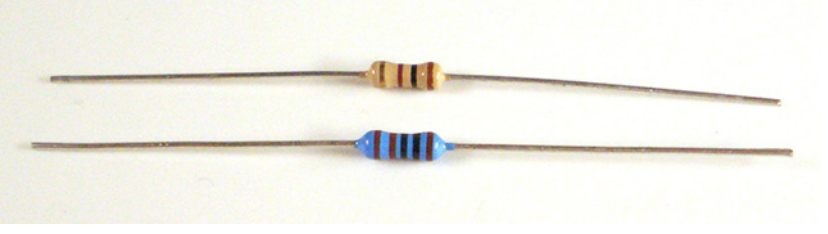
Elimizde 24 volt ile çalışabilen ve 0.4 amper akım ile yanan bir lamba olsun. Bu lambayı 230 volt ile çalıştırabilir miyiz? Bir düzenek kurmadan bu mümkün değil, yani alıp lambayı 230 volta bağlarsak, lamba bozulur. Ama 230 voltluk gerilimi 24 volta düşürebilirsek, o zaman lambayı 230 voltluk düzende çalıştırabiliriz. Yapmamız gereken tek şey, fazlalık olan 206 voltu yok etmek. Bunu örneğin bir direnç kullanarak yapabiliriz. Bu direncin büyüklüğü ne olmalı, şimdi onu hesaplayalım:

```
U = Gerilim
I = Akım
R = Direnç

U = R x I
```

Yok etmek istediğimiz gerilim miktarı 206 volt, lambanın ihtiyaç duyduğu akım ise 0.4 amper. Bu durumda direnç büyüklüğü şöyle olmalıdır:

```
206 = R x 0,4
R = 206 / 0.4 = 515 ohm
```

515 ohm değerine sahip bir direnç 206 voltu ısıya dönüştürebilecek kabiliyetteymiş demek ki. Elektronik ile ugraşanlar, aşağıdaki resimde yer alan dirençleri tanırlar.



Ben de arasıra kurduğum devrelerde kullanıyorum. Eğer böyle bir direnci 206 voltu eritmek için kullanırsanız, şansınız varsa sadece hayal kırıklığına ugrarsınız, şansınız yoksa, eliniz, oranız, buranız yanar, çünkü bu direnç sadece 0.5 watt güce dayanabilir.

Benim dirençler hakkında bilmediğim parametre de buydu. Şimdi 206 voltu bu direnç ile götürmenin neden mümkün olmadığını inceleyelim. Bunun için ne kadar bir gücün söz konusu olduğunu tespit etmemiz gerekiyor.

```
P = Güç
U = Gerilim
I = Akım

P = U x I
```

Hesaplayalım:

```
P = 206 x 0.4 = 82,4 watt.
```

Evet, 206 voltu yok etme işlemi esnasında oluşan güç tam, tamına 82.4 watt. Oturma odanızdaki lambada 80 ya da 100 wattır. Sadece bir kıyas yapabilmeniz için söylüyorum. 82.4 wattı 515 ohm gücünde ama 0,5 watt kaldıran bir dirence verdiğinizde, nasıl bir sonuç alabileceğinizi düşünebilirsiniz. Ortalığı çok kısa zamanda duman kaplar. Denemeyin bence :)

Direncin sahip olması gereken ohm değeri 515 lakin kaldırdığı güç 80-85 watt civarında olması gerekiyor. Böyle bir direnci aşağıdaki resimde görmektesiniz:

İlk dirençle aradaki farkı görebiliyor musunuz? Birisi karınca boyunda, digeri ele sığmayan bir metal parçası. Birisi 10 kuruş, diğeri 30 lira.

Ben bu bilgiyi belki bir kitapta okudum ya da okurdum, belki deneme, yanılma yöntemiyle keşfederdim. Her ikisi de benim için zahmetli bir öğrenim süreci olurdu. Anlayıp, anlayamayacagımın garantisi de olmazdı. Bu yüzden diyorum ki: **bir noktadan başka bir noktaya gitmenin en kısa yolu bir ustayı ziyaret etmekten geçiyormuş**. İşin inceliklerini sadece ustası bilebilir.

Java Şampiyonluğu Nedir ve Nasıl Olunur?

http://www.kurumsaljava.com/2014/11/17/java-sampiyonlugu-nedir-ve-nasil-olunur/

Ben 2009 senesinde Java sampiyonu olarak seçildim. Bilişim sohbetleri söyleşimde Java şampiyonluğu hakkında bilgi vermeye çalıştım. Bu yazımda kısaca Java şampiyonluğunun ne olduğunu tekrarlamak ve nasıl Java şampiyonu olunabileceği konusuna değinmek istiyorum.

Dünya çapında yüz otuza yakın Java şampiyonu bulunuyor. Bu rakam aslında yüz altmış civarında, lakin bazı Java şampiyonları Oracle firması için çalışmaya başladıklarından dolayı, bu unvanı bırakmak zorunda kaldılar.

Java şampiyonları proje sayfasına baktığımızda, Java şampiyonu olabilmek için bir adayda aranan özellikleri görmekteyiz. Bunlar:

- Liderlik vasfı: Java şampiyonları JUG (Java User Group) kurucusu ve yöneticisidirler ya da Java ile ilgili projelere katkıda bulunurlar.
- Teknik bilgi: Java şampiyonları senior seviyesinde yazılımcı ya da mimardırlar. Java platformu hakkında geniş kapsamlı bilgi ve tecrübeye sahiptirler.
- Bağımsız ve güvenilir: Java şampiyonlarının Oracle firmasına bağımlılığı yoktur. Bağımsız bir şekilde fikirlerini beyan ederler. Söyledikleri ciddiye alınır ve topluluk bünyesinde güvenilir şahıslar olarak tanınırlar. Yazarlık ve egitmenlik: Java şampiyonları Java ve diğer yazılım konularında kitap ya da blog yazarlar, konferanslara konuşmacı olarak katılıp, üniversitelerde Java eğitimleri verirler.
- Örnek olma: Java şampiyonları yaptıkları çalışmalar aracılığı (consulting, eğitmenlik, yazarlık ve konferans konuşmacılığı) ile sektör çalışanlarına örnek olurlar.

Türkiye'de bu özelliklere sahip birçok yazılımcı tanıyorum. Bunlardan bir tanesi Mert Çalışkan. Kendisi AnkaraJUG kurucusu ve bir senior Java developer. Daha önce Mert ile birlikte birçok konferansa konuşmacı olarak katıldık. Kendisi "bana Java şampiyonu olmak için sponsor olur musun" diye sorduğunda, tereddüt etmeden evet dedim. Bu şekilde Mert'in Java şampiyonu olma süreci başladı ve kısa bir zaman önce Java şampiyonu olarak seçildi.

Şimdi kısaca bu sürecin nasıl işlediğinden bahsetmek istiyorum. Yeni kurallara göre bir şahsın Java şampiyonu olabilmesi için mevcut bir Java şampiyonun bu şahsa sponsor olması gerekiyor. Sponsorun görevi Java şampiyonu adayını Java şampiyonları topluluğuna tanıtmak ve adaya bu süreçte destek olmak. Sponsor olan Java şampiyonunun bu desteği önemli, çünkü topluluk kendi içlerinden birisinin bu şahsa kefil olduğunu görmek istiyor. Eğer aday gerekli vasıflara sahip ise, yapılan oylama sonucunda adaylığı kabul görüyor ve Java şampiyonu olarak seçiliyor ya da adaylığı reddediliyor. Reddedilmenin başlıca sebebi, aranan vasıfların adayda olmaması. Ama aday olarak gösterilip de, Java şampiyonu seçilmeyeni görmedim.

Java şampiyonluğunu Microsoft firmasının MVP programı ile kıyasladığımızda, bu iki programın farklılıklar taşıdığını görmekteyiz. Java şampiyonu unvanını alan bir şahıs, bu

unvanı Oracle firması için çalışmadığı ve Java alanında aktif kaldığı sürece bir ömür boyu taşıyabilir. MVP ler her sene yeniden seçilmektedirler. MVP ler daha ziyada Microsoft firmasının ürünlerine konsantre olurken, Java şampiyonları Java platformunun tümünü kapsayabilecek çalışmalar yaparlar. 2014 senesi itibariyle dünya çapında 4000 e yakın MVP bulunuyor. Aynı rakam Java şampiyonları için 130 civarında ve bu rakamın 1000 ile sınırlanması planlanıyor. İki program arasındaki en belirgin özellik ise şu: MVP leri doğrudan Microsoft firması seçerken, Java şampiyonlarını bu unvanı taşıyan topluluk üyeleri seçmektedir. Oracle firmasının Java şampiyonu seçimlerinde söz hakkı bulunmamaktadır. Ayrıca Oracle firması için çalışmaya başlayan bir Java şampiyonunun Java şampiyonluğu son bulmaktadır.

Buraya kadar Java şampiyonluğunun ne olduğu konusunda değindim. Yazımın bundan sonraki bölümünde nasıl Java şampiyonu olunur sorusuna cevap vermek istiyorum.

Java şampiyonluğuna giden yol uzun olabilir. Bu yolu çok kısa bir zamanda kat etmenin tek yolu, Java ve yazılım konusunda tutku sahibi olmaktır. Bu ve aşağıdaki özellikler bir araya geldiğinde, Java şampiyonu adayı olmak için fazla bir engel kalmamış demektir.

- Java şampiyonu adayı mutlaka Java konularını işleyen bir blog sahibi olmalıdır. Bu adayın bilgi paylaşımına değer verdiğini gösteren en belirgin özelliktir. Aday düzenli aralıklarla blog yazarak, bilgi paylaşımını istikrarlı bir şekilde sürdürmelidir.
- Aday edindiği tecrübeleri bir kitap haline getirip, yayımlanmasını sağlamalıdır. Bu onun uzmanlık alanına işaret eden ve tecrübe seviyesini gösteren bir durumdur.
- Aday bulunduğu şehirde bir JUG (Java User Group) kurmalı ve düzenli aralıklarla toplantılar düzenleyerek, topluluğa önderlik etmelidir. Bu onun Java'nın tanıtılması ve yayılmasına verdiği önemi gösterir. Eğer bulunduğu şehirde bir JUG varsa, aday yeni toplantıların düzenlenmesine yardımcı olarak, topluluğun büyümesine katkıda bulunmalıdır.
- Aday konferans ve üniversite toplantılarına konuşmacı olarak katılarak, Java'nın tanıtılmasına katkıda bulunmalıdır. Sadece bir toplum önünde bir konuyu sunabilen, bu konu hakkında derin bilgi sahibi olabilir.
- Aday okul ve üniversitelerde seminerler düzenleyerek, ilgi duyanlara Java dilini ve platformunu öğretmelidir. Adayın Java ve yazılım konusunda eğitime verdiği destek, onun bu konuda ne kadar tutku sahibi olduğunun göstergesidir.
- Aday herhangi bir açık kaynaklı Java projesine yazılımcı olarak katkıda bulunmalıdır. Bu onun açık kaynaklı Java projelerine verdiği önemi gösterir. Bu listede yer alan kriterleri yerine getirmek için beş ila on senelik bir çalışma söz konusu olabilir. Kanımca usta bir yazılımcı olmak için gerekli süre de on sene gibi bir zaman dilimidir.

Yazımın başında da belirttiğim gibi, aday olabilmek için bir Java şampiyonunun sponsor olarak adayı desteklemesi gerekmektedir. Şimdi bana yüzlerce "bana da sponsor olur musunuz" e-postası gelmeden önce, benim bir adayda aradığım özelliklere kısaca değinmek istiyorum.

Benim için en önemli kriterden birisi, benim adayı şahsen tanıyor olmamdır. Sadece bu

şekilde aday hakkında bir fikir sahibi olabilirim. Diğer önemli bir kriter, adayın Java ve yazılım konusunda ne kadar üretken olduğu konusudur. Bir adayın üretkenlik seviyesini düzenli olarak yazdığı bloglarda görmek mümkündür. Ayrıca adayın bir ya da birden fazla kitap yazmış olması, adayın yazılımı ne kadar ciddiye aldığının göstergesidir. Diğer önemli bir kriter, adayın hangi yazılım projelerinde yer aldığı ve hangi pozisyonda kariyerine devam ettiğidir. Adayın geniş çaplı yazılım projelerinde yer almış olması, yazılım konusunda geniş tecrübe sahibi olması anlamına gelir. Adayın benlik gütmeyen yazılımcı olması da diğer önemli bir konu.

Java şampiyonu olmak zor değil. Bu konuda istikrarlı bir çalışma gerekiyor. Java şampiyonluğunu üstadlık ya da ustalık olarak görmemek lazım. Java şampiyonları daha ziyada yazılıma tutkuyla bağlı kişilerdir. Bu tutku onların üretken ve topluluğa iyi örnek olmalarını sağlayan en belirgin özellikleridir.

Ben şahsen Türkiye'den çok daha fazla Java şampiyonu çıkmasını istiyorum. Bu konuda üzerime düşeni yapacağım. Aklımda destek vermek istediğim birkaç isim var. Genç yazılımcıların da gelişim süreçlerine katkıda bulunarak, geleceğin Java şampiyonları olmalarını sağlayabilirsek, ne mutlu bize…

Güncelleme (16.4.2015):

Mert'in sponsor ve benim oylamada destek olduğum Murat Yener 5.4.2015 tarahinde Java Champion olarak seçildi. Kendisini tebrik ediyorum.

Neden Kafam Bu Kadar Rahat?

http://www.kurumsaljava.com/2014/11/30/neden-kafam-bu-kadar-rahat/

Kişisel Gelişim başlıklı yazımda kullandığım metot ve araçlardan bahsetmiştim. Bu yazımda çok faydalı bulduğum bir aracı daha tanıtmak istiyorum.

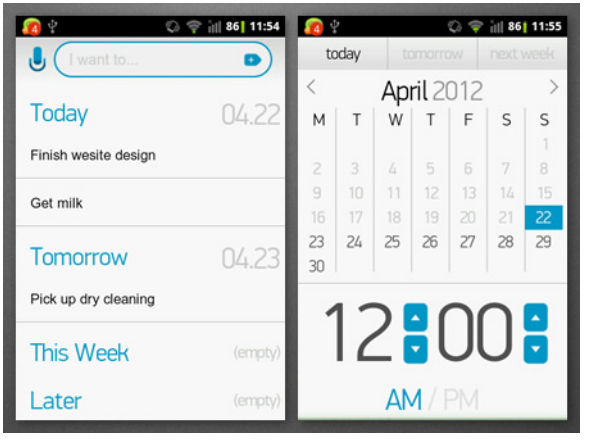
Yazının devamını okumadan önce sizden şunu yapmanızı rica ediyorum: Kağıt kalem kullanarak ya da bilgisayarınızda bir editör açarak yapmanız gereken şeylerin listesini çıkarın. Bu liste içinde mesai bitiminde sonra bakkaldan alacağınız ekmekten, iki hafta sonraki diş hekimi randevunuza kadar her şey yer alabilir. Önemli olan aklınızda tutmaya çalıştıklarınızı listeye dökmenizdir. Siz listenizi oluştururken, ben bekliyor olacağım…..

Listenizi görmedim, lakin listenizde en az 5, en kötü ihtimalle 20 başlığın olduğunu tahmin edebiliyorum. Aynı şeyi ben de üç gün önce tekrar yaptım ve kafamda 15 değişik şeyin yapılmak üzere beklediğini gördüm ve şok oldum.

Her insan kafasında ortalama 10-15 şeyle gün içinde yapması gereken işlerin üstesinden gelmeye çalışıyor. Bunun ne kadar verimsiz bir girişim olduğunu kendi edindiğim tecrübelerden biliyorum. Kafaların böyle yapılacak işlerle yüklü olması, verimli bir şekilde bir işe odaklanılmasının önündeki en büyük engellerden birisi. Çoğu zaman nedense akla gelen daha önemli bir işten dolayı üzerinde çalışılan iş yarıda bırakılıyor. Bunun tek sebebini kafada yapılacak işlerin cirit atması olarak görüyorum.

Her aktiviteyi akılda tutmak, ne zaman neyin yapılması gerektiğini organize etmeye çalışmak beyin için çok yorucu bir işlem. Beyin devamlı böyle yarım kalmış aktivitelerle haşır neşir ve onların nasıl üstesinden geleceğinin planını yapıyor. Bundan kurtulmanın ve beyni rahatlatmanın en verimli yöntemi, yapılacak işlerin bir listesini oluşturmak ve beyinden bu organizasyon yükünü uzaklaştırmak.

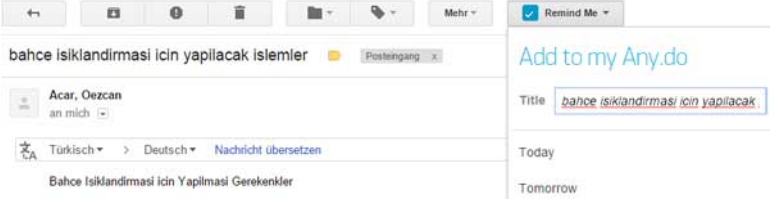
Ben bu işlem için daha önce Kanban panonsu kullanıyordum. Geçenlerde Any.do isminde yeni bir uygulama keşfettim. Bu yazımda bu uygulamayı size tanıtmak istiyorum.



Any.do hem web üzerinden hem de akıllı telefonda app olarak kullanabileceğiniz basit bir “yapılacak şeylerin listesi” uygulaması. Oluşturduğum listeye internet erişimi olan herhangi bir bilgisayardan ulaşabiliyorum. Bu şekilde listemi bir bilgisayardan başka bir bilgisayara taşımam gerekmiyor. Özellikle cep telefonları için hazırlamış uygulamanın çok faydalı olduğunu gördüm. Kanban panosunu kullanırken bunun eksikliğini hissetmiştim. Cep telefonumda bir app olmadığı için kanban panomda olup, bitenleri takip etmem zor oluyordu. Gözden ırak olan, gönülden ırak olur misali bir zaman sonra kanban panosunun bir kopyasını kafamda oluşturup, işlerimi yine eski yöntemlerimle takip etmeye çalışıyordum. Taaki Any.do ile tanışana kadar.

Aklıma yapmam gereken bir şeyler geldiğinde, hemen cep telefonum aracılığı ile aktivite listemde yeni bir kayıt oluşturuyorum. Cep telefonum devamlı yanımda olduğu için olup, bitenleri takim etmem kolaylaşıyor. Any.do gerekli uyarıları yaparak, işleri takip etmemi kolaylaştırıyor.

Any.do nun birde Chrom eklentisi mevcut. Özellikle gelen e-postaların başlıklarını yapılacak iş listesine kaydetmek birçok yazışmadan dogan işleri takip etmeyi çok kolaylaştırıyor.



Aklımda artık yapmam gereken şeyler konusunda hiçbir kayıt yok. Yapmam gereken işlerin izini beynim değil, cep telefonum ve kullandığım bilgisayarlar sürüyor. “Kafam çok rahat“ sözünün tam olarak ne anlama geldiğini şimdi anlamış oldum :)

Kendinize bir iyilik yapmak istiyorsanız, Any.do uygulamasana bir göz atmanızı tavsiye ederim. Bu konudaki tecrübelerinizi bizimle paylaşın.

Java Dilinde Neden Göstergeçler (Pointer) Yok?

Java'da göstergeçler var, ama C dilinde olduğu şekilde işlemiyorlar. Bu yazımda Java'da göstergeçlerin neden olmadığını aktarmaya çalışacağım.

C dilinde aşağıdaki şekilde bir göstergeç tanımlaması yapilabilmektedir:

```
int  i, *ptr;
ptr = &i;
*ptr=10;
```

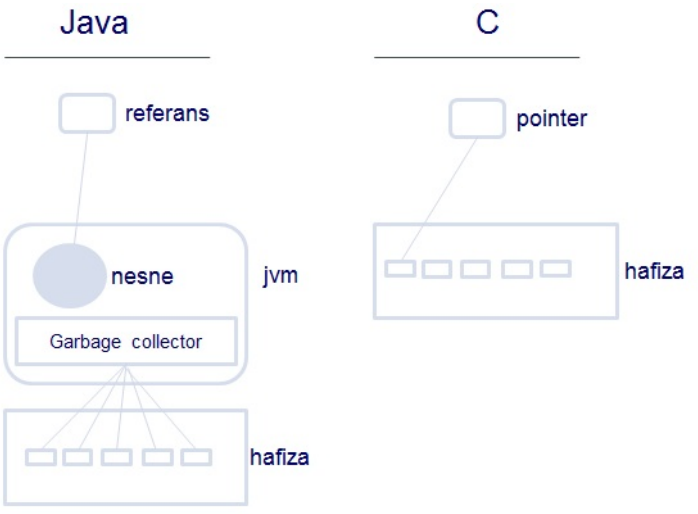
Bu örnekte i isminde ve int veri tipinde bir değişken ve yine int veri tipine sahip ve ptr isminde bir göstergeç tanımladık. C dilinde göstergeçler bir hafıza alanına işaret ederler. Yukarıdaki örnekte ptr in sahip olduğu değer i değişkeninin hafızadaki adresidir. ptr üzerinde yapılan her işlem i değişkenini etkileyebilir, çünkü ptr dolaylı olarak i değişkeninin sahip olduğu değere işaret etmektedir. Göstergeç aritmetiği sayesinde hafıza alanlarını doğrudan adreslemek ve o hafıza alanlarında yer alan değerler üzerinde işlem yapmak mümkündür.

Java'da göstergeç tanımlamaları ve hafıza alanları üzerinden doğrudan işlem yapmak mümkün değildir, çünkü:

- Java sanal makinesi (Java Virtual Machine) hafıza alanını kendisi yönetir.
- Java'da array ve list nesnelerinin uzunlukları JVM tarafından kontrol edilir. Olmayan array elemanları üzerinde işlem yapmaya çalışmak ArrayIndexOutOfBoundsException hatasını oluşturur.
- Java dilinde array dilin bir parçasıdır ve string işlemleri için String sınıfı kullanılır. C dilinde göstergeçler aracılığı ile bu yapılar oluşturulur ve dilin bir parçası değildirler.
- Java nesne referanslarını JVM bünyesinde yer alan garbage collector aracılığı ile yönetir.

Java dilinde göstergeçler sadece nesnelere işaret eden değişken isimleri olarak kullanılırlar, ama:

- Bu değişkenler üzerinde göstergeç aritmetiği yapılamaz
- Sadece nesnelere işaret ederler, adres alanlarına değil



Java'da bu nesne göstergeçlerine referans ismi verilmektedir.

Aşağıdaki Java kodunu koşturduğunuzda NullPointerExceptin oluşmaktadır:

```
Nesne nesne = null;
nesne.birMethod();
```

Peki neden oluşan hatanın ismi NullPointerException seklindedir de NullReferenceException değildir? Bu sorunun cevabını bende bilmiyorum. Java'da NullReferenceException olmalıydı. NullPointerException sınıfının kaynak koduna baktığımızda, sınıfın kim tarafından geliştirildiği görememekle birlikte, JDK1.0 dan beri Java'nın bir parçası olduğunu görmekteyiz. Bu sanırım Java dilini oluşturanların C/C++ kökenli olmalarından kaynaklanıyor. Ama C# dilinde NullReferenceException isminde ve aynı vazifeyi gören bir sınıf mevcut. C# dil geliştiricileri aynı hatayı yapmamış.

```
/**
 * @author  unascribed
 * @version %I%, %G%
 * @since   JDK1.0
 */
public
class NullPointerException extends RuntimeException {
}
```

Şüphesiz Java dilinde göstergeçlerin ve göstergeç aritmetiğinin olmaması Java'da geliştirilen kodun daha kolay anlaşılır olmasını sağlamaktadır. C kodu çok kısa zamanda göstergeçlerin yerli, yersiz kullanılmalarından dolayı kaotik bir yapıya bürünebilir. Ama bunun karşılığında göstergeçler yardımı ile donanıma çok yakın kalınarak, uygulama geliştirmek mümkündür. Java'da bu ne yazık ki mümkün değil. Sizinle donanım arasında her zaman JVM olacak.

20 Yaş, 30 Yaş, 35 Yaş ve Bugünki Programcı Ben

http://www.kurumsaljava.com/2015/02/06/20-yas-30-yas-35-yas-ve-bugunki-programci-ben/

Bu yazıyı kaleme alırken Can Yücel'in o güzel yazısından esinlendim. Onun yaptığı gibi yirmi, otuz ve otuz beş yaşında olan benleri coding dojo ya davet ettim. Maksat biraz yazılım üzerine sohbet etmekti. Ortaya çıkan bu oldu.

Birlikte kod kata yapalım, eşli programlama süper dedim.
35 yaşımdaki ben iyi olur, ben varım dedi.
30 yaşımdaki ben, ilgi alanıma giriyor, öğrenmek isterim dedi.
20 yaşımdaki ben, ilgimi çekmiyor, oyun oynayabilir miyiz diye sordu.

Eşleşdik ve birlikte stack katayı yapmaya başladık.
35 yaşımdaki ben, 20 ve 30 yaşındaki benlerin yazdığı kodları hiç beğenmedi. Kendisi interface bazlı programlamayı ve temiz kod (clean code) yazmayı yeğliyormuş. Baktım kimse test yazmamış. Test güdümlü yazılım nasıl yapılır, gösterdim. 35 yaşımdaki ben, biliyorum, ama her zaman mümkün değil işte dedi. Her zaman mümkün olduğunu göstermeye çalıştım. 30 yaşındaki ben, uygulama kodundan daha çok test kodu yazdın diye bana çıkıştı. Müşterinin bunu istediğinden emin değilim dedi. 20 ve 30 yaşındaki benler kim daha uzun metot yazar yarışına girdiler. 35 yaşındaki ben onların yazdıkları kodları yeniden yapılandırma metotlarıyla okunur hale getirdi. 20 ve 30 yaşındaki benlerin yeni kodu okuduklarından ağızları açık kaldı.

Öğle arası verdik, laflıyoruz. 20 yaşındaki ben sabahlara kadar kod yazdığını, bundan büyük zevk aldığını söyledi. Düzenli bir hayatının olmadığı belliydi. 30 yaşındaki ben çok oturmaktan dolayı sırt ağrılarından şikayet etmeye başladı. 35 yaşındaki ben sırt agrılarına öğle arası yürüyüşlerinin iyi geldiğini, yürüyüşleri düzenli olarak yaptığını söyledi. Bana da düzenli olarak spor yapıp, yediğime, içtiğime, düzenli uyumaya dikkat ettigimi söylemek kaldı. 20 sene önce bilgisayar başında sabahlardım. Bunun yaşlanınca sürdürülebilir bir durum olmadığını, mesai saatlerinde verimli olmak için işin ve özel hayatın ayrılması gerektiğini belirttim. Sağlam kafanın sağlam vücutta olduğunun bilincine varmıştım.

20 yaşındaki ben en kral dilin Java olduğunu iddia etti. Yeni, yeni öğrenmeye başlamıştı. Başka bir dil bilgiği de yoktu. 30 yaşındaki ben de işi Java dili ile götürüyorum dedi. Başka bir dil öğrenmenin gerekli olduğunu düşünmüyorum dedi. 35 yaşındaki ben ikisine de kızdı. İyi bir programcı olmak için birden fazla dili bilmek lazımdır dedi. Ben 35 yaşındaki bana katıldığımı söyledim. 20 yaşındaki ben çok biliyorsunuz dedi. Ölene kadar Java ile program yazabileceğimi size göstereceğim dedi. Bundan bu kadar emin olma dedim. İlgi alanların değişebilir.

20 yaşındaki ben kitaplığımdaki kitapları ve masamdaki düzenekleri görünce, sen programcı mısın, yoksa elektronikçi misin diye sordu? Bu sorunun cevabını 20 sene sonra alacaksın dedim. O kendisinin programcı olduğunu ve öyle kalacağını söyledi. Kod yazmaktan başka hiçbir şeyin onu ilgilendirmediğini söyledi. İyi bir programcı olabilmek için programcılık haricinde ilgi alanları geliştirmesi gerektiğini henüz bilmiyordu.

Kim donanımdan anlar diye sordum. 30 yaşındaki ben, işim olmaz, ben programcıyım dedi. 35 yaşımdaki benin bu konuda fikri değişikti. İyi bir programcı olabilmek için donanımdan ve

özellikle mikro işlemcilerin içinde olup, bitenlerden haberdar olunmalı fikrindeydi, ama nereden başlayacağı konusunda fikri yoktu. Onlara mikro denetleyicilere göz atmaları tavsiyesinde bulundum. İçlerinde elektroniğe karşı bir ilgi varsa, micro denetleyiciler onlara çok başka bir dünyanın, programcılığı da içine alan bir dünyanın kapılarını açacaktı. İçeri girip, girmeyeceklerine kendilerine kadar vereceklerdi, ama şimdi en azından kapının nerede olduğunu biliyorlardı.

35 yaşım kendisini mühendis gibi değil, daha çok zanaatkar gibi gördüğünü söyledi. Bir usta-çırak ilişkisinden gelmemenin ezikliğini çektiğini söyledi. 30 yaşım ben bilgisayar mühendisiyim, bir mühendisim, boşuna okumadım ben dedi. 20 yaşım ben bilgisayar mühendisi olmak istiyorum dedi. İyi bir maaş alabilmek ve kariyer yapabilmek için bu şart dedi. Yazılım hem mühendislik, hem zanaat, çözüm üreterek mühendis, çözümü koda dökerek zanaatkar oluyoruz dedim. Programcı olarak usta-çırak geleneğinden gelmediğimiz için zanaatkar tarafımızın eksik olduğunu, bu yüzden kötü programlar yazdığımızı, ama pratik yaparak, bu sorunun üstesinden gelebilecegimizi söyledim.

Programcı yeteneklerinizi nasıl geliştiyorsunuz diye sordum. 35 yaşım kod katalarını keşfettiğini ve bu şekilde mesai dışında pratik yaptığını söyledi. 30 yaşım mesai çerçevesindeki aktivitelerinin iyi bir programcı olmak için yeterli olduğu görüşündeydi. Katalardan ve pratik yapmaktan bihaberdi. Kodkata.com a bakmalarını tavsiye ettim. Pratik yapmanın daha iyi bir programcı olmak için gerekli olduğunu biliyordum.

35 yaşıma diğer ikisini nasıl bulduğunu sordum. Çaylak bunlar, bir şeyden anlamıyorlar dedi. Kendisini nasıl bulduğumu sordu. Olacak, doğru yoldasın, ilgi alanlarını çoğaltmaya bak, her şey program yazmak değil dedim. İyi bir programcı olabilmek için edebiyat kitapları okumanın, kendi ana diline hakim olmanın, elektronikte uğraşmanın, spor yapmanın ve bunun gibi programcılıkla ilgisi olmayan birçok aktivitenin gerekli olduğunu anlamaya başlamıştım. Bunları 20 ve 30 yaşındaki benlere anlatmanın faydası olmayacağını biliyordum, çünkü onlar tek bir programlama dilinin insanlarıydı.

Ben ne düşünüyordum onlar hakkında;

35 yaşım yazılımda bir üst seviyeye çıkmak için çaba sarfediyor, ama lazım gelen kapıları keşfetmekte zorlanıyordu. Ama programcılığın diğer yönleriyle de ilgilenmeye başlamıştı. Er ya da geç o kapıları keşfedecekti. Bunun için çok çalışması gerektiğini biliyor muydu?

30 yaşım yazılımın sadece kod yazmaktan ibaret olduğunu zannediyordu. Tek bir dille işi götürebileceğini düşünmesi, onu sadece uzmanlığa mı götürecekti?

20 yaşım programcılığa en uzak mesafede olandı. Bu kafayla iyi bir programcı olabilecek miydi?

Peki kendi hakkımda ne düşünüyorum;

Bildiğim bir şey varsa o da hiç bir şey bilmediğimdir (Sokrates).

Çok değişik düşüncelere sahip dört insan bir araya gelmişti. Programcılığın ne olduğu konusunda herkesin fikri farklıydı. Onları çağırdığıma pişman mı olmuştum. Belki de….

Bende kabahat.

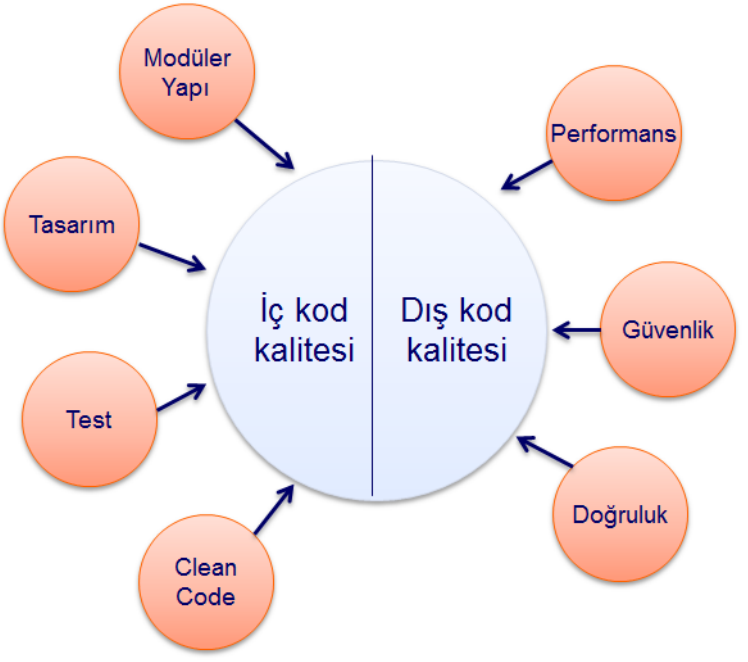
Ne çağırıyorsun tanımadığın adamları evine ...

(Not: Son iki cümle Can Yücel ustanın şiirinden alıntıdır.)

Kod Kalitesi Denince Akla Gelenler

http://www.pratikprogramci.com/2015/04/14/kod-kalitesi-denince-akla-gelenler/

Son zamanlarda temiz kod (clean code) teriminin kod kalitesi terimi ile eş anlamlı kullanıldığına şahit olmaktayız. Kod kalitesini sadece tek bir boyuta indirgemenin doğru olmadığını düşünüyorum. Temiz kod ya da testler çok boyutlu olan kod kalitesinin sadece iki boyutunu oluşturmaktadırlar. Kod kalitesinin görebildiğim boyutlarını aşağıdaki resimde bir araya getirmeye çalıştım.



Kod kalitesini iç ve dış kod kalitesi olarak ikiye ayırabiliriz. İç kod kalitesi daha ziyada yazılımcı ve testçilerin doğrudan şahit oldukları ve daha da iyi olması için çaba sarf etmeleri gereken objektif olgudur. Buna karşın dış kod kalitesi kullanıcı ve müsterinin uygulamayı kullanırken sahip oldukları subjektif hislerden oluşan algıdır. Burada subjektif kelimesini kullandım, çünkü kullanıcıdan kullanıcıya kalite algısı sahip olunan kalite tanımına göre değişmektedir.

Resimde yer alan ve kod kalitesini oluşturan boyutları şu şekilde tanımlayabiliriz.

Testler

Kod kalitesini ölçülebilir hale getiren faktörlerin başında yazılım testleri gelmektedir. Yazılım testleri olmadan uygulamayı müşteri gereksinimleri doğrultusunda yeniden şekillendirmek ya da

geliştirmek mümkün değildir.

Bir yazılım sisteminin kalitesi söz konusu olduğunda, sistemin müşteri gereksinimlerini ne oranda tatmin ettiğine bakmak gerekmektedir. Uygulama ne kadar çok müşteri gereksinimine cevap verebiliyorsa, o oranda dışarıya doğru kaliteli olma imajına bürünür. Bu imajın sürdürülebilir olmasını sağlamak için uygulamanın yeni müşteri gereksinimlerine cevap verebilecek yapıda kalmasını sağlamak gerekmektedir. Bunun tek yolu refactoring yöntemi ile uygulamanın yeni gereksinimler doğrultusunda yoğrulabilir olmasından geçmektedir. Refactoring işlemi için testleri varlığı mecburidir. Testlerin olmadığı yerde rafactoring işleminin güvenli bir şekilde yapılması mümkün değildir. Bu yüzden kalitenin en önemli boyutlarından birisini ya da kalitenin temelini uygulama testleri oluşturmaktadır. Testlerin olmadığı bir uygulamada uzun vadede kaliteden ödün vermemek çok zor hale gelmektedir.

Kodun test edilebilir yapıda kalması önem taşımaktadır. Kodun test edilebilir yapıda olması yanı sıra testlerin çeşitliği de gereklidir. Birim, entegrasyon ve onay/kabul test türleri ile uygulamayı çok değişik perspektiflerden test etmek mümkündür. Uygulamanın hangi tür testler ve ne oranda test edilmesi gerektiğine ekip karar vermelidir.

Temiz Kod

Okunabilir ve kısa kod birimlerinden oluşan bir uygulamayı anlamak ve geliştirmek kolaydır. Lakin testlerin yeterli olmadığı bir ortamda kodun ne kadar temiz yazıldığı önem taşımamaktadır, çünkü testler olmadan uygulamayı yeniden yapılandırmak mümkün değildir. Uygulamayı yeniden yapılandırma gerekliliği her zaman müşterinin yeni isteklerinden doğduğundan, kodun temiz yazılmış olması yanı sıra test edilebilir ve test edilmiş durumda olmasına dikkat edilmelidir. Sadece test edilmiş temiz kod uygulamanın geliştirilebilir olmasını mümkün kılabilir.

Tasarım

Tasarımı uygulamanın temeli, şekli ve işleyiş tarzı olarak tanımlayabiliriz. Yeni müşteri gereksinimleri tasarımın adapte edilmesini zorunlu kılabilir. Yeni müşteri gereksinimleri ile adapte edilmeyen uygulama tasarımı zaman içinde sıkı bir korzet haline geldiğinden, yeni müşteri isteklerinin gelecekte implemente edilmesi zorlaşır. Bu sebepten dolayı esnek bir tasarımın oluşturulması ve korunması zaruridir. Esnek bir tasarım oluşturmak için SOLID tasarım prensipleri uygulanabilir. Tasarım prensiplerinin hepsini önemli bulmakla birlikte, tek sorumluluk prensibinin nesneye yönelik programlama teknikleri kullanıldığında, kendi içinde bir bütün ve kırılgan olmayan kod birimlerinin oluşmasını sağladığı söylenebilir.

İyi bir tasarımı iyi kılan uygulanmış olan tasarım prebsipleri değil, uygulamanın sahip olduğu testlerdir. Testler olmadan mevcut yapının adapte edilmesi mümkün değildir. Bu yüzden kalitenin bu boyutu diğer bir boyutu olan testlere dogrudan bağımlıdır. Testler olmadan sürdürülebilir bir tasarımın var olma şansı düşüktür.

Modüler Yapı

Bir modülü tek bir işten sorumlu, test edilmiş ve etrafı ile iletişimi kendi bünyesinde tanımlı olan arayüzler aracılığı ile gerçekleştiren kod birimi olarak tanımalayabiliriz.

Modüllerden oluşan bir uygulamada genel tasarım esnek kalma eğilimi gösterir. Bu uygulamanın geliştirilebilir olma özelliğini koruması anlamına gelmektedir. Uygulamanın hızla yeni müşteri gerensinimlerine adapte edilebilmesi dışarıya kalite artışı olarak yansır.

Performans

Kullanıcı perspektifinden bakıldığında, bir uygulamanın kalitesini tanımlayan en belirgin faktörlerin başında uygulamanın cevap verme hızı gelir. Tasarımın iyi, testlerin yeni gereksinimleri implemente etmek için yeterli, kodun temiz olması, performansın neden beklenenin altında olduğunu kullanıcıya anlatmak için yeterli argüman değildirler. Bu sebepten dolayı kullanıcı gözünde kaliteli bir uygulama kullandığı hissini uyandırabilmek için uygulama performansının hak ettiği düzeyde olmasına dikkat edilmelidir. Bu amaçla uygulama geliştiricileri performans ve yük (load) testleri yapabilirler.

Güvenlik

Giriş yaptıktan sonra kendi kullanıcı ismi yerine başka bir şahsın ismini gören kullanıcıya uygulamanın kaliteli olduğu hissini vermenin tek yolu, bu hatayı gidermektir. Uygulamanın ihtiyaç duyduğu güvenlik çemberini oluşturma görevi yazılım ekibine düşmektedir. Güvenliği sağlamak için tasarımın bu aspekt göz önünde bulundurularak oluşturulması gerekmektedir. Yeni müşteri gereksinimleri ile adapte edilen tasarım güvenlik tedbirlerini kapsama özelliğini de yitirmemelidir. Bu dışarıya kalite kaybı olarak yansır.

Doğruluk

Yapılan işlemin doğruluğu, kullanıcıların sahip oldukları kalite algısı ile doğru orantılıdır. Beklenenin dışında bir işlem sonucunun alınması, uygulamaya kullanıcılar tarafından kalitesiz yaftasının yapıştırılması için çok geçerli bir nedendir. Uygulama iç kod kalitesi bakımından çok büyük eforlar sarf edilerek hazırlanmış olsa bile, gün sonunda kalite seviyesini yapılan işlemlerin doğruluğu belirler. Uygulamanın doğru çalıştığını garanti etmek için yazılım testlerine ihtiyaç duyulmaktadır. Testler sayesinde işletme mantığı üzerinde istenmeden yapılan değişiklikler ve buradan doğacak yanlış işlem sonuçları kısa sürece tespit edilebilirler.

Yazımda kod kalitesini koruyabilmek için kullanabilecek bazı araçlardan bahsettim. Bunların başında şüphesiz yazılım testleri geliyor. Sürekli entegrasyon (continuous integration) aktiviteleri bünyesinde otomatik koşturulabilen yazılım testleri sayesinde hızlı geri bildirim döngüsü oluşturulabilir. Bu döngünün varlığı kod kalitesinin ne seviyede olduğu bilgisinin her daim gözler önünde olmasını sağlayacaktır.

Kalitenin ne seviyede olduğunu tespit edici geri bildirim döngülerinin olmaması, gözden ırak, gönülden ırak misali kalite olgusunun ekip tarafından göz ardı edilmesine sebep verecektir. Bunu bir nebze önleyebilmek için kod inceleme (code review) seanslarının yapılması, Sonar gibi statik kod analizi yapan araçların kullanılması ve başlama ve bitirme kriterlerinin kod kalitesini göz önünde bulunduracak şekilde yapılandırılması gerekmektedir.

Kod Redaktörlügü

http://www.pratikprogramci.com/2015/03/18/kod-redaktorlugu/

Programcılar yazar olsalardı keşke! yazımı bugün tekrar okudum ve bir şey aklıma geldi; Bunu sizinle paylaşmak istedim.

Kod kalitesi deyince akla hemen yazılım testleri gelir. Kod kalitesinde madalyonun bir yüzü testler ise, diğer yüzü kodun okunabilirlik seviyesidir. İkisi bir araya geldiğinde müşteri gereksinimleri doğrultusunda hamur gibi yoğrulan uygulamalar geliştirilebilir.

Programcı olarak hepimizin ortak yani, yazdığımız kodun zaman içinde okunamaz hale gelmesidir. Bunun birçok nedeni var. Nedenlerin hepsini çok iyi biliyoruz, ama ne yazık ki kötü kod yazmaktan da vazgeçemiyoruz, çünkü artık bu bir alışkanlık haline gelmiş. Bu durumun kolay, kolay da değişeceğini zannetmiyorum.

Programcılar yazar olsalardı keşke! başlıklı yazımda programcıları kitap yazarlı ile kıyaslamıştım. Şimdi tekrar düşünüyorum da, belki bir kitabın oluşma sürecini örnek alarak, daha okunur kod yazmanın yollarını bulabiliriz, şöyle ki ...

Bir kitap kitap olarak piyasaya sürülmeden önce kalite kontrolünden geçer. Bunu kitabın içeriğini kontrol eden redaktörler yapar. Redaktörler imla hatalarını düzeltirler, içeriği anlaşılır hale getirir ve yazarın konu dışına taşmasını engellerler. Redaktörler yazarlardan bağımsızdırlar. Genelde kitabın çıkacağı yayınevinde çalışırlar ya da hatta bu yayınevinden bile bağımsızdırlar. Bu bağımsızlık onların yazarın gözünün yaşına bakmayacak derecede gerekeni yapmalarını kolaylaştırır.

Şimdi kendimizi yazar olarak düşünecek olursak, yazdıklarımızı kontrol eden ve bizi hizaya sokan redaktörlerden yoksun olduğumuzu görebiliriz. Bakın, her yazılım ekibinde mutlaka testciler, gereksinim analizcileri, proje yöneticileri vs. vardır ama kodu gözden geçiren bir redaktör yoktur. Bunun çesitli sebepleri var. En önemli sebep, bizlerin progrmacı olarak kodun incelenmesine verdigimiz değer ve bu konuya bakış tarzımızdır. Programcı olarak bu konuya değişik şekillerde bakarız. Örneğin:

- Yazdığımız kodun başkaları tarafından incelenmesi ve düzeltilmesini gerekli görmeyiz, çünkü yazdığımız kod olması gerektiği şekildedir.
- Kod incele seanslarını (code review) angarya olarak görürüz, çünkü en iyi kodu yazdığımızdan böyle şeyler gereksizdir.
- Kod inceleme seanslarının önemine inanırız, ama bir kod incele seansında kritize edilmeyi kaldıramayız. Bu yüzden kod incele seanslarından mümkün mertebe uzak dururuz.

Klasik kod incele seansları çok verimsiz geçer, çünkü bir tarafta “aman ha, karşı tarafı yanlış bir şey söyleyerek kırmayayım” diğer tarafta “ne olursa olsun, yazdıklarımın arkasında durup, savunmalıyım, değiştirilmesine izin verememem!” prensibine göre uygulanırlar. Mevzu burada karşı tarafı kırmamak değil, kodu kırılmayacak hale getirmektir. Ama bunun anlaşılması çogu zaman kolay değildir, çünkü programcılar arasındaki kanka kültürü, karşı tarafa yüklenememe yetersizliği ve koda gereğinden fazla sahip çıkma içgüdüsü kod inceleme işini verimsiz ve

gereksiz kılar.

Aynı ekip içinde çalışan ve birlikte kod yazan programcıların birbirlerini kod incele seanslarında kontrol etmeleri mümkün değildir, çünkü bunun için yeterli otoriteye sahip değildirler. İstenilen verimin alınabilmesi için ekipten bağımsız bir redaktörün bu işi eline alması gerekmektedir. Kod inceleme seanslarında "redaktör ne derse o geçerlidir, o yapılır" prensibi geçerli olmalıdır. Redaktörün okey vermediği kod birimleri uygulamanın bir parçası haline gelemez ve geri çevrilir.

Evet biliyorum, kod redaktörlerinden faydalanmak uçuk bir fikir, ama artık başımızı kaldırıp, bazı sorunları çözmek için yeni çözümlerin peşine düşmemiz gerekiyor. Kod redaktörlüğü de böyle bir çözüm olabilir. İlla sadece kod radaktörlüğü yapacak birisinin işe alınması gerekmiyor. Ekip içinde yeterli otoriteye ve bilgiye sahip birisi bu işi yapabilir, örneğin ekip lideri. Ama bu neye benziyor biliyor musunuz? Bu yayınevi patronunun piyasaya süreceği kitapları kendisinin kontrol etmesi gibi bir şey. Ekip liderinin de kendine has sorumlulukları ve çalışma alanları var. Ekip liderine kod redaktörlüğü sorumluluğunu da vermek ne kadar doğru olur, bilemiyorum. Bana doğru gibi gelmiyor. Nasıl Scrum'da bir Scrummaster, projede bir proje yöneticisi, sürümleri oluşturan sürüm yöneticisi varsa, proje bünyesinde kod kalitesini kontrol eden bir kod redaktörü de olmalı. Biliyorum, böyle bir pozisyon yine lüks olarak algılanacak, ama o zaman proje battı, kalite yerlerde serzenişlerini de hazırlıklı olunmalı.

Kod redaktörü ekip içinde huzursuzluğa neden olabilir. Programcılar sadece kod redaktörünü tatmin edebilmek için kod yazmaya başlayabilirler. Ama istediğimizde zaten tam da bu değil mi? Programcılar kafalarına göre takılmasınlar, belli standartlarda kod yazsınlar ve bunun kontrolcüsü ve bekçisi de kod redaktörü olsun. Patronunun söylediklerini yapmakta zorlanmayan bir programcının moralinin kod redaktörünün itirazları sonucu bozulacağını bir safsata olarak görüyorum. Nihayetinde programcı olarak amacımız ileriye gitmek ve daha iyi kod yazabilmek ise, birilerinin yaptığımız yanlışları düzeltmesinde ne gibi bir sakınca olabilir? Bu durumdan sadece benlik güden programcıların şikayet edeceği malum!

Kod redaktörünün sorumluluğu çok büyük. Bu yüzden yazılım yapma konusunda da çok büyük bir bilgi birikimine sahip olması gerekiyor. Bu birikim yazılım mimarilerinden tasarım şablonlarına, tasarım prensiplerinden çevik yazılım yöntemlerinin uygulanışına kadar çok geniş bir yelpazeyi kapsamak zorundadır. Bu durumda en az son on yılını aktif programcı olarak geçirmiş bir şahsın kod redaktörlüğü görevini layıkıyla yerine getirebileceğini düşünüyorum. Kod redaktörü de aslında bir programcı olduğu için çalışma zamanının belli bir kısmını ekip içinde programcılığa, belli bir kısmını da kod redaktörlüğüne ayırabilir. Bu durumda sadece kod redaktörü rolüne sahip bir programcıyı işe almak yerine, tecrübeli bir yazılımcıdan faydalanılabilir.

Er ya da geç her projede kod birimleri okunamaz hale geliyor, çünkü birden fazla programcı kontrolsüz bir şekilde kod birimlerini değiştiriyorlar. Projeler her zaman hızlı ilerlemeli filozofisiyle yönetiliyor. Okunurluk seviyesi düşük bir proje benzini bitmiş bir araba gibidir. Ne kadar o arabayı itmeye çalışsanız da, benzin koymadığınız sürece hızlı ilerleme özelliğinden faydalanamazsınız. Aynı şekilde okunurluk seviyesi düşük kodun da çok hızlı bir şekilde değiştirilmesi mümkün olmadığından, projenin hızlı ilerlemesi belli bir zaman sonra imkansız

hale gelebilir. Yazımın başında da belirttigim gibi, hızlı ilerleyebilmek için madalyonun iki yüzünü de göz önünde bulundurmamız gerekiyor.

Programcıların kodun okunurluğuna çok önem vermedikleri bir dünyada kod redaktörlerine ihtiyaç büyük. Önümüzdeki zamanlarda projelerde böyle bir rolün olusması ümidiyle...

Yazılımda Otopilot

http://www.pratikprogramci.com/2015/02/27/yazilimda-otopilot/

Akıllı ev projem çerçevesinde lavabolara sensörlü musluklar taktım. Doğal olarak kısa bir zaman sonra, musluğu açmadan, sadece ellerimi musluğun altına tutatak, el yıkama işlemi beynimde rutinleşti. Evin içinde olduğum sürece bu bir sorun teşkil etmiyor. Lakin iş yerinde iken ellerimi lavaboda yıkamak istediğimde, garip bir durumla karşılaşıyorum ve beynimin bu durumu anlaması saniyeler alıyor. Öylece saniyeler boyunca musluğu açmadan, suyun akmasını bekliyorum, ta ki evde olmadığımı anlayana kadar! Bu gerçekten garip bir durum.

Beyin sürekli işlemleri rutin hale getirmeye, yani onları otomatize etmeye çalışıyor. Bu otomasyon sayesinde günlük işlerimizin üstesinde kolaylıkla gelebiliyoruz, çünkü ne yapmamız gerektiğini uzunca düşünmemiz gerekmiyor. Bu bir nevi otopilot. Bu rutinleşme olmadan örneğin mutfağa gidip, bir bardak su içmek bile bizim için tahammülü zor bir prosedür haline gelebilirdi. Düşünsenize, su içmek istediğinizde her defasında mutfağın nerede olduğunu, sizin evin neresinde olduğunuzu, bulunduğunuz yerden mutfağa gitmek için gerekli yol planlamasını nasıl yapacağınızı, bardağı bulup, suyu bardağa nasıl koyacağınızı ve suyu içtikten sonra tekrar geriye nasıl döneceğinizi her defasında düşünmek zorunda kalmanız, beyninizi müthiş derecede zorlardı. Mutfağa giderek, bir bardak suyun nasıl içildiğini daha önce en az bir kere yaptığımız ve bu işlemin nasıl yapıldığını öğrendiğimiz için, daha sonraki su içme seanslarını düşünmeden gerçekleştirebiliyoruz. Bunun beynin otopilota bağlaması neticesinde başarabiliyoruz.

Yazılımda da bu söz konusu. Program yazarken yapmamız gereken işlemleri çok fazla düşünmek zorunda kalmadan beynimizde yer alan rutinler aracılığı ile gerçekleştiriyoruz. Beynimizde yer alan bu rutinlerin çoğu bazı problemleri çözmek için ilk öğrendiğimiz bilgilerden ve uyguladığımız ilk yöntem ve davranış biçimlerinden oluşuyor. Beyin bir işlemi rutinleştirmek için gerekli bilgileri topladığı andan itibaren otomasyon için gerekli kodu oluşturup, gerekli gördüğü anlarda bu kodu koşturmaya başlıyor.

Buna örnek olarak switch komutunu verebiliriz. Switch komutunu ilk defa uygulayan bir programcı, daha sonra hiç düşünmeden bu yapıyı kullanmaya meğil gösterecektir, çünkü beyni onu alışık olduğu bir rutin kullanmaya teşvik edecektir. Oysaki if/else ya da switch yapılarını nesneye yönelik programlama teknikleri ile ortadan kaldırılabilir. Ama önce switch kullanmak için beyinde yer alan rutinin silinmesi ve yeni bir rutin oluşturulması gerekmektedir. Bunu mümkün mü? Mümkün ise, nasıl yapılabilir?

Benim musluk problemini çözmem mümkün değil. Büyük bir ihtimalle beynim mevcut musluk kullanma rutinlerini bulunduğu ortama göre zaman içinde adapte edecektir. Aktif olarak benim bir şey yapmam gerekmiyor. Ama beynim bir şeyler yapması gerektigini kısa zamanda anladı ya da anlayacak, çünkü musluğun başında durup, açılmasını beklemek sadece beni değil, onu da zaman içinde rahatsız edecektir. Bu gerekliliği gördüğü için mevcut musluk rutinlerini adapte edeceğini düşünüyorum.

Lakin bu durumu switch kullanan bir programcı ile kıyasladığımızda, durumun çok farklı olduğunu görmekteyiz. Programcı switch komutunu kullandığında, benim gibi anormal bir

durumla karşılaşmıyor. Programcı için bir iritasyon söz konusu değil. Bu yüzden switch komutunu kullanmayı öğrenmiş bir programcı ömrü boyunca switch komutuna sadık kalacaktır. Ama programcının beynide switch komutunu kullanma rutininin oluşması, bu komutun kullanımının doğru olduğu anlamına gelmez. Burada, ne yazık ki programcı gözünü ilk açtığında switch komutunu gördüğünden, switch komutunu ilk uygularken beyni gizlice olup, bitenlerin fotoğrafını çekmiş ve hemen gerekli rutinleri oluşturmuştur.

İşte daha iyi bir programcı **olamamanın** başlıca sebebi budur. Ne yazık ki birçok programcı ilk öğrendiklerinin esiri olarak kalırlar. Bu durumu sorgulamadıkları sürece, mevcut kod yazma rutinlerinin kullanımı devam eder. Onların da ne kadar iyi oldukları ortadadır.

Bu durumu sorgulamaya başlamış ve daha iyi kod yazmak isteyen bir programcı ne yapmalıdır? Bu sorunun cevabı çok kolay olmakla birlikte, uygulama zorluk derecesi kişiye göre değişecektir. Kısaca cevap vermek gerekirse, eski rutinleri silip, beynimizi yeni rutinler oluşturmaya tesvik etmemiz gerekmektedir. Bunu sağlayan tek bir yöntem tanıyorum: **pratik yapmak!** Bu konuda daha önce yazmış olduğum yazılara aşağıdaki linkler üzerinden ulaşabilirsiniz.

- Kod Kata ve Pratik Yapmanın Önemi
- Alışkanlıkların Gücü
- KodKata.com
- Kataların Eşli Programlanması
- Deneme Yanılmanın Bedeli

Mevcut rutinleri silip, yenilerini oluşturmak mümkün. Ama bunun bir bedeli var.

Mevcut Bir Uygulama Koduna Nasıl Adapte Olunur?

http://www.pratikprogramci.com/2015/02/11/mevcut-bir-uygulama-koduna-nasil-adapter-olunur/

Mevcut bir yazılım projesine dahil oldunuz ve sizden kısa zamanda koda adapte olmanız ve yeni gereksinimleri implemente etmeniz bekleniyor. Nasıl en kısa zamanda, kendinizi evinizde hissedecek şekilde sisteme adapte olabilirsiniz? Bu ve buna benzer sorular bana ulaşıyor. Bu yazımda bu tür soruların cevabını vermeye çalışacağım.

İki tür programcı vardır:

- Değişikliği arayan
- Değişikliği istemeyen

Değişikliği kabul etmeyenlerin bünyeleri yeni projeleri kaldırmakta zorlanabilir, lakin değişikliği arayanlar için yeni projelerin ganimet olduğunu biliyorum. Korkunun ecele faydası yoktur misali yeni projeyi cesur bir şekilde göğüslemeye hazır olmak, değişikliklere adapte olmak için en önemli koşulların başında gelmektedir.

Cesaret programcının ismi ise, resmin genelini görmek soyismidir. Programcı yeni bir projede gidişatı anlayabilmek için genel resmi görmeye çaba sarfetmelidir. Genel resmi görmenin faydaları şu şekilde sıralanabilir:

- Genel resmi görmek insanı rahatlatır, çünkü resmin neresinde olduğunu bilmek insana güven verir.
- Genel resmi görmek çözüm üretme sürecini hızlandırır, çünkü insan nasıl ilerlemesi gerektiğini kestirebilir.
- Genel resmi görmek detayları anlamayı kolaylaştırır. Bir uygulamada genel resim ile tabir ettiğim şey nedir?

Genel resim:

- Müsterinin ne istediğidir.
- Mimaridir.
- Testlerdir.
- Kod değildir.

En son nokta ile başlamak istiyorum. Genel resim kod değildir dedim. Buradaki yazımda aktarmaya çalıştım. Başkalarının yazdığı kodu okuyarak, olup, bitenleri kavramak çok güçtür. Bu yüzden salt kod analizi ile genel resmi görmeye çalışmak, açma ipi olmayan paraşüt ile uçaktan atlamak gibidir. Bakarsın açılır ihtimalinin olmaması gibi, bakarsın olup, bitenleri kavrarım ihtimalinin olması da çok düşüktür.

Genel resmi görme kabiliyetine erişebilmek için müşterinin ne istediğini bilmek gerekir. Müşterinin vizyonunu tanımadan ve hangi gereksinimlere sahip olduğunu bilmeden genel resme bakılsa bile, anlaşılması çok zordur.

Yazılım projelerinde genel resim iki bölümden oluşur:

- Müşterinin vizyonu
- Projenin teknik yapısı

Teknik yapı öncelikle uygulamann sahip olduğu mimaridir. Her uygulamanın bir mimariye sahip olduğu söylenemez. Çoğu zaman ilk müşteri gereksinimlerine göre oluşturulan mimari, zaman içinde yeni gereksinimler ile adapte edilmediği için, uygulamanın altında ezilir ve varlığını gösteremez. Bu genel resmi görmek için mimariyle ilgilenmenin faydalı olmayacağı anlamına gelmektedir.

Testler teknik yapının bir parçasıdır. Bircok projede testler varlıklarıyla değil, yokluklarıyla göze batalar. Testlerin olmaması, herhangi bir teknik yapının varlığına engel olmamakla birlikte, teknik yapının kendi ifade gücünü azaltır. Genel resmi görememekteki en büyük zorluğun eksik olan testlerden kaynaklandığını düşünüyorum. Yazımın ilerleyen bölümlerinde bu konuya açıklık getirmeye çalışacağım.

Bir uygulamanın nasıl yapılandırıldığı (build), sürümlendiği (release) ve yüklendiği de (deployment) teknik yapının bir parçasıdır ve genel resmi görmek için anlaşılması gereken unsurlardandır.

Projenin teknik yapısına ekibi, projenin yönetiliş tarzını ve yazılım geliştirme sürecini de dahil edebiliriz.

Yazılımcı olarak yeni bir uygulamaya adaptasyon için genel resmi görmenin önemini vurgulamaya çalıştım. Genel resmi görmeye çalışmak yine de soyut bir kavram. Bu soyutluğu teknik yapıdan bahsederek, somutlaştırmaya çalıştım. Konuyu daha da somutlaştırmak adına projeye adapte olmak için atılması gereken adımlardan bahsetmek istiyorum.

Bir uygulamanın nasıl çalıştığını anlamanın en kolay yolu, uygulamanın kullanıcılar tarafından nasıl kullanıldığını görmekten geçmektedir. Kullanıcı uygulamayı bir kara kutu olarak görür. Kullanıcı uygulama ile interaksiyona girer ve girdiği bilgiler (input) doğrultusunda, uygulamadan belli davranış biçimleri (output) bekler. Ama ne yazık ki bir kullanıcının yanına oturarak, uygulamanın nasıl kullanıldığını görmek, yeni yazılımcıyı projeye hazırlamak için yeterli değildir. Bunun başlıca sebebi uygulamanın kullanıcı için bir kara kutu gibi çalışması ve yazılımcının kullanıcı interaksiyonlarının kodsal karşılığını görememesidir. Buradan uygulamanın nasıl kullanıldığını gösteren kod birimlerine ihtiyacımız olduğu sonucunu çıkarabiliriz. Aşağıda böyle bir kod birimini görmektesiniz:

```
@Test
public void verifyThatCustomerCanBeFound() {
    HomePage home = new HomePage(driver);
    SearchResultPage searchResult = home.searchFor("özcan acar");

    CustomerPage page = searchResult.clickOnTopSearchResultLink();

    String expected = "Özcan Acar HomePage";
    String actual = page.getHeadLine();
    assertTrue(actual.contains(expected));
}
```

Bu web bazlı bir uygulamayı test etmek için Selenium ile yazılmış bir onay/kabul (acceptance) testi. Onay/kabul testleri uygulamayı en üst seviyede ve çalışır haldeyken test eden entegrasyon testleridir. Testde görüldüğü gibi müşteri arama sayfasına gidilerek, aranan müşteri ismi giriliyor ve gelen bilgiler kontrol ediliyor. Aynı işlemi kullanıcı da muhtemelen yapıyor olacaktır, lakin kullanıcı işlem esnasında HomePage, SearchResultPage ve CustomerPage gibi sınıfları görmez. Yazılımcı için testlerin avantajı burada başlamaktadır.

Testler genel olarak bir uygulamanın nasıl kullanıldığını gösterir niteliktedir. Testler uygulama için kullanıcı (client) konumundadır. Testlere bakarak, uygulamanın nasıl çalıştığını, uygulamanın giriş ve çıkış noktalarını keşfedebiliriz. Bu sebepten dolayı yeni bir projeye dahil olan yazılımcılar için testler uygulamanın **kullanma rehberi** olma özelliğini taşırlar. Testler incelendiği ve çalıştırıldığı zaman, uygulamanın nasıl çalıştığı hakkında çok kısa bir zamanda fikir sahibi olmak mümkündür.

Testler aracılığı ile uygulamayı çalıştırdıktan sonra durdurmak (debugging) ve adım, adım takip de etmek mümkündür. Bunu yapabilmek de uygulamayı anlama adına atılabilecek en önemli adımlardandır. Bunu nasıl çalıştığını bilmediğimiz ya da çalıştıramadığımız kod birimlerini sadece okuyarak anlama yöntemiyle kıyasladığımızda, testlerin uygulama bünyesindeki mevcut davranış biçimlerini dokümente etmeleri yanı sıra, uygulamayı çalışır hale getirmek için de kullanılabilecek araçların başında geldiğini söyleyebiliriz.

Değişik türde testler hazırlamak mümkündür. Kullanıcı interaksiyonlarını simüle etmek için onay/kabul, uygulamayı entegre ederek, çalıştıran entegrasyon ve işletme mantığını test eden birim (unit) testleri oluşturulabilir.

En iyi testler uygulamanın dış dünyaya sunduğu arayüzlere karşı yazılmış testlerdir. Bu arayüzlere API (Application Programming Interface) ismi verilir. APIyi uygulamanın giriş kapısı olarak düşünebiliriz. APIyi hedef alan testler, uygulamanın nasıl çalıştığına dair ipuçları verirler. Aşağıdaki örnekte uygulamanın işletme katmanının APIsine ait IJobConfigurationService sınıfı için bir entegrasyon testi yer almaktadır.

```
@Test
public void when_property_file_is_found_then_a_valid_jobconfiguration_instance_is_returned
            throws Exception {

    // Given
    IJobConfigurationService service =
                new FileBasedJobConfigurationServiceImpl();

    // When
    final JobConfiguration configuration =
                this.service.getConfiguration("dummy");

    // Then
    assertThat(configuration.getConfig()
            .getString("ARTIFACT_ID"), is(equalTo("test")));

}
```

Bu örnekte işletme katmanı APIsinin bir parçası olan IJobConfigurationService sınıfının getConfiguration() isimli metodunun nasıl kullanıldığını görmekteyiz. Bu test bize IJobConfigurationService sınıfının nasıl kullanılması gerektiğini apaçık göstermektedir. Bu testi çalıştırarak, neler olup, bittiğini gözlemleyebiliriz. Yaptığımız gözlemler uygulamanın hangi boyutta işlem yaptığını ve hangi parçalardan oluştuğunu anlamamızı yani genel resmi görmemizi kolaylaştıracaktır.

Haklı olarak her projede test olmayabiliyor diyebilirsiniz. Bu konuda size hak veriyorum. Ne yazık ki teknolojik gelişmelerin en hızlı yaşandığı çagda olmamıza rağmen, yazılım projeleri halihazırda iptidai yöntemlerle götürülmeye çalışılmaktadır. Bunların başında da testlerin eksikliği gelmektedir.

Testler yoksa, uygulamayı anlamak için yeni testlerin yazılması geçerli bir seçenek olabilir. Sadece bu şekilde uygulamanın nasıl çalıştığını deneme, yanılma yöntemiyle anlayabiliriz. Testler doğaları itibari ile belli arayüzleri kullanma eğilimi gösterirler. Yeni testler yazarak, bu arayüzleri keşfedebilir ve uygulamanın APIsi hakkında fikir sahibi olabiliriz. Mevcut kod için sonradan test yazmak zahmetli bir iş olabilir, ama oluşan yeni testlerin uygulamanın nasıl çalışdığına dair ipuçları vereceği gerçeğini unutmamak lazım.

Uygulamayı anlamanın diğer bir yöntemi, uygulamayı çalışır hale getirerek, oluşan log kayıtlarını incelemek ve yeni log kayıtları oluşturmaktır. Bu debugging yöntemiyle kombine edildiğinde, uygulamanın nasıl işlediği ve hangi parçalardan oluştuğu konusunda fikir sahibi olmayı kolaylaştıracaktır.

Yeni bir projeye adapte olmak için teknik beceriler yanı sıra sosyal becerilerin de rol oynadığı bir gerçek. Buraya kadar bahsetmiş olduğum yöntemler teknik anlamda bireyin kendi başına uygulayabileceği yöntemlerdir. Bunun yanı sıra proje hakkında bilginin ekip içinde çalışan bireylerde de olduğunu unutmamak gerekir. Ekip bireyleri ile sıkı bir iletişim, uygulamanın nasıl çalıştığını anlamayı kolaylaştıracaktır. Ekip bünyesindeki bilgiyi bir sünger gibi çekebilmek için, ekip çalışanları ile diyalog içinde olmak ve ekibin bir parçası haline gelmek önemli bir adımdır.

Günümüzde birçok projede Scrum, XP (Extreme Programming) ve FDD (Feature Driven Development) gibi çevik süreçler kullanılmaktadır. Bu tarz yönetilen bir projeye dahil olmadan önce, çevik süreçlerin işleyisi hakkında ön bilgiye sahip olmak gereklidir. Bu yüzden projeye hazırlıklı olarak dahil olmak, projeye teknik hakimiyetin kolaylaşmasını sağlayacaktır.

Neden Spring, JPA ve Diğer Çatılar ÖğrenilmeMEli

http://www.pratikprogramci.com/2014/12/12/neden-spring-jpa-ve-diger-catilar-ogrenilmemeli/

Bu başlığın çok provokasyon yüklü olduğunu biliyorum. Ama zaman ayırıp, yazımın geri kalanını okuyabilirseniz, ne demek istediğimi açıklamaya çalışacağım.

Yazılımda bağımlılıkların tersine çevrilmesi (DIP – Dependency Inversion Principle) isminde bir tasarım prensibi var. Burada yer alan yazımda bu tasarım prensibinin ne olduğunu ve nasıl uygulandığını göstermeye çalıştım. DIP kullanılmadığı taktirde kırılgan kod birimleri oluşur. Bunun başlıca sebebi, bağımlılıkların sıkça değişikliğe uğrama potansiyeline sahip olan somut sınıflar yönünde olmasıdır.

DIP yazılımda iki kavramın varlığına işaret ediyor. Bunlar:

- Soyutluk
- Somutluk

Yazılımda soyut yapılar hangi işlemin yapılması gerektiğini tanımlarlar, ama nasıl yapıldıklarını göstermezler. Örneğin Java dilinde interface sınıfları yardımı ile soyut işlemler tanımlanır. Aşağıda bunun bir örneğini görmekteyiz:

```
public interface Logger{

  void log(String msg);
}
```

Bu örnekte Logger isminde soyut bir sınıf tanımladık. Bu sınıf bünyesinde yine soyut olan ve log ismini taşıyan bir metot yer alıyor. Logger sınıfı loglama işlemini soyut bir şekilde tanımlamakta. Sınıf bünyesinde loglama işleminin somut olarak nasıl yapıldığını göremiyoruz. Şimdi Logger sınıfının somut bir halini görelim:

```
public class Logger{

  void log(String msg){
       System.out.println(msg);
  }
}
```

Yukarıda yer alan Logger sınıfı somut bir sınıftır, çünkü loglama işlemi için somut bir implementasyona sahiptir. Bu implementasyon değişikliğe uğrayabilir. Örneğin log4j bazlı bir loglama yapılmak istendiğinde, somut olan Logger sınıfının adapte edilmesi gerekmektedir. Bu adaptasyon Logger sınıfını kullanan diğer sınıflar üzerinde etki yaratır ve onların da değiştirilmesine sebep olabilir. Bu yüzden somut sınıflara olan bağımlılıklar uygulamanın kırılganlığını artırır. Bunun önüne geçmenin yolu DIP tasarım prensibini uygulamaktan geçer. Ama bu yazımda maksadım DIP tasarım prensibini tanıtmak değildi. DIP'i örnek vererek, çok daha değişik bir konuya değinmek istiyorum.

DIP ve diğer tasarım prensipleri uygulamaların nasıl yapılandırılmaları gerektiği konusunda

fikir beyan eden soyut yapılardır. Kendilerinin nasıl implemente edilmeleri gerektiği konusuna değinmezler. Kullanılan programlama dilini göre somut implementasyonları değişmektedir. Bu yüzden tasarım prensiplerini bir yazılım felsefesi olarak düşünebiliriz. Bu felsefeyi tanıdığımız sürece onları istediğimiz bir dilde uygulayabiliriz. Onlardan faydalanabilmek için nasıl uygulandıklarını değil, ne için var olduklarını bilmemiz, yani bu felsefeye hakim olmamız gerekmektedir.

Dün genç bir yazılımcı arkadaşımızdan bir e-posta iletisi aldım. "Spring çatısını öğrenmeye çalışıyorum, ama ne amaçla kullanıldığını bir türlü anlayamıyorum" demiş. Spring'i yeni öğrenen herkesin bu sıkıntıyı çektiğinden eminim. Bu gerçek problemin ne olduğunu örten bir semptom. Spring ve diğer çatıları öğrenmek isteyenler bu acıları çekiyorlar, lakin asıl sorunun ne olduğunu bilmiyorlar.

Yazımın başında yazılımda var olan iki kavramdan bahsetmiştim: soyutluk ve somutluk. Eğer DIP soyutluk ise, Spring ve var olan diğer tüm yazılım çatıları somutluktur. DIP soyut bir felsefe ise, Spring somut bir teknolojidir.

Spring'in temelinde şu felsefeler yatıyor:

- **interface bazlı programlama**: Spring bağımlılıkların soyut sınıflara doğru olmasını teşvik ediyor. Spring somut bir DIP implementasyonudur.
- **Bağımlılıkların enjekte edilmesi**: Spring ile bir sınıfın ihtiyaç duyduğu tüm bağımlıklar o sınıfa dışarıdan enjekte edilebilir. Bu somut implementasyonun altında Hollywood prensibi (dont call us, we call you) ya da felsefesi yatıyor. Bu felsefeye göre bir sınıf ihtiyaç duyduğu şeyleri toplamak yerine, bu şeyler sınıfa dışarıdan enjekte ediliyor.
- **Nesnelerin olusturulması**: Spring singleton ve prototype tasarım şablonlarını kullanarak, nesneleri oluşturur ve ihtiyaç duyan nesnelere enjekte eder.
- **Transaksiyon yönetimi**: Spring bir veri tabanı transaksiyonunu yönetmek için vekil tasarım şablonunu implemente eder.

Tasarım prensipleri ve şablonları konularında hiç ihtisas yapmamış birisinin Spring çatısının ne olduğunu ve neden kullanılması gerektiğini anlaması imkansızdır. Bir teknolojinin nasıl kullanıldığını bilmek ile neden kullanılması gerektiğini bilmek arasında büyük farklılıklar vardır. Nedenin cevabını verebilmek için altında yatan felsefeyi kavramış olmak gerekmektedir. Bir şeyin nasıl kullanıldığını öğrenmek için neden kullanılması gerektiğini bilmek zorunluluğu yoktur. Lakin neden kullanılması gerektiği bilinmeyen bir şeyin nasıl kullanıldığını bilmekte çok faydasızdır. Yazılım piyasasında ne yazık ki bir takım çatıların nasıl kullanıldığını bilen ama neden kullanılmaları gerektiği konusunda fikir sahibi olmayan yazılımcılar mevcuttur, çünkü temelde yatan felsefe ve soyutluktan bihaberdirler.

Günümüzde, özellikle yazılım sektöründe somutluk soyutluğun yani teknoloji felsefenin önüne geçmiş durumda. Yazılımcılar işin temelinde yatan felsefeyi anlamadan somut teknolojilere yönelmektedirler. Birçok yazılımcının Spring ve Hibernate gibi bir takım çatıları öğrenme çabasında olduklarını gözlemliyorum. Bana gelen sorular da genelde "hangi çatıyı öğrenmeliyim" şeklinde oluyor. Bu konuda benim cevabım aynı oluyor: "teknolojiyi değil, bu işin felsefesini öğrenmeye gayret edin". Ne olduğu kavranamayan bir çatıyı kullanmaya çalışmak beyhude bir iştir. İşin felsefesine değil de teknolojisine yönelen yazılımcılar daha iyi programcı

olamama yolunda istikrarlı bir şekilde yürümektedirler.

Şimdi tekrar yazımın başlığına geri dönmek istiyorum. "Neden Spring, JPA ve diğer çatılar öğrenilmemeli" demiştim. Bu başlığı "Neden teknoloji yerine önce felsefesi öğrenilmeli" şeklinde değiştirmek istiyorum.

Teknolojiler gelip, geçicidir, felsefeler ise her zaman daim...

Versiyon ve Sürüm Numaraları Nasıl Oluşturulur?

http://www.pratikprogramci.com/2014/11/22/versiyon-ve-surum-numaralari-nasil-olusturulur/

Bir yapılandırma (build) işleminin ardından derlenen kodu ve kodla ilgili diğer dosyaları ihtiva eden bir dosya oluşur. Bu dosyaya yapı (artifakt) ismi verilir. Yapı bünyesinde uygulamanın çalışır şekli yer alır. Java dünyasında yapılar jar, war, ear gibi ZIP kökenli dosyalar içinde yer alır. Microsoft dünyasında yapılar dll, Linux dünyasında rpm, tar ya da gz formatındadır.

Bu yazımda oluşturulan yapıların nasıl versiyonlandırıldığını ve bunun neden gerekli olduğunu örnekler üzerinden göstermek istiyorum. Öncelikle yapının ne olduğunu ve ne ihtiva ettiğini inceleyelim.

Java dilinde aşağıdaki sınıfı oluşturduğumuzu düşünelim:

```
import java.io.File;
public class FileUtils {

	public static void copy(File file, File dir){
		// ...
	}
}
```

FileUtils sınıfında yer alan copy metodunu bir dosyayı bir dizine kopyalamak için kullanabiliriz. Bu çok temel bir fonksiyon gibi görünüyor, yani bu fonksiyonu projemizde ya da başka bir proje bünyesinde dosya kopyalama işlemleri yapmak için kullanabiliriz. Başka bir projede bu sınıfı kullanmak istediğimizde, bu sınıfı ihtiva eden bir jar dosyası oluşturmamız gerekiyor. Jar dosyası aşağıdaki formatta olacaktır:

```
fileutils.jar
```

Eger .Net dünyasında olsaydık, oluşturduğumuz kütüphaneyi şu şekilde başkalarına verirdik:

```
fileutils.dll
```

Linux dünyasında bu kütüphaneyi şu şekilde başkalarıyla paylaşırdık:

```
fileutils.so
```

fileutils.jar/dll/so dosyasını tekrar kullanılabilir yapıda bir kod birimi olarak düşünebiliriz. Bu kütüphaneyi kendi projemize ekledikten sonra, FileUtils sınıfında yer alan copy fonksiyonunu dosya kopyalama işlemleri için kullanabiliriz. Başka birisi tarafından oluşturulan fonksiyon kütüphanelerini bu şekilde tekerleği tekrar icat etmek zorunda kalmadan kullanabiliriz. Nitekim bir Java uygulamasının çok büyük bir bölümü (Hibarnete, Spring, Jdbc...) tekrar kullanılabilir fonksiyon kütüphanelerinden oluşmaktadır.

Şimdi şu senaryoyu hayal edelim. fileutils.jar fonksiyon kütüphanesini kullanılmak üzere başka projelerle paylaştık. Bize copy ya da sunduğumuz başka bir fonksiyon bünyesinde bir hata

olduğu bilgisi ulaştı. Bu durumda hatayı giderdikten sonra, fileutils.jar dosyasını tekrar oluşturup, kullanıcı projelerle tekrar paylaşmamız gerekiyor. Bu çok zahmetli bir iş gibi görünse de, Maven ya da Ivy yapı araçlarıyla kütüphane dağıtımını otomatize edebiliriz.

FileUtils sınıfını oluşturup, fileutils.jar dosyasında dağıtmaya başladığımız andan itibaren bir uygulama arayüzü (API – Application Programming Interface) oluşturmuş olduk. FileUtils sınıfında yer alan her public fonksiyon bu arayüzün bir parçasıdır. Herkes bu arayüzü kullanarak, mevcut kodun ve kütüphanenin tekrar kullanılmasını sağlayabilir. Bunun yanı sıra fileutils.jar bünyesinde yer alan her public fonksiyon ya da metot oluşturduğumuz bu uygulama arayüzüne dahildir.

Başka programcılar kendi projelerinde fileutils.jar kütüphanesinde yer alan herhangi bir fonksiyon ya da metodu kullanmaya başladıklarında, oluşturduğumuz uygulama arayüzüne bağımlı hale gelirler. Bu noktadan itibaren fileutils.jar kütüphanesinden sorumlu programcı olarak, bu kütüphane bünyesinde yer alan public fonksiyon ya da metotlar üzerinde istediğimiz şekilde değişiklik yapamayız. Bunu yaparsak, çalışmaz hale gelen diger projelerdeki yazılımcıların ne kadar sinirleneceği sanırım tahmin edilebilir. Örneğin copy fonksiyonunu aşağıdaki şekilde değiştirmemiz mümkün değildir. Eğer bunu yaparsak, yeni fileutils.jar sürümünü alan ve copy fonksiyonunu kullanan her uygulama çalışmaz hale gelecektir, çünkü copy fonksiyonunun eski haline göre derlenmişlerdir.

```
import java.io.File;
public class FileUtils {

	public static void copy(File[] files, File dir){
		// ...
	}
}
```

Bu noktada şu sonuçları çıkarmak mümkün:

- Programcı public fonksiyon ya da metot oluşturmamaya dikkat etmelidir. Public olan bir fonksiyon ya da metodu herkes kullanabilir. Bu fonksiyon ya da metot sahibinin oluşturduğu fonksiyon ya da metodu yaşam süresi boyunca desteklemesi gerektiği anlamına gelmektedir. Java metotlara erişimi public ve private gibi direktiflerle desteklemektedir. Bu desteği sağlamayan dillerde tanımlanan tüm fonksiyon ve metotlar public konumundadır. Bu durumda kütüphane oluşturularak, bu kod birimlerinin dağıtılması, programcısının destek sağlaması anlamına gelmektedir.
- Kod birimlerinin tekrar kullanılabilir yapıda olmalarınını ya da kullanıcı ile kullanılan arasındaki ilişkiyi tanımlamak için bir uygulama arayüzü (API) oluşturulmalıdır. Bu çoğu zaman fonksiyon ve metotların public olmaları gerektiği anlamına gelmektedir. Bir uygulama arayüzü oluşturma niyeti, bu arayüzün bakımı ve geliştirilmesi için gerekli desteğin sağlanmasını da kapsamalıdır.
- Oluşturulan API'nin başka uygulamalar bünyesinde kullanılmasını sağlamak amacıyla jar ya da herhangi başka bir formatta bir kütüphane dosyasına yerleştirilmesi gerekmektedir. Sadece bu şekilde derlenmiş kod başka projelerde doğrudan kullanılabilir.

Uygulama arayüzleri oluşturulması görüldüğü gibi beraberinde bazı sorumluluklar

getirmektedir. Bu arayüzleri kullanıcı ile kullanılan arasında bir nevi anlaşma metni olarak düşünebiliriz. Kullanılan taraf neyin, nasıl kullanılacağını oluşturduğu arayüzler aracılığı ile ifade etmektedir. Kullanıcı taraf bu arayüzleri kullanarak, bağımlı hale gelmekte birlikte, anlaşma metnine dayanarak anlaşmanın bozulmamasını arzu etmektedir. Bu anlaşma metnine göre kullanılan taraf kullanım şeklinde arayüzde tanımlananın haricinde değişiklik yapamaz. Şimdi hangi durumlarda böyle değişikliklerin yapılabileceğini inceleyelim.

Bir fonksiyon kütüphanesin versiyon kullanılmadan dağıtılması durumunda, API'lerde meydana gelen değişikliklerden dolayı kullanıcılarını olumsuz etkileyebileceğini gördük. Bunun önüne geçmek için versiyon numaraları kullanılabilir. Örneğin FileUtils sınıfını ihtiva eden bir fonksiyon kütüphanesinin şu şekilde bir sürümünü oluşturabiliriz:

```
fileutils-0.1.0.jar
```

fileutils-0.1.0.jar FileUtils sınıfının ilk sürümünü ihtiva etmektedir. Herhangi bir kullanıcı fileutils-0.1.0.jar kütüphanesini alarak, kendi projesinde kullanabilir. Şimdi FileUtils sınıfı üzerinde bir değişiklik yaptığımızı düşünelim. Örneğin copy fonksiyonunu değiştirmiş ya da başka bir fonksiyon eklemiş olalım. Bu durumda aşağıdaki sürümü oluşturmamız gerekmektedir:

```
fileutils-0.2.0.jar
```

Bu durumda iki değişik fonksiyon kütüphanesi sürümüne sahip olduk:

```
fileutils-0.1.0.jar
fileutils-0.2.0.jar
```

İsteyen 0.1.0 ya da 0.2.0 sürümünü kullanabilir. Bu yöntemin sağladığı avantajlar şunlardır:

- fileutils-0.1.0.jar kullanıcısı fileutils-0.2.0.jar bünyesinde meydana gelen değişikliklerden etkilenmez, çünkü fileutils-0.1.0.jar sürümüne sahiptir. Eğer duyduğu bağımlılık fileutils.jar formatında olsaydı, yeni bir fileutils.jar sürümü ile değişikliklerden etkilenebilirdi. Bu şekilde fileutils-0.1.0.jar isimli sürümü kullanarak, hayatını sürdürebilir ve diğer değişikliklerden etkilenmez.
- Kullanıcı istediği zaman başka bir sürüme geçme avantajına sahiptir. Örneğin fileutils-0.1.0.jar kullanıcısı fileutils-0.2.0.jar ile gelen yeni copy metodunu kullanmak isterse, bağımlılığını fileutils-0.2.0.jar şeklinde değiştirerek, yeni copy fonksiyonunu kullanmaya başlayabilir.
- fileutils-0.1.0.jar kütüphanesini oluşturan programcı, fileutils.jar kullanıcılarını olumsuz bir şekilde etkilemek zorunda kalmadan oluşturduğu API üzerinde değişiklik yapabilir.
- Versiyon numaraları sayesinde aynı API'nin değişik sürümlerini kullanmak mümkün hale gelmektedir.

Versiyonlamış sürümlerin sağladığı avantajları gördük. Sürümlerin neden versiyonlanmaları gerektiğine Tekrar Kullanım ve Sürüm Eşitliği (REP – Reuse-Release Equivalence Principle) tasarım prensibi işaret etmektedir. Buna göre tekrar kullanımı kolaylaştırmak için paket sürümlerinin oluşturulması şarttır. REP'e göre tekrar kullanılabilirlik (reuse) sürüm (release) ile direk orantılıdır. Sürüm ne ihtiva ediyorsa, o tekrar kullanılabilir.

Şimdi sürüm numaralarının nasıl yönetilmesi gerektiği konusuna değinmek istiyorum. Şimdiye kadar verdiğim örneklerde üç rakamdan oluşan bir versiyon numarası kullandım. Versiyon numarası şu bölümlerden oluşmaktadır:

```
MAJOR.MINOR.BUGFIX
```

- **MAJOR**: Büyük versiyon numarasıdır. Sadece API üzerinde değişiklikler yapıldığında artırılır.
- **MINOR**: Küçük versiyon numarasıdır. Uygulamaya yeni özellikler (feature) eklendiğini artırılır
- **BUGFIX**: Giderilen hataları gösteren versiyon numarasıdır. Her bugfix işlemi ardından artırılır.

Versiyon numaralarının nasıl artırıldığını bir örnek üzerinde birlikte inceleyelim. Kısa bir zaman önce fileutils-0.1.0.jar sürümünü oluşturduğumuzu düşünelim. İsteyen her proje fileutils-0.1.0.jar dosyasını bağımlılık olarak kullanmaya başlamış olsun. Kısa bir zaman sona kullanıcılarımızdan birisi bir hata keşfeder ve bize durumu bildirir. Bu durumda hatayı giderdikten sonra, yeni bir sürüm oluşturmamız gerekmektedir. Sadece yeni sürüm ile kullanıcılarımız hatanın giderilmiş haliyle çalışabilirler. Hatayı giderdikten sonra oluşturmamız gereken sürüm şu şekilde olmalıdır:

```
fileutils-0.1.1.jar
```

Tekrar bir hatanın keşfedildiği haberini alıyoruz. Bu durumda yeni sürüm şöyle olur:

```
fileutils-0.1.2.jar
```

Görüldüğü gibi hatalar giderildiğinde sadece BUGFIX numarası artırılmaktadır. Kütüphanemize yeni bir fonksiyon eklediğimizde, sürüm numarası şu şekilde olmalıdır:

```
fileutils-0.2.0.jar
```

Kütüphanemize yeni bir özellik (örneğin yeni bir copy fonksiyonu) ekledikten sonra, MINOR numarasını artırdık ve BUGFIX numarasını sıfırladık. Oluşturduğumuz bu yeni sürümde bir hata olursa, bu hatayı düzelttikten sonra aşağıdaki sürümü oluşturmamız gerekir:

```
fileutils-0.2.1.jar
```

Yeni bir fonksiyon ile sürüm aşağıdaki versiyon numarasını taşımaya başlar:

```
fileutils-0.3.0.jar
```

Şimdi MAJOR versiyon numarasının ne zaman değiştirilmesi gerektiği konusunu inceleyelim. Eğer mevcut bir API'yi bir önceki sürümünde yer aldığı hali ile uyuşmaz hale getirirsek, yani API'yi değiştirirsek, bu durumdan kullanıcılarımızı haberdar etmemiz gerekir. Bunu MAJOR versiyon numarasını yükselterek yapabiliriz. fileutils-0.1.0.jar sürümünde copy fonksiyonu şu şekilde yer almaktadır:

```
import java.io.File;
public class FileUtils {

	public static void copy(File file, File dir){
		// ...
	}
}
```

Ne yazık ki başka alternatifimiz olmadığı için copy fonksiyonunu şu şekilde değiştirmemiz gerekiyor:

```
import java.io.File;
public class FileUtils {

	public static void copy(File[] file, File dir){
		// ...
	}
}
```

Görüldüğü gibi API'mizi oluşturan FileUtils.copy() metodunun parametre listesi değişti. Bu durumda sürüm numarası şu şekilde olmamalıdır:

```
fileutils-0.2.0.jar
```

fileutils-0.1.0.jar sürümünü kullanan bir proje fileutils-0.2.0.jar sürümüne baktığında, kütüphanenin sahip oldugu API'de bir değişiklik olmadığını, kütüphaneye sadece yeni bir özelliğin eklendiğini düşünecektir. Proje fileutils-0.2.0.jar sürümüne geçtiği zaman yazılımcılar copy fonksiyonunun düşündükleri gibi çalışmadığını görürler, çünkü bu fonksiyon değişikliğe uğramıştır. API değişmiştir. kullanıcılarımıza API değişikliklerini göstermek için MAJOR numarasını artırmamız gerekmektedir. API değişikliğinden sonra sürüm numarası ku şekilde olmalıdır:

```
fileutils-1.0.0.jar
```

Bu durumda fileutils-0.1.0.jar kullanıcısı fileutils-1.0.0.jar sürümüne baktığında, kullandığı API'nin değişmiş olabileceğini görebilir. Eğer isterse fileutils-1.0.0.jar sürümüne geçebilir. Ama bu durumda gerekli değişiklikleri yaparak, uygulamasını yeniden derlemesi gerekmektedir. Eğer bu değişiklikleri yapmak istemezse ya fileutils-0.1.0.jar sürümünde kalabilir ya da bu sürüm ile uyumlu olan başka bir sürüm bakar. fileutils-0.1.0.jar ile aşağıdaki sürümler uyumludur:

```
fileutils-0.1.1.jar
fileutils-0.1.2.jar
fileutils-0.1.3.jar
fileutils-0.2.1.jar
fileutils-0.2.2.jar
fileutils-0.3.0.jar
fileutils-0.4.0.jar
fileutils-0.5.0.jar
fileutils-0.6.0.jar
```

Bu sürümlerin hepsinde fileutils-0.1.0.jar bünyesinde yer alan copy fonksiyonu değişmemiştir. Kütüphaneye sadece yeni özellikler eklenmiştir. Sürüm numarasına bakarak, yapılan değişiklikleri kabaca görmek mümkündür. fileutils-1.0.0.jar dosyasına baktığımızda, kullandığımız API'nin değişmiş olabileceğini görüyoruz. Bu değişiklik doğrudan kullandığımız copy fonksiyonu ile ilişkili olmayabilir. Örneğin Java dilinde interface sınıfları üzerinde yapılan değişiklikler API değişikliği olarak sınıflandırılır ve yeni sürümlerde mutlaka MAJOR sürüm numarası artırılır.

Şimdi kısaca sürüm oluşturulurken dikkat edilmesi gereken konulara değinmek istiyorum. Bunlar:

- fileutils-1.0.0.jar şeklinde yeni bir sürüm oluşturulduktan sonra, bu sürüm dosyasının değişikliğe uğramaması için gerekli tüm tedbirler alınmalıdır. Örneğin birisi fileutils-2.1.0.jar sürümünü alıp, fileutils-1.0.0.jar şeklinde degistirdikten sonra, bu sürümü fileutils-1.0.0.jar olarak deklare edememelidir. Kullanıcılar ilk etapta versiyon numarasına bakarak, hangi sürümü kullanmaları gerektiğine karar verirler. Sürüm numarasının sürüm içinde olan fonksiyonları ve değişiklikleri yansıtmaması durumunda, kullanıcılar potansiyel API değişikliklerinden dolayi çalışmaz hale gelebilirler.
- Yeni bir kütüphanenin sürüm numarası 0.1.0 ile başlamalıdır. 0.0.1 sürüm numarası oluşan yeni kütüphane için bugfix yapıldığı anlamına gelir ki bu doğru değildir.
- Bir fonksiyon kütüphanesinin değişik versiyonları bir yazılım deposu (repository) bünyesinde kullanım için tutulmalıdır. Bu şekilde kullanıcılar API değişikliklerinde yeni sürümü kullanmak zorunda bırakılmazlar. Hangi sürümü kullanacağına kullanıcı karar verebilmelidir.

Bir sürüm dosyasına baktığımızda, içinde neler olduğunu ilk bakışta görmek mümkün degildir. Versiyon numaraları kullanılarak, hangi sürüm dosyasının kullanılabileceğini kestirmek mümkün hale gelmektedir.

Hangi Programlama Dilini Öğrenmeliyim?

http://www.pratikprogramci.com/2014/11/18/hangi-programla-dilini-ogrenmeliyim/

Bana şu tarz mailler son zamanlarda çok gelir oldu:

"C# biliyorum, kariyerime Java ile devam etmek istiyorum. Sizce iş bulabilir miyim?", "PHP programcısıyım, Java'ya geçmemi tavsiye eder misiniz?", "Java öğrenirsem, iş bulma şansım artar mı?", "Geleceği aydınlık olmayan ActionScript ile yurt dışında mı, yoksa Türkiye'de kalıp Android programcılığı ile mi devam etmeliyim?"...

Ben bunları nasıl algıladığımı Java kodundan bir örnek vererek açıklamak istiyorum.

```
public static final String kullandigimDil ="Java";
```

Yukarıda yer alan örnekte kullandigimDil ismini taşıyan ve String veri tipinde olan bir değişken tanımladım. Final direktifi ile bu değişkene Java haricinde başka bir değer atanmasını engellemiş oldum. kullandigimDil her zaman Java değerine sahip olacak. Ben çoğu programcının da programlama dillerini seçerken, bu seçimin final olduğunu düşündüklerini düşünüyorum.

Her programlama dilinin güçlü ve zayıf olduğu alanlar bulunmaktadır. Örneğin Java dili her platformda çalışır olma özelliği ve zengin kütüphanesi sayesinde çok güçlü bir dil olarak görülebilir. Zayıf olduğu nokta ise bir problemi çözmek için gereğinden fazla kod yazmak zorunluluğudur. Bunun aşağıda bir örneğini görmekteyiz:

```
// Java
List<String> list = new ArrayList<String>();
list.add("1");
list.add("2");
list.add("3");
List<Integer> ints = new ArrayList<Integer>();
for (String s : list) {
    ints.add(Integer.parseInt(s));
}
```

Ayni kodu Scala dilinde bu sekilde yazabiliriz:

```
// Scala
val list = List("1", "2", "3")
val ints = list.map(s => s.toInt)
```

Java 8 ile yeni gelen dil özellikleri sayesinde daha kısa kod yazmak mümkün. Lakin verdiğim örnekte de görüldüğü gibi Scala dilinde bazı şeyleri yapmak çok daha kolay. Bunun başlıca sebebi Scala dilinin hem fonksiyonel hem de nesneye yönelik programlama paradigmalarını destekliyor olmasıdır. Scala dilinin fonksiyonel programlamayı desteklemesi sayesinde daha az kod yazarak, çözüme daha kısa zamanda ulaşmak mümkün.

Bir aşçı ustası mutfağında değişik işlemler için değişik araçlar kullanır. Örneğin bir patatesi soyduğu bıçak ile eti kestiği bıçak aynı değildir. Kemikli eti kesebilen bir bıçak ile patates soymak

mümkün olmakla birlikte, akıllıca bir seçim değildir. Bir patatesi soymak için kullanılan bıçak daha küçük ve ele sığacak şekildedir. O bıçak kullanılarak, büyük bir et bıçağına kıyasla patates çok daha az bir zamanda soyulabilir. Aynı şey programlama dilleri için de geçerlidir. Programlama dili yapılan işleme uygunluğuna göre seçilmelidir.

Duvara bir resmi asmak için kullanacağımız çivinin üzerine balyozla da vurabiliriz, ufak bir çekiçle de. Akılcı olan, elde edilmek istenen neticeyi en az eforla elde edebileceğimiz aracı kullanmaktır. Programlama dilleri de araçtırlar. Problem sahasına göre kullanılan programlama dili değişir/değişmelidir.

Buradan ben tek bir sonuç çıkarabiliyorum: **Programcının takım taklavat çantasında birden fazla programlama dili olmalıdır**. Birden fazla programlama diline hakim bir programcı hem programcılık konusundaki yetkinliğini ortaya koymaktadır, hem uygun aracı seçerek, kısa zamanda çözüm üretmektedir, hem de daha iyi programcı olmak için gerekli altyapıyı oluşturmaktadır.

Tek bir programlama dili ile bir ömür programcı olarak çalışmanın mümkün olmadığı bir devirde yaşıyoruz. Bilginin yarı ömrünün günlerle ölçüldüğü zamanımızda yeni programlama dilleri mantar gibi yerde bitiyorlar. Onları programcı olarak biz göz ardı etmeye çalışsak da, içinde bulunduğumuz sektör ve uygulama sahipleri sürekli daha az zamanda, daha çok işin nasıl yapılabileceği sorusunun cevabını arıyorlar. Hangi dilin kullanılacağını onlar belirliyorlar. Biz programcı olarak onları takip etmek zorundayız.

Çok Gezen mi Bilir, Çok Okuyan mı? başlıklı yazımda programcıların neden belli bir zaman sonra iş yerlerini değiştirmeleri gerektiği konusuna açıklık getirmeye çalıştım. Çok uzun bir zaman aynı firma için çalışmak programcının yeni bir şeyler öğrenmesinin önünde engeldir. Yeni bir şeyler öğrenmeyen programcının piyasa değeri sürekli düşer. Belli bir zaman sonra piyasa kullanılan yeni programlama dilleri ve araçlar nedeniyle çağ atlamıştır ve programcı bu gelişmelerden bihaberdir. Elektrik çağında elektrikle çalışan lokomotifler varken kimse su buharı ile çalışan lokomotifi kullanmak istemez.

Benim öğrenilmelerini tavsiye ettiğim diller şöyle:

- Java, çünkü nesneye yönelik programlama özelliği, platform bağımsızlığı, zengin kütüphanesi ve kurumsal projelerin gözdesi olduğu için,
- Clojure, çünkü LISP tarzı fonksiyonel programlamayı öğrettiği için,
- Groovy, çünkü daha kısa Java kodu yazmayı mümkün kıldığı için,
- Python, çünkü öğrenmesi çok kolay olduğu ve birçok iş için kullanılabildiği için,
- C, çünkü donanıma yakın alt seviye (low level) programlamayı mümkün kıldığı için,
- C#, çünkü .Net dünyasına girişi kolaylaştırdığı için,
- Javascript, çünkü web bazlı uygulamaların gözdesi olduğu için.

Sonuçta hangi dili öğrendiğiniz önemli değil. Önemli olan yeni diller öğrenebilme kabiliyetine ve azmine sahip olmanız. Yeni programlama dilleri öğrenmeyi iki sebepten dolayı gerekli görüyorum. Bunlar:

- Daha iyi bir programcı olabilmek için

- Piyasadaki rekabete ve teknolojik dönüşümlere göğüs gerebilmek için

Daha iyi programcı olmayı bir kenara bırakalım. Ama piyasanın talebine programcı olarak cevap veremiyorsak, o zaman bittik demektir. Umarım şimdi bu konunun ne kadar önemli olduğunun altını çizebilmişimdir. Eğer iki lisan iki insan ise, iki programlama dili, iki programcıdır. Bir backup oluşturun derim.

Kokan Kod – 1. Bölüm

http://www.pratikprogramci.com/2014/11/15/kokan-kod-1-bolum/

Söz konusu yemek olduğunda, insanlar kötü kokuları yemeğin formda olmadığının ibaresi olarak algılarlar. Aç kalmadıkça kötü koku saçan bir yemeği kimse yemez. Kötü kok yemeğin hangi durumda olduğunu gösteren bir işarettir. İnsanlar kötü kokan bir yemeğin yenmemesi gerektiğini bilirler.

Yazılımda da koku (smell) metaforu yeniden yapılandırılmaya ihtiyaç duyan kodlar için kullanılmaktadır. Kent Beck tarafından ilk kullanılan koku terimi Martin Fowler tarafından yazılan Refoctoring kitabıyla yazılım camiasında yapısal deformasyon içinde olan ve programcılar tarafından anlaşılması zor kod birimleri için kullanılır hale gelmiştir.

Kokan kod ilk etapta program hatalarına işaret etmez. Daha ziyada kokan kod terimi çalışır durumda olmasına rağmen anlaşılması ve geliştirilmesi çok zor kod birimleri için kullanılır. Kokan kod birimlerinde yapılan her yeni ekleme ile hata (bug) ihtimali yükselir. Kokan kod genelde kodun daha derinlerdeki yapısal problemlerini perdeler. Bu problemleri kesletmek için kodun yeniden yapılandırılması zaruridir.

Tasarım şablonlarında (design patterns) olduğu gibi kod kokularına isim vermek, bu konuda ortak bir kelime hazinesi oluşmasını kolaylaştıracaktır. Kod kokusunun oluştuğu durumları şu şekilde sıralayabiliriz:

- Kopyala/yapıştır yöntemiyle oluşmuş kod tekrarları (**Duplicated Code**)
- Uzun metotlar (**Long Method**)
- Birçok şeyden sorumlu büyük sınıflar (**Large Class**)
- Uzun metot parametre listeleri (**Too Many Parameters**)
- Temel veri tiplerine eğilim (**Primitive Obsession**)
- İç içe geçmiş karmaşık if/else/for/do/while direktifleri (**Cyclomatic Complexity**)
- Her şeyi yapmaya çalışan callback metotları (**Ubercallback**)
- Aynı sınıf bünyesinde farklı değişiklikler yapma zorunluluğu (**Divergent Changes**)
- Bir sınıf için diğer sınıflar bünyesinde farklı değişiklikler yapma zorunluluğu (**Shotgun Surgery**)
- Bir metodun kendi sınıfındaki değişkenler yerine başka bir sınıfın değişkenleri ile ilgilenmesi (**Feature Envy**)
- Switch komutunun kullanılması (**Switch Statements**)
- Paralel kalıtım hiyerarşileri oluşturulması (**Parallel Inheritance Hierarchies**)
- Sınıf olmayı hak etmeyen sınıflar (**Lazy Class**)
- Birgün belki işimize yarar sendromu (**Speculative Generality**)
- Belli şartlarda kullanılan sınıf değişkenleri (**Temporary Field**)
- Kendisine gelen metot çağrılarını başka sınıf metotlarına delege eden aracı sınıflar (**Middle Man**)
- Çok sıkı, fıkı olan sınıflar (**Inappropriate Intimacy**)
- Aynı şeyi yapan, ama değişik arayüzlere sahip olan sınıflar (**Alternative Classes with Different Interfaces**)

- İşini yapabilmek için değişikliğe ihtiyaç duyan kütüphane (3rd library) sınıfları (**Incomplete Library Class**)
- Demeter kuralına ters düşen sınıflar (**Message Chains**)
- Fonksiyon sahibi olmayan ve sadece veri tutan sınıflar (**Data Class**)
- Miras aldıkları metot ve değişkenlerin tümüne ihtiyaç duymayan alt sınıflar (**Refused Bequest**)
- Kötü durumdaki bir kod birimini açıklamak için oluşturulmuş yorumlar (**Comments**)
- İç dünyasını arayüzünün (interface) parçası olarak dışa açan sınıflar (**Indecent Exposure**)
- Bir şey ifade etmeyen sınıf, metot ya da değişken isimleri (**Uncommunicative Name**)
- Uzun sınıf, metot ya da değişken isimleri (**Long name**)
- Kısa sınıf, metot ya da değişken isimleri (**Short name**)
- Gereksiz tasarım şablonu (design pattern) kullanımı (**Contrived complexity**)
- Grup halinde sınıf değişkeni ya da metot parametresi olarak kullanılan nesneler (**Data Clumps**)

Kod kokuları ilk etapta teknik borçlanmaya işaret ederler. Kod kokularını yok etmek ve teknik borçlanmayı azaltmak için kullanılabilecek yöntemleri bu yazı dizisinde sizlerle paylaşmak istiyorum. Önümüzdeki günlerde listede yer alan her koku türünü inceleyen bir blog yazısı hazırlayacağım. Bu yazı dizisine giriş noktası olarak birinci bölüm niteliğinde olan bu blog yazısını kullanabilirsiniz.

Akıllı Ev Projem

http://smartev.im/2014/04/30/akilli-ev-projem/

Bir yazılımcının aklına inşaa edilen evi için ilk gelen soru hangisidir? Bu evi nasıl programlarım? :-)

Ve böylece akıllı evim projesi doğmuş olur.

Bir ev hayal edin, sizinle konuşan: "Özcan bey, banyonun ışığı bir saattir açık ve banyo kullanımda değil. Işığı kapatmamı ister misiniz?". "Özcan bey, evinize hoş geldiniz. Saat 18:45. Bugün siz evde yokken üç ziyaretçiniz vardı. Resimlerini çekip, cep telefonunuza gönderdim.", "Saat 22:00 ve istediğiniz şekilde evin tüm kepenklerini kapatıyorum. Kapılar kilitlendi ve alarm sistemi aktif hale getirildi. Size iyi uykular dilerim.", "Günaydın, saat 06:45. Tüm kepenkleri isteğiniz üzere kaldırıyorum…."

Bunların hepsi biraz bilim-kurgu gibi kulağa gelse de, teknik olarak bir evi bu şekilde otomatize etmek mümkün. Bu blog sayfasını, inşaatı süren yeni evimi akıllı bir eve dönüştürmek için yaptığım çalışmaları bu konuya ilgi duyanlarla paylaşmak için oluşturdum.

İnşaat yakında bitiyor ve ilk kabloları çekmeye ve ilk programları yazmaya başlayacağım. Yakında görüşmek üzere…

Akıllı Evlerde Kablo Kullanımı Neden Önemli?

http://smartev.im/2014/11/08/akilli-evlerde-kablo-kullanimi-neden-onemli/

Blog sayfamda akıllı ev projem hakkında bilgiler paylaşmaya başladıktan sonra, okurlarımdan genelde şu soruyu almaya başladım: Neden bu kadar kablo kullandınız? Bu blog yazımda bu soruya cevap vermeye çalışacağım.

Günümüzde lanse edilen akıllı ev ve ev otomasyonu çözümlerinin büyük bir kısmı (%99.9) kablosuz sistemlere dayanıyor. Bunun çok basit bir açıklaması var: Kimseye mevcut bir binada kablo döşetemezsiniz! Bu sebepten dolayı akıllı ev çözümleri sunan firmalar ürünlerini kablosuz ağ sistemlerine dayandırıyorlar. Bu tür sistemlerin kurulması ve yönetilmesi çok kolay.

Çoğu akıllı ev çözümünde yer alan komponentler arasındaki iletişimi sağlamak için oluşturulan kablosuz ağ güvenli değil. Kullanıcının hangi protokolün nasıl kullanıldığı bilmesi mümkün değil, çünkü çözüm sahibi firma bu bilgileri sır gibi saklama eğilimi gösterebiliyorlar. Bunun başlıca sebebi, popüler olmuş bir otomasyon ürünü için başka firmaların komponent ve servis sunmalarını engellemek. Bu şekilde ev otomasyonu çözümü üreten firmalar bu alandaki yatırımlarını korumak istiyorlar.

İnternet erişimi gibi konularda kablosuz ağ kullanmak bir yere kadar sorun teşkil etmiyor. WPA2 (Wi-Fi Protected Access) gibi standartlar güvenli kablosuz ağlar oluşturmak için kullanılabilir. Lakin ev otomasyonu gibi güvenlik açışından çok kritik olabilecek uygulamalarda WPA2'ye bile güvenmem. Birilerinin, sistemin bir açığını bularak, oturma odamdaki lambayı açıp, kapatmalarını istemiyorum! Bu verebileceğim en basit örnekti. Teoride kablosuz ağ oluşturarak ev otomasyonu yapan çözümlerde evin her yerinde otomasyonu dahil olmuş aletlere erişmek ve onları yönetmek mümkün. Artık gerisini siz düşünün!

Bu sebepten dolayı akıllı ev projemde tamamen kablo bazlı bir otomasyon çözümü oluşturma niyetindeyim. Sistem bünyesindeki aletlere erişebilmek için öncelikle evin içinde yer alan yerel kablolu ağa erişmek, yani sistemi sabote edebilmek için evin içinde olmak gerekiyor. Yerel ağa bir şekilde eriştiğinizi düşünelim. Daha sonra yerel ağın bir parçası haline gelmeniz gerekiyor. MAC adresi üzerinden yabancı bilgisayarların yerel ağın bir parçası olması engellenebilir. Hadi fake bir MAC adresi ile yerel ağa dahil olduğunuzu düşünelim. Bu durumda sistemin yönetildiği merkezi sunucuyu bulmanız gerekiyor. Hadi bu sunucuyu da buldunuz diyelim. Yönetim paneline erişmek için kullanıcı ismini ve şifresini bilmeniz gerekiyor. Görüldüğü gibi kablolu bir ağ sistemine dayalı ev otomasyonuna erişim izni olmayan şahısların müdahale etmesi o kadar kolay değil.

Bu sebepten dolayı bu tür ev otomasyonu yapmak isteyen okurlarıma mutlaka kablonu sistemleri tavsiye ediyorum. Bu da ne yazık ki yeni inşa edilen binalarda mümkün. Mevcut bir evde bu çapta kablo döşemek ne yazık ki mümkün değil.

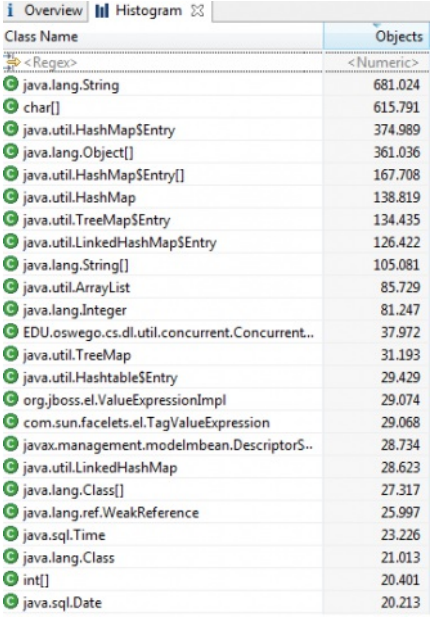
Java String Nesnelerinin Hafıza Kullanımı Nasıl Azaltılır?

http://www.pratikprogramci.com/2014/09/18/java-string-nesnelerinin-hafiza-kullanimi-nasil-azaltilir/

Bir Java uygulaması için Java sanal makinesi (JVM) tarafından oluşturulan ve yönetilen hafıza alanının (heap) ortalama %25'ini String nesneleri kaplar. Bir heapdump oluşturduğumuzda, String nesneleri ve String nesnelerini oluşturan char[] arraylerin ilk sıralarda olduğunu görebiliriz. Şu şekilde örneğin çalışan bir Java uygulamasının hafıza resmi alınabilir.

```
jmap -dump:live,format=b,file=<filename> <PID>
```

Eclipse MAT ile heapdump çıktısını inceledigimizde, şu şekildeki bir resimle karşılaşmamız muhtemel:



Bu örnekte hafıza alanının yaklaşık olarak %25'inin String nesneleri ve String sınıfında yer alan char[] array nesneleri tarafından kullanıldığını görmekteyiz. Uygulamanın yapısına göre bu değer %40'lara kadar çıkabilir.

Java 8, 20 update sürümü ile String nesnesi kullanım oranlarını aşağıya çekmek mümkün. Bu sürüm ile G1 garbage collector tarafından kullanıma sunulan String deduplication mekanizmasından faydalanarak, String nesne kullanım oranını düşünebiliriz.

String deduplication ile aynı yapıda olan (s1.equals(s2) == true) birden fazla String nesnesi aynı char[] array nesnesini ortak kullanacak şekilde garbage collector tarafından yeniden

yapılandırılmaktadırlar. String sınıfının aşağıdaki şekilde iki değişkeni bulunmaktadır:

```
private final char value[];
private int hash; // Default to 0
```

Bir String nesnesini oluşturan harfler value[] array nesnesi içinde yer alırlar. Her harf için 2 byte değerinde hafıza alanına ihtiyaç duyulmaktadır. Dışarıdan value[] nesnesine erişmek mümkün değildir. Bu yüzden String nesneleri değiştirilemez. Lakin String sınıfına baktığımızda, sınıf bünyesinde de value[] arrayi üzerinde işlem yapılmadığını görmekteyiz, yani value[] arrayi ne içten, ne de dıştan değişikliğe uğramaktadır. Bu durumda birden fazla String nesnesini aynı value[] array nesnesini kullanacak şekilde yeniden yapılandırmak mümkündür. Örneğin aynı içeriğe sahip olan on değişik String nesnesi sadece bir char[] array nesnesine işaret edecektir. Bu hafızada on yerine sadece bir adet char[] array nesnesi oluşturulması anlamına gelmektedir. Bu şekilde hafıza alanından tasarruf yapılabilir.

String deduplication G1 garbage collector tarafından yapılabilen bir işlemdir. Programcı olarak dışarıdan bu mekanizmaya müdahil olmamız mümkün değil. Sadece gerekli JVM ayarlarını yaparak, bu işlemi başlatabiliriz. G1 garbage collector String nesnelerinin hash değerlerini oluşturur ve zayıf bir referans (weak reference) ile String nesnesi ile char[] array arasındaki ilişkiyi yeniden yapılandırır. G1 hafıza alanını temizlerken String nesnelerinin hash değerlerini kıyaslar. Eğer iki String nesnesinin hash değerleri aynı ise, aynı içeriğe sahip olma ihtimalleri yüksektir. Bu durumda G1 iki nesnenin value[] array içeriklerini kıyaslar. Eğer iki value[] array aynı değere sahipse, G1 elinde tuttuğu String nesnesinin value[] arrayini kıyasladığı value[] arrayi gösterecek şekilde yeniden yapılandırır. Bu durumda ilk String nesnesinin value[] nesnesi boşta kalır. Zayıf bir referansa sahip olan bu nesne bir sonraki garbage collection işleminde hafızadan silinir.

Şimdi bu mekanizmanın nasıl işlediğini bir örnek üzerinde inceleyelim.

```
package com.pratikprogramci.string;

import java.util.LinkedList;

public class StringDeduplicationTest {

    private static final LinkedList<String> STRING_LIST =
                            new LinkedList<>();

    public static void main(String[] args) throws Exception {
        int iteration = 0;

        while (true) {

            for (int i = 0; i < 10; i++) {

                for (int j = 0; j < 10000; j++) {
                    STRING_LIST.add(new String("Test String " + j));
                }
            }
            iteration++;
            System.out.println("Iterasyon sayisi: " + iteration);
            Thread.sleep(100);
        }
    }
}
```

Yukarıda yer alan kodu -Xmx128m -XX:+UseG1GC JVM parametreleri ile koşturduğumuzda, şu ekran çıktısını alırız:

```
Iterasyon sayisi: 1
Iterasyon sayisi: 2
Iterasyon sayisi: 3
Iterasyon sayisi: 4
Iterasyon sayisi: 5
Iterasyon sayisi: 6
Iterasyon sayisi: 7
Iterasyon sayisi: 8
Iterasyon sayisi: 9
Iterasyon sayisi: 10
Iterasyon sayisi: 11
Iterasyon sayisi: 12
Iterasyon sayisi: 13

Exception: java.lang.OutOfMemoryError thrown from the
            UncaughtExceptionHandler in thread "main"
```

Uygulamayı 128 MB hafıza alanı ile başlattık. Sonsuz bir döngü içinde binlerce String nesnesi oluşturuyoruz. Uygulamamız 13 iterasyondan sonra OutOfMemoryError hatası ile son buldu, cünkü kullanabileceğimiz hafıza alanı kalmadı.

Şimdi uygulamayı String deduplication mekanizması ile çalıştıralım. Aşağıdaki JVM parametreleri kullanıyoruz:

```
-Xmx128m -XX:+UseG1GC -XX:+UseStringDeduplication
        -XX:+PrintStringDeduplicationStatistics
```

Uygulamamızı tekrar çalıştırdığımızda, şöyle bir ekran çıktısı alırız:

```
Iterasyon sayisi: 24
[GC concurrent-string-deduplication, 1998,8K->0,0B(1998,8K), avg 99,5%, 0,0101200 secs]
   [Last Exec: 0,0101200 secs, Idle: 0,1232091 secs, Blocked: 3/0,5220587 secs]
      [Inspected:           42641]
         [Skipped:              0(  0,0%)]
         [Hashed:           42641(100,0%)]
         [Known:                0(  0,0%)]
         [New:              42641(100,0%)   1998,8K]
      [Deduplicated:        42641(100,0%)   1998,8K(100,0%)]
         [Young:                0(  0,0%)      0,0B(  0,0%)]
         [Old:              42641(100,0%)   1998,8K(100,0%)]
   [Total Exec: 17/0,7226112 secs, Idle: 17/2,6424357 secs, Blocked: 16/1,4228223 secs]
      [Inspected:         2112793]
         [Skipped:              0(  0,0%)]
         [Hashed:         2111921(100,0%)]
         [Known:              767(  0,0%)]
         [New:            2112026(100,0%)     96,7M]
      [Deduplicated:      2101850( 99,5%)     96,2M( 99,5%)]
         [Young:               21(  0,0%)   1008,0B(  0,0%)]
         [Old:            2101829(100,0%)     96,2M(100,0%)]
   [Table]
      [Memory Usage: 320,5K]
      [Size: 8192, Min: 1024, Max: 16777216]
      [Entries: 10803, Load: 131,9%, Cached: 140, Added: 10943, Removed: 140]
      [Resize Count: 3, Shrink Threshold: 5461(66,7%), Grow Threshold: 16384(200,0%)]
      [Rehash Count: 0, Rehash Threshold: 120, Hash Seed: 0x0]
      [Age Threshold: 3]
   [Queue]
      [Dropped: 0]

Exception: java.lang.OutOfMemoryError thrown from the UncaughtExceptionHandler in thread "m
```

İterasyon sayısının 13'den 24'e yükseldiğini görmekteyiz. Aynı şartlar altında uygulamamız daha uzun soluklu çalıştı. Bunun yanı sıra -XX:+PrintStringDeduplicationStatistics parametresi ile deduplication istatistiklerini görmekteyiz. Son iterasyonda G1 42641 adet String nesnesini 0,0101200 saniyede analiz etti. Bu analiz sonucunda 1998,8 KB hafıza alanını tekrar kullanılmak üzere temizlendi. Lakin bu uygulamanın OutOfMemoryError ile son bulması için yeterli olmadı.

Çalışan Bir Java Uygulamasında Bytekod Nasıl Değiştirilir?

http://www.pratikprogramci.com/2014/11/02/calisan-bir-java-uygulamasinda-bytekod-nasil-degistirilir/

Çalışan Bir Java Uygulamasında Bytekod Nasıl Değiştirilir?

Java uygulamaları bytekoduna derlendikten sonra Java sanal makine (JVM – Java Virtual Machine) bünyesinde koşturulur. Bu yazımda çalışan bir Java uygulamasında mevcut bytekodun nasıl değiştirilebileceğini bir örnek üzerinde göstermek istiyorum.

Hangi durumlarda çalışan bir uygulama için bytekod değiştirme işlemi gerekli olabilir? Benim aklıma gelenler:

- Kaynak dosyaları olmayan yabancı kütüphaneler üzerinde değişiklik yapılmak istendiğinde,
- Loglama ve transaksiyon yönetimi gibi işletme mantığının doğrudan parçası olmayan işlemlerin yapılması gerektiğinde,
- Performans ölçümleri için,
- Kodu tekrar derlemek mümkün olmadığında,
- Uygulama hakkında istatistiksel bilgiler toplanmak istendiğinde.

Bytekodun nasıl değiştirilebildiğini şimdi küçük bir örnek üzerinde inceleyelim. Aşağıda bytekodunu değiştirmek istediğimiz HelloWorld sınıfı yer alıyor.

```
package com.pratikprogramci.jvm.agent;

public class HelloWorld {
    public static void main(final String args[]) {

        new HelloWorld().run();
    }

    public void run() {
        System.out.println("Merhaba Dünya");
    }
}
```

run() metodunun hangi zaman diliminde koştuğunu tespit etmek istediğimizi ve bu sınıfın koduna sahip olmadığımızı düşünelim. Eğer bu sınıfın kaynak dosyasına sahip olsaydık, şöyle bir kod değişikliği ile run() metodunun hangi zaman diliminde koşturulduğunu ölçebilirdik:

```
package com.pratikprogramci.jvm.agent;

public class HelloWorld {
    public static void main(final String args[]) {
        new HelloWorld().run();
    }

   public void run() {
        long executionTime = System.currentTimeMillis();
        System.out.println("Merhaba Dünya");
        executionTime = System.currentTimeMillis() - executionTime;
        System.out.println("Metot " + executionTime + " ms icinde tamamlandi");
    }
}
```

Yukarıda yer alan kod örneğinde görüldüğü gibi run() metoduna executionTime isminde yeni bir değişken ekledik. System.currentTimeMillis() ile metoda girdiğimizde mevcut zamanı tutarak, metot son bulmadan önce gecen zamanı hesapladık.

Kodsal seviyede yaptığımız bu değişiklikleri, bytekod seviyesinde yapabilir miyiz? Evet! Bunun için iki şeye ihtiyacımız var:

- Java Instrumentation API
- Bytekodunu değiştirmek için kullanılabilecek Javasist ya da ASM çatısı

Önce Java Instrumentation API ile başlayalım. Bir sınıfın sahip olduğu bytekodu değiştirebilmek için bu sınıfa JVM bünyesinde erişmemiz gerekmektedir. Bu işlem için bir agent sınıfı kullanabiliriz. JVM bünyesindeki bir sınıf bir classloader tarafından yüklendiği zaman, aşağıda yer alan agent sınıfının premain() metodu devreye girer.

```
package com.pratikprogramci.jvm.agent;

import java.lang.instrument.ClassFileTransformer;
import java.lang.instrument.IllegalClassFormatException;
import java.lang.instrument.Instrumentation;
import java.security.ProtectionDomain;

public class LoggerAgent implements ClassFileTransformer {

    public static void premain(final String agentArguments,
                               final Instrumentation instrumentation) {
        instrumentation.addTransformer(new LoggerAgent());
    }

    public byte[] transform(final ClassLoader classloader,
                    final String className, final Class<?> clazz,
                    final ProtectionDomain domain, final byte[] bytes)
                          throws IllegalClassFormatException {
        return null;
    }
}
```

Mevcut bir sınıfın sahip oldugu bytekodu değiştirebilmek icin Java Instrumentation API'sinin bir

parçası olan ClassFileTransformer interface sınıfını implemente etmemiz gerekiyor. Yukarıda yer alan LoggerAgent sınıfı hem bir JVM agent olma özelliğine sahiptir hem de ClassFileTransformer interface sınıfını implemente ederek, bytekodu değiştirme işleminden sorumlu sınıf haline gelmektedir.

ClassFileTransformer interface sınıfında transform() isminde, yüklenen sınıf üzerinde gerekli değişikliklerin yapıldığı bir metot bulunmaktadır. Bu metodun parametrelerine göz attığımızda, ClassLoader tipinde bir parametrenin olduğunu görmekteyiz. Bu değiştirmek istediğimiz sınıfı yükleyen sınıf yükleyicisidir. Bunun yanı sıra transform() metoduna yüklenen sınıfın ismi ve Class nesnesi olarak kendisi parametre olarak verilmektedir. ProtectionDomain sınıfı belli sınıflardan ve bu sınıflara olan erişim haklarının tanımlandığı bir uygulama alanını (domain) temsil etmektedir. byte[] parametresi yüklenen Java sınıfının bytekodunu byte formatında ihtiva etmektedir. transform() metodu değişikliğe uğrayabilecek sınıfı yine byte[] veri tipinde geriye vermektedir.

Şimdi Javasist bytekod değiştirme çatısı yardımı ile HelloWorld sınıfı üzerinde düşündüğümüz değişiklikleri yapalım. Bu amaçla transform() metodunu aşağıdaki şekilde implemente ediyoruz:

```
public byte[] transform(final ClassLoader classloader, final String className,
        final Class<?> clazz,
        final ProtectionDomain domain, final byte[] bytes)
                throws IllegalClassFormatException {

    if (className.equals("com/pratikprogramci/jvm/agent/HelloWorld")) {
        try {

            System.out.println(">>Bytecode enjeksiyonu basliyor.....");

            final ClassPool cp = ClassPool.getDefault();
            final CtClass cc = cp.get("com.pratikprogramci.jvm.agent.HelloWorld");
            final CtMethod m = cc.getDeclaredMethod("run");
            m.addLocalVariable("executionTime", CtClass.longType);

            m.insertBefore("executionTime = System.currentTimeMillis();");
            m.insertAfter("{executionTime = System.currentTimeMillis() - "
                    + "executionTime;"
                    + "System.out.println(\">>Metot \" + executionTime + "
                    + "\" ms icinde tamamlandi.\" );}");

            final byte[] byteCode = cc.toBytecode();
            cc.detach();
            return byteCode;

        } catch (final Exception ex) {
            ex.printStackTrace();
        }
    }
    return null;
}
```

transform() metodunda değiştirmek istediğimiz sınıfı if komutu ile tespit ettikten sonra, değiştirmek istediğimiz metodu getDeclaredMethod() ile lokalize ediyor ve run() metoduna

addLocalVariable() ile executionTime isminde yeni bir değişken ekliyoruz. insertBefore() ile metoda giriş yapılmadan önce executionTime değişkenine içinde bulunduğumuz zamanı atıyoruz. insertAfter() ile run() metodu son bulduktan sonra geçen zamanı hesaplayacak kodu oluşturuyoruz. Bu değişikliklerin ardından bytekodunu değiştirdiğimiz sınıfı byte[] olarak cc.toBytecode() metodu yardımıyla geri veriyoruz. Bu noktada itibaren JVM oluşturduğumuz yeni sınıfı koşturmaya başlıyor.

LoggerAgent sınıfı için bir MANIFEST.MF dosyası oluşturmamız gerekiyor. Aşağıda yer alan MANIFEST.MF dosyasında Main-Class parametresi ile koşturmak istediğimiz sınıfı tanımlıyoruz. Premain-Class parametresi agent sınıfını tanımlamak için kullanılmaktadır. Boot-Class-Path parametresi ile agent tarafından kullanılan kütüphaneleri tanımlamak mümkün. Bizim örneğimizde kullanılan tek kütüphane Javasist kütüphanesidir.

```
Main-Class: com.pratikprogramci.jvm.agent.HelloWorld
Premain-Class: com.pratikprogramci.jvm.agent.LoggerAgent
Boot-Class-Path: lib/javassist-3.18.2-GA.jar
```

Şimdi oluşturduğumuz tüm sınıfların ve MANIFEST.MF dosyasının yer aldığı bir Jar dosyası oluşturmamız gerekiyor. Jar dosyasını oluşturmak ve gerekli bağımlılıkları yönetmek için bir Maven projesi oluşturdum. Pom.xml şu yapıda:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0

http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>
    <groupId>com.pratikprogramci</groupId>
    <artifactId>jvm.agent</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <dependencies>
        <dependency>
            <groupId>org.javassist</groupId>
            <artifactId>javassist</artifactId>
            <version>3.18.2-GA</version>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-jar-plugin</artifactId>
                <configuration>
                    <archive>
                        <manifestFile>src/main/resources/
                                                META-INF/MANIFEST.MF</manifestFile>
                    </archive>
                </configuration>
            </plugin>
            <plugin>
                <artifactId>maven-dependency-plugin</artifactId>
                <executions>
                    <execution>
                        <phase>install</phase>
                        <goals>
                            <goal>copy-dependencies</goal>
                        </goals>
                        <configuration>
                            <outputDirectory>${project.build.directory}/
                            lib</outputDirectory>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

Proje yapısı aşağıda görüldüğü şekilde olacaktır. Projenin ana dizini içinde mvn clean install ile uygulamayı derleyerek, bir Jar dosyası haline getirebiliriz.

```
acar@acarnb:~/Development/workspace/jvm.agent> dir
-rw-r--r-- 1 acar users 1205 31. Okt 16:15 pom.xml
drwxr-xr-x 4 acar users 4096 31. Okt 15:07 src
drwxr-xr-x 7 acar users 4096 31. Okt 16:23 target
acar@acarnb:~/Development/workspace/jvm.agent>
```

Jar dosyası target dizininde yer almaktadır. cd target ile bu dizine geçiyoruz:

```
acar@acarnb:~/Development/workspace/jvm.agent/target> dir
drwxr-xr-x 4 acar users 4096 31. Okt 16:23 classes
-rw-r--r-- 1 acar users 4476 31. Okt 16:23 jvm.agent-0.0.1-SNAPSHOT.jar
drwxr-xr-x 2 acar users 4096 31. Okt 16:23 lib
drwxr-xr-x 2 acar users 4096 31. Okt 16:23 maven-archiver
drwxr-xr-x 2 acar users 4096 31. Okt 16:23 surefire
drwxr-xr-x 2 acar users 4096 31. Okt 16:23 test-classes
acar@acarnb:~/Development/workspace/jvm.agent/target>
```

jvm.agent-0.0.1-SNAPSHOT.jar oluşturduğumuz Java sınıflarını ihtiva eden Jar dosyasıdır. Bunun yanı sıra lib dizini içinde javassist-3.18.2-GA.jar kütüphanesi yer almaktadır. Maven maven-dependency-plugin aracılığı ile bu kütüphanenin target/lib dizinine kopyalanmasını sağladık. MANIFEST.MF dosyasına tekrar göz attığımızda, Boot-Class-Path parametresinin lib dizininde bulunan javassist-3.18.2-GA.jar dosyasına işaret ettiğini görmekteyiz. LoggerAgent sınıfının bytekodu işlemlerini yapabilmesi için javassist-3.18.2-GA.jar kütüphanesine ihtiyaç duymaktadır.

Agent sınıfının JVM tarafından yüklenebilmesi için -javaagent parametresi kullanılmaktadır. -javaagent parametresi değer olarak içinde agent sınıfının ve MANIFEST.MF dosyasının yer aldığı Jar dosyasını almaktadır. MANIFEST.MF dosyası bünyesinde premain() metodunu taşıyan sınıf tanımlandığından, JVM hangi agent sınıfını yüklemesi gerektiğini bilmektedir.

HelloWorld uygulamasını şu şekilde koşturabiliriz:

```
acar@acarnb:~/Development/workspace/jvm.agent/target> java -jar jvm.agent-0.0.1-SNAPSHOT.ja
Merhaba Dünya
acar@acarnb:~/Development/workspace/jvm.agent/target>
```

Görüldüğü gibi sadece Merhaba Dünya çıktısını aldık. Şimdi uygulamamızı bytekod manipülasyonu için oluşturduğumuz LoggerAgent sınıfı ile koşturalım:

```
acar@acarnb:~/Development/workspace/jvm.agent/target> java -javaagent:jvm.agent-0.0.1-SNAPS
>>Bytecode enjeksiyonu basliyor.....
Merhaba Dünya
>>Metot 2000 ms icinde tamamlandi.
acar@acarnb:~/Development/workspace/jvm.agent/target>
```

Görüldüğü gibi LoggerAgent sınıfının transform() metodu devreye girdi ve çalışan bir uygulamada gerekli bytekod değişikliğini yaptı. Ekran çıktısında metodun koşma zamanını 2000 ms olarak görmekteyiz. HelloWorld sınıfının run() metoduna uygulamayı 2 saniye durduran

Thread.sleep(2000); eklentisini yaptım. Bu eklenti olmadan 0 ms değerini görülürüz, cünkü run() metodu JVM tarafından bir milisaniyenin bile altında bir zaman diliminde koşturulmaktadır.

Maven projesini aşağıdaki linkten indirebilirsiniz:

Link

Teknik Borç Nedir ve Nasıl Ödenir?

http://www.pratikprogramci.com/2014/10/19/teknik-borc-nedir-ve-nasil-odenir/

Her borçlanmanın sonu, bedeli ödenmediği zaman iflastır. Aynı şey yazılım projeleri için de geçerli. Teknik borcu ödenmeyen projelerin başarıyla tamamlanmaları ya da sürdürülebilmeleri mümkün değildir. Bu yazımda projelerde yaşanan teknik borçlanmalardan ve bu borçların nasıl ödenebileceğinden bahsetmek istiyorum. Önce teknik borcun tanımını yaparak başlayayım.

Teknik borçlanma uygulamanın kalitesinden ödün vermektir. Geliştirme sürecinde kalitesi düşen uygulamada teknik borçlanma artar. Teknik borçlanma kötü kaliteye sahip bir uygulamanın, iyi kaliteye sahip bir uygulama haline dönüştürülmesi için planlanması gereken çalışmaların neden olduğu maliyettir.

Teknik borçlanmanın ana sebepleri şunlardır:

- Sürüm oluşturma baskısı
- Projenin sahip olduğu kalite kontrol mekanizmalarının yetersiz olması.
- Ekibin teknik yetersizliği.
- Kodun yeni müşteri isteklerini tatmin edecek şekilde yeniden yapılandırılmaması (refactoring).
- Uygulamanın sürekli entegre edilmemesi.

İki teknik borçlanma türü vardır. Bunlar:

- Bilinçli olarak teknik borçlanma
- Bilinçsiz olarak teknik borçlanma

Sürüm oluşturma baskısı bilinçli teknik borçlanmaya gidilmesini beraberinde getirebilir. Sürümü yetiştirmek için her şey mubahtır filozofisi teknik borçlanmayı kolaylaştırır. Sürüm oluşturulduktan sonra teknik borçlanmalar gerekli yapısal değişikliklerle ödenebilir. Bu yapılmadığı taktirde teknik borçlanmanın faizi işlemeye başlar. Çoğu projede teknik borçlanma bilinçli olarak gerçekleşir. Bu bilincin var olmasına rağmen birçok projede teknik borçların ödenmesi konusunda bilinçli bir çalışma yapılmaz.

Bilinçsiz olarak gerçekleşen teknik borçlanmanın temelinde teknik yetersizlik yatar.

Teknik Borçlanma Örnekleri

Orta ve büyük çaplı projelerde tasarım prensipleri uygulanmadığı taktirde, doğrudan teknik borçlanma oluşur. Bunun en güzel örneğini tek sorumluluk prensibi oluşturmaktadır.

Tek sorumluluk prensibine göre bir sınıf ya da bir metot sadece bir işten sorumlu olmalıdır. Bu prensibe uyulmadığı taktirde, birden fazla işten sorumlu olan sınıf ve metotlar oluşur. Birden fazla sorumluluğu olan sınıf ve metotlar kırılgandır. Mevcut bir sınıfa ikinci bir sorumluluk yüklendiği andan itibaren teknik borçlanma başlar, çünkü sınıfın sahip olması gereken teknik yapıdan taviz verilmiştir. Burada tek sorumluluk prensibine uygun bir yapının oluşturulması için gerekli zaman dilimi teknik borcun miktarını belirlemektedir. Eğer bu anomaliye anında

müdahale edilmezse, teknik borçlanma oluşur ve sınıf bu teknik borcu geleceğe doğru sırtında taşımaya başlar. Sınıf için gerekli teknik yapı düzeltilmediği sürece teknik borçlanma devam eder. Bu sınıfa yeni bir sorumluluk verildiğinde, teknik borçlanma artar.

Yazılım projelerinde en büyük sıkıntılar uygulama entegrasyonlarında yaşanır. Yazılım geliştirme sürecinde ileriye atılan entegrasyon çalışmaları için her an teknik borçlanma gerçekleşir. Şimdi bu ve buna benzer teknik borçlanmaların nasıl ödenebileceğini inceleyelim.

Teknik Borçlar Nasıl Ödenir?

Gelişim sürecinde bulunan bir uygulama sürekli değişikliğe maruz kalır. Bunun başlıca sebebi, müşteri gereksinimlerinin piyasa koşulları doğrultusunda değişmesidir. Yazılımcı olarak öncelikle değişimin kendisini sabit ve değişmeyen bir parametre olarak kabullenmemiz gerekiyor. Buradan uygulamanın sürekli yeni özellikler eklenerek, yoğrulabilir yapıda olması gerektiği sonucu çıkarabiliriz.

Bir uygulamanın istenildiği şekilde yoğrulabilmesi için bazı teknik özelliklere sahip olması gerekmektedir. Bir uygulamayı yeniden yapılandırmada kullanılabilecek en güçlü silah testlerdir. Birim ya da entegrasyon türünde testleri olmayan bir uygulamayı yeniden yapılandırmak yani hamur gibi yoğurmak mümkün değildir, çünkü yapılan değişikliklerin yan etkilerini ölçmek mümkün olmaz.

Teknik borçlanmayı ödemek demek, uygulamayı hak ettiği teknik seviyeye getirmek demektir. Teknik borçların ödenebilir olmalarını sağlamak için uygulamanın her an yeniden yapılandırılabilir yapıda olması gerekmektedir. Bunu mümkün kılan uygulama testleridir. Testlerin olmadığı ya da yetersiz oldugu bir uygulamada teknik borçların ödenmesi imkansızdır. Bu er ya da geç uygulamanın belli bir büyüklükten sonra bu borcun altında ezilmesi ile kendisini gösterecektir.

Tasarım prensiplerinin uygulanması teknik borçların ödenmesini kolaylaştırır ya da oluşmalarını engeller. Tek sorumluluk prensibine uyulmamasının teknik borçlanmayı nasıl çabuklaştırdığını daha önce inceledik. Aynı şekilde örneğin açık, kapalı prensibinin (Open Closed Principle) uygulanması, uygulamanın teknik borçlanmaya gerek kalmadan, gelecekteki değişikliklere açık olmasını sağlar.

Teknik borçları ödemenin en kolay yolu, oluşmalarını engellemektir. Extreme Programming gibi çevik yazılım süreçleri bu amaçla uygulanabilecek teknik metotlar ihtiva etmektedirler. Örneğin sürekli entegrasyon (continuous integration) yöntemiyle entegre etmemekten doğan teknik borçların oluşması engellenir. Test güdümlü (test driven development) yazılım testlerin oluşmasını, kod kapsama alanının geniş olmasını ve bu şekilde uygulama için gerekli adaptasyonların çok daha kolay yapılabilmelerini beraberinde getirir.

Teknik borçlanma kodun bakımı ve geliştirilebilirliğini doğrudan etkiler. Ciddiye alınmadığı taktirde projenin başarısız olmasını beraberinden getirir. Yazılım ekibi teknik borçları sürekli gözetlemekle yükümlüdür. Bu amaçla örneğin Sonar kullanılabilir. Sonar aracılığı ile kodun teknik analizi yapılır. Bilinçsizce yapılan teknik borçlanmaları Sonar ile keşfetmek mümkündür. Örneğin modüller arası döngüsel bağlar sıkça oluşan teknik borçlanmaların başında gelir. Koda

bakarak, bu tür teknik borçlanmaları keşfetmek kolay değildir. Bu yüzden Sonar gibi bir aracın kullanılması, teknik borçlanmanın önüne geçebilmek ya da artmasını engellemek için zaruridir.

Teknik borçlanmanın sürüm baskısı gibi dışsal sebepleri olsada, teknik borçlanmaya sebep veren kodun yazarıdır. Bu sebepten dolayı kodu yazanların bu konuda hassas davranmaları gerekmektedir. Örneğin izci kuralına uyulduğu taktirde, teknik borçların ödenmesi yazılımın bir parçası haline gelir.

Teknik borçlanma yazılımda kullanılan ve uygulamanın içinde bulundugu teknik duruma işaret eden bir metafordur. Bu metafor ekip içinde bu konu hakkında fikir alışverişini sağlar ve yapısal değişikliklerin her zaman gerekli olduğu bilincinin oturmasını kolaylaştırır.

Başkalarının Kodu Okunarak Daha İyi Programcı Olunabilir mi?

http://www.pratikprogramci.com/2014/09/30/baskalarinin-kodu-okunarak-daha-iyi-programci-olunabilir-mi/

Son zamanlarda programcılara sıkça verilen bir öğüt var: **Bol, bol başkalarının yazdıkları kodları okuyun.** Bu sizin daha iyi programcı olmanızı sağlayacaktır.

Başkalarının kodunu okumamız söylenir, ama bu okumanın bizi programcı olarak neden ileri götürdüğüne açıklık getirilmez.

Benim savım: **Başkalarının kodunu okumak bizi daha iyi bir programcı yapmak için yeterli bir aktivite değildir.** Ben bir noktadan sonra başkalarının kodunu okumanın zaman kaybı olduğunu düşünüyorum. Bunun neden olduğunu bu yazımda açıklamaya çalışacağım.

Ben çok nadir programcının kodu başka programcılar okuyabilsin diye yazdığına şahit oldum. Çoğu programcı kodu doğrudan mikro işlemci için yazar. Sadece bu sebepten dolayı bile başkasının kodunu okumak anlamlı değildir, çünkü okunmak için yazılmamış kodun içinde olup, bitenleri kavramak imkansız ya da zordur. O kod karmaşasını okumaya çalışmak programcıya bir şey katmaz, daha ziyade boşuna kafa patlatarak, yorulmasına sebep olur ve demotive eder.

Zaman içinde bazı kod bölümleri optimize edilir. Bu işlemin sonucunda kullanılan algoritmalar optimizasyon kurbanı olur ve botokslanmış bir surat gibi tanınmaz hale gelirler. Programcı olarak kodun bazı şartlarda gerekli davranışı sergileyebilsin diye optimize edildiğini ilk bakışta göremeyebiliriz. Şekil değiştirmiş olan algoritmaları da tanımamız artık mümkün değildir. Optimizasyon neticesinde tanınmaz hale gelen algoritmaları anlamak ve nasıl kullanıldıklarını kavramak mümkün değildir. Bu yüzden ne olduğunu bile kavrayamadığımız bir algoritmanın nasıl kullanıldığını anlamaya çalışmak beyhude bir iş olabilir.

Aynı şey deneme, yanılma usulüyle ortaya çıkmış algoritma-vari yapılar için de geçerlidir. Programcı büyük bir ihtimalle kendisinin de bir zaman sonra anlamakta zorluk çektiği bir yapıyı deneme, yanılma usulüyle oluşturmuş olabilir. Kodun çalışıyor olması, ortada bir algoritmanın olduğu anlamına gelmez. Geliştiricisinin bile büyük bir ihtimalle bir zaman sonra anlamakta güçlük çektiği bir kod birimini bizim okumaya çalışmamız, bize ne kadar fayda sağlayabilir?

Uygulamanın nasıl çalıştığını anlamanın en kolay yolu, önce birim/entegrasyon/onay-kabul testlerini incelemek ve bu testleri koşturmaktır. Bu şekilde kullanıcı gözüyle uygulamanın nasıl işlediğini anlamak mümkün olacaktır. Testler programcının uygulamanın belli noktalarından giriş yaparak, uygulamanın nasıl çalıştığını anlamasını kolaylaştırır. Birçok projede bir can simidi değil de, lüks olarak görülen testlere rastlamak neredeyse mümkün değildir. Testler olmadan kodu okuyup, anlamaya çalışmak, rastgele birkaç kod dosyasını açıp, içine bakmadan öteye gidemez. Kod okurken nihai amacımız if, while, switchlerin nasıl kullanıldığını görmek değil, belli bir problemin hangi modele dayandırılarak çözüldüğünü anlamaya çalışmak olmalıdır. Testler bu genel resmi görmemizi sağlarken, testleri olmayan bir uygulamayı incelemeye çalışmak, programlama dilinin nasıl kullanıldığını anlamaktan öteye gitmeyecektir.

Kodun okunamaz hale gelmesinin bir başka nedeni de, zaman içinde değişik türdeki hataların (bug) değişik programcılar tarafından yamalanmasıdır (bugfix). Kod bünyesinde olup, bitenleri anlamayı engellemenin en emin yolu bu yamaların oluşturulmasıdır.

Kod okuyarak daha iyi programcı olma konusuna bu kadar olumsuz baktığımı düşünmeyin. Bu aktivitenin yapılış tarzına göre programcıya katkıda bulunması mümkün. Buraya kadar verdiğim örnekler pasif kod okumayla ilgiliydi.

Kod okuma seanslarından kazanç sağlayabilmek için kodun **aktif** bir şekilde okunması gerekmektedir. Aktif kod okuma seanslarında testler çalıştırılır ve uygulamanın giriş noktaları keşfedilir. Bu giriş noktalarını uygulamanın temelinde yatan prensipleri anlamak için kullanabiliriz. Büyük bir ihtimalle gördüklerimizi ilk etapta anlamamız mümkün olmayacaktır. Gördüklerimize anlam verebilmek için kodu sahip olduğumuz tarza göre yeniden yapılandırabiliriz (refactoring). Örneğin değişken ve metot isimlerini o anki düşündüklerimizi yansıtacak şekilde değiştirebiliriz. Uzun metotları da parçalara bölmeye çalışmak (extract method) kodun bizim açımızdan daha anlaşılır hale gelmesini kolaylaştıracaktır. Gönül rahatlığı ile kodu yeniden yapılandırabiliriz, çünkü testler bu süreci destekleyici nitelikte olacaktır. Bu şekilde yeniden yoğurmaya başladığımız kodu anlamamız daha kolaylaşacaktır. Kendi tarzımızı yansıtan kodu anlamak daha kolay olacaktır.

Aktif kod okuma seansları için testlerin olması bir gereklilik değildir. Kod üzerinde istediğimiz türde değişiklikleri yaparak, kodu adım, adım daha iyi anlamaya çalışabiliriz. Yeterince kod okuduğumuzu, değiştirdiğimizi ve anladığımızı düşündüğümüz noktada, edindiğimiz tecrübeler bizde kalacak şekilde kodu çöpe atabiliriz.

Kod ne yazık ki okunmak için yazılmış bir roman değildir. Ne kadar okunabilir kod yazmaya da gayret etsek, kodu konuşulan bir dilin kelimeleri kullanılarak oluşturulmuş bir roman gibi yazmamız ve okumamız mümkün olmayacaktır. Okuduğumuz kodu anlayabilmek için onu bakış açımızı yansıtacak şekilde değiştirmemiz zaruridir. Aktif kod okuma olarak isimlendirdiğim bu aktivite programcının kodu yeniden yapılandırma yetilerini geliştirici niteliktedir. Aktif kod okuma seanslarinda programcı hem kodu anlayarak hem de kodu yeniden yapilandirma tecrübesi edinerek, bir taşla iki kuş vurulmuş olur.

Çok Gezen mi Bilir, Çok Okuyan mı?

http://www.pratikprogramci.com/2014/09/11/cok-gezen-mi-bilir-cok-okuyan-mi/

Bu soru birçok tartışmaya yol açacak cinsten ve ilk bakışta cevabı göreceli gibi görünüyor. Vereceğim örnekte ise bu sorunun cevabı çok gezenin lehinde olacak.

Maaşlı çalışan bir programcının çalıştığı firma bünyesinde yaptıklarının haricinde öğrenebileceklerinin tümü üç yıl ve daha az bir zaman dilimine sığdırılabilir. Bu zaman diliminden sonra programcı bildiklerini tekrar eder ve yeni bir şeyler öğrenemez. Buradan şöyle bir sonuç çıkarabilir miyiz? **"Her programcı üç yılda bir iş yerini değiştirmelidir"**.

Eğer firmaya bir gün müdür olma planlarınız yoksa, her üç yılda bir iş değiştirmeniz programcı olarak sizin yararınıza olacaktır. Bilginin günümüzdeki yarı ömrünü göz önünde bulunduracak olursak, on sene aynı firmada çalışmış bir programcının sahip olduğu bilgi ve tecrübe, içinde çalıştığı piyasa için değerini kaybetmiştir. Bunu önlemenin en kolay yolu, sık aralıklarla iş değiştirmek ve at gözlüklerinden kurtulmaktır. Aynı şeyleri sürekli tekrar etmek, tecrübe kazanmak anlamına gelmez. Örneğin birinci sene öğrenileni sonraki dokuz sene tekrar etmek, bu konuda on sene tecrübe sahibi olmak değildir. Sahip olduğunuz tecrübeyi tekrar ederek yeni tecrübe sahibi olamazsınız, **yani tecrübe tecrübeyi doğurmaz**. Sadece yeni bir şeyler yaptığınızda, yeni tecrübe edinme şansını yakalayabilirsiniz.

Programcının daha iyi bir programcı olmasını sağlayan edindiği tecrübelerdir. İyi bir programcıyı tanımlamak için şu formülü kullanabiliriz: Bir programcının iyilik seviyesi edindiği tecrübelerle doğru orantılıdır. Burada tecrübe kelimesi ile programcının görüp, öğrendiği yeni metot, teknoloji ve bilgiyi kastediyorum.

Her üç senede bir iş değiştirme fikri size çok radikal gelebilir. Bu göründüğü kadar radikal bir durum değil. Yılda bir, altı ayda bir ya da ay başı iş değiştiren programcılar var. Serbest çalışan (freelancer; Burada freelancer terimini serbest programcılığın Avrupa'da icra ediliş şeklini yansıtacak şekilde kullandım. Türkiye'de freelancer terimi daha başka bir anlama sahip!) programcılar sürekli iş değiştirmeye mecburlar. Çok sık iş değiştirmek zorunda kaldıklarından, çok değişik türde tecrübeleri çok kısa bir zaman diliminde yapma fırsatı buluyorlar. Bu onların her yeni proje ile daha da değerli hale gelmelerini sağlıyor. Tercih edilmelerinin ana sebeplerinden birisi sahip oldukları tecrübeler.

Şimdi yazımın başlığına tekrar bir göz atalım. Yazım çok gezen mi bilir, çok okuyan mı? başlığını taşıyor. Serbest çalışan programcılar yanı sıra belli aralıklarla iş yerini değiştiren programcıların, yıllarca aynı şeyleri tekrar etmek zorunda kalan programcılara nazaran yaptıkları işten uzun yıllar ekmek yeme şansları çok daha yüksektir. Bunun nedenleri:

- Geldikleri her yeni ortamda olup, bitenleri kavrayabilmek için belli bir süre learning ve panic zone içinde kalırlar. Bu onların zamanla learning ve panic zone geçişine içgüdüsel karşı koyma bariyerlerini kırar ve çok daha kolay yeni şeyler öğrenmelerini sağlar.
- Sürekli yeni diller öğrenmek zorunda kalırlar. Daha iyi programcı olabilmenin başlıca şartlarından birisi çok değişik türde bilgisayar programlama dillerini tanımak ve kullanmaktır.

- Başka ekiplerde çalışma fırsatına sahip olmaları çiçekten, çiçeğe geçen arı misali birçok değişik şeyi görme ve adapte etmelerini kolaylaştırır. Bu onların zaman içinde basit çözüm oluşturma yetisini pekiştirir ve zamanla senior developer olmalarını sağlar.
- Sürekli değişik çalışma alanlarına ait (working domain) sorunları çözmek zorunda kalmaları, sorunları değişik perspektiflerden analiz etmelerini kolaylaştırır.
- Değişiklikten korkmazlar, çünkü onunla yaşamayı öğrenmişlerdir.
- Yeni şeyler öğrenmeye açıktırlar, çünkü bu şekilde iş gücü olarak daha da kıymetlendiklerini kavramışlardır.

Yazılımda iyi olmanın sırrı sürekli değişikliğe maruz kalmakta gizlidir. Bunun da yolu kanımca belli aralıklarla yeni projelere talip olmaktan geçiyor.

Yazılımda Geviş Getirme Taktiği Nasıl Uygulanır?

http://www.kurumsaljava.com/2014/09/05/yazilimda-gevis-getirme-taktigi-nasil-uygulanir/

Koyun, keçi, deve gibi hayvanlar otları çiğnemeden yutarlar. Daha sonra dinlenme esnasında yuttukları otları ağızlarına getirerek, çiğnerler. Buna işleme geviş getirme denir.

Geviş getirme hayvanların evrim sürecinde düşmanlarına karşı geliştirdikleri bir savunma mekanizmasıdır. Bu tür hayvanlar düşmanlarından kaçabilmek için buldukları besinleri çiğnemden yutarlar. Daha sonra kendilerini güvende hissettikleri bir yer ve anda çiğnemeden yuttukları bu besinleri geviş getirme yöntemiyle tekrar çiğnerler.

Bu yazımda geviş getirme mekanizmasının yazılımda nasıl kullanılabileceğini göstermek istiyorum.

Öncelikle yazdığımız kodların yazıcıdan çıktısını almamız gerekiyor. Daha sonra kağıtları küçük parçalara bölerek, çiğnemeden, yutuyoruz….. :)

İşin şakasını yaptıktan sonra, geviş taktiğine tekrar geri dönelim. Geviş getirme mekanizması iki kademeli işlemektedir. Birinci kademede hayvan çok hızlı bir şekilde otları dişleriyle koparark, çiğnemeden yutar. İkinci kademede hayvan geviş getirerek, daha önce çiğnemeye zaman bulamadığı besinleri çiğner.

Şimdi kod yazma sürecini de bu iki kademeyle şekillendirmeye çalışalım. Birinci kademenin geviş getirme mekanizmasına göre hızlı bir şekilde, kodun hangi yapıda olduğu düşünülmeden yazılması gerekiyor. Ben bu kademede test güdümlü yazılım yaparak, kodu oluşturmayı tercih ediyorum. Amacım çalışır durumda olan testleri ve kodları oluşturmak. Birinci kademede kodun hangi durumda olduğu beni ilgilendirmiyor. Ot yiyen hayvanın yaptığı gibi kodu yazıyor ve kaçıyorum. Kaçıyorum, çünkü ikinci kademede kodu tekrar çiğneyerek, yani kodu yeniden yapılandırarak (refactoring), tekrar elden geçireceğim. Kaçabilmem için testlerin ve kodun çalışır durumda olması gerekiyor.

Şimdi bunun nasıl yapıldığını bir örnek üzerinde inceleyelim. Bir dosyada yer alan tüm kelimeleri, kaç kez kullanıldıklarını görecek şekilde bir dosyaya eklememiz gerekiyor. Örneğin dosyamız içerisinde şu satır olsun:

```
bu bir test
```

Elde edeceğimiz dosyanın içeriği şu şekilde olmalı:

```
test:   1
bir:    1
bu:     1
```

Her kelimenin dosya içinde kaç kez kullanıldığını görüyoruz. Bu kodu test güdümlü şu şekilde yazdım:

```
package test.the.west;
```

```
import java.io.File;
import java.io.IOException;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;
import java.util.Set;

import org.apache.commons.io.FileUtils;
import org.hamcrest.CoreMatchers;
import org.junit.Test;

import static org.hamcrest.CoreMatchers.is;
import static org.junit.Assert.assertThat;

public class WordSorterTest {

    private class FileNotFound extends RuntimeException {

        private static final long serialVersionUID = 1L;

    }

    private class FileIsEmpty extends RuntimeException {

        private static final long serialVersionUID = 1L;

    }

    private class SortError extends RuntimeException {

        public SortError(final Exception e) {
            super(e);
        }

        private static final long serialVersionUID = 1L;

    }

    private class WordSorter {

        public void sort(final File file) {

            if (!file.exists()) {
                throw new FileNotFound();
            }

            List<String> lines = null;

            try {
                lines = FileUtils.readLines(file);

                if (lines.size() == 0) {
                    throw new FileIsEmpty();
                }
            } catch (final IOException e) {
```

```
                throw new SortError(e);
            }

            final Map<String, Integer> values = new HashMap<>();

            for (final String line : lines) {

                final String[] words = line.split(" ");

                for (final String word : words) {

                    Integer value = values.get(word);
                    if (value == null) {
                        values.put(word, new Integer(1));
                    } else {
                        values.put(word, ++value);
                    }
                }
            }

            final StringBuilder newContent = new StringBuilder();

            final Set<Entry<String, Integer>> entrySet =
                    values.entrySet();
            for (final Entry<String, Integer> entry : entrySet) {
                newContent.append(entry.getKey() + ":\t" +
                    entry.getValue()).append("\n");
            }

            System.out.println(newContent);

            try {
                FileUtils.write(file, newContent);
            } catch (final IOException e) {
                throw new SortError(e);
            }
        }
    }

    @Test(expected = FileNotFound.class)
    public void when_file_does_not_exits_error_is_thrown()
                throws Exception {

        final WordSorter sorter = new WordSorter();
        sorter.sort(new File("xx"));

    }

    @Test(expected = FileIsEmpty.class)
    public void when_file_is_emptry_error_is_thrown()
                throws Exception {

        final File file = File.createTempFile("myfile",
                ".txt");

        final WordSorter sorter = new WordSorter();
        sorter.sort(file);
```

```
    }

    @Test
    public void words_are_sorted_by_count_and_written_to_the_same_file()
             throws Exception {

        final File file = File.createTempFile("myfile", ".txt");

        FileUtils.write(file, getDummyContent());

        final WordSorter sorter = new WordSorter();
        sorter.sort(file);

        final List<String> lines = FileUtils.readLines(file);

        assertThat(lines.size(), is(3));
        assertThat(lines.get(0), is(CoreMatchers.equalTo("test:\t1")));
        assertThat(lines.get(1), is(CoreMatchers.equalTo("bir:\t1")));
        assertThat(lines.get(2), is(CoreMatchers.equalTo("bu:\t1")));

    }

    private String getDummyContent() {
        return "bu bir test";
    }
}
```

İşletme mantığı WordSorter sınıfının sort() isimli metodunda yer alıyor. İlk bakışta bu metot bünyesinde ne olup, bittiğini anlamak mümkün değil, çünkü kod testleri tatmin edecek seviyeden ileri gitmiyor, yani metot temiz kod (clean code) prensiplerine uygun değil. Tüm testlerimiz çalışır durumda ise, o zaman birinci kademeyi tamamlamış oluyoruz. Bu kademede gerisi bizi ilgilendirmiyor. Şimdi tekrar geviş getiren hayvanı hatırlayalım. O da ilk kademede dişleriyle otları koparıp, yuttuktan sonra geviş getirebileceği bir yere gider. Hayvan ilk kademede otun çiğnenmesi ile ilgilenmez. Biz de ilk kademede kodun çiğnenmesi, yani yeniden yapılandırılarak, daha okunur hale gelmesi ile ilgilenmiyoruz.

Şimdi geviş getirme zamanı. İkinci kademede kodu yeniden yapılandırarak (çiğneyerek), daha okunur hale getiriyoruz. Kodun yeni hali aşağıda yer almaktadır.

```
private class WordSorter {

    public void sort(final File file) {

        checkFile(file);

        final List<String> lines = getLines(file);

        final Map<String, Integer> sortedKeys = new HashMap<>();

        sortWords(lines, sortedKeys);

        final String newContent = buildSortedContent(sortedKeys);
```

```
        logSortedContent(newContent);

        writeSortedContentToFile(file, newContent);
    }

    private List<String> getLines(final File file) {
        List<String> lines = null;

        try {
            lines = FileUtils.readLines(file);
            checkLines(lines);
        } catch (final IOException e) {
            throw new SortError(e);
        }
        return lines;
    }

    private void checkLines(final List<String> lines) {
        if (noLines(lines)) {
            throw new FileIsEmpty();
        }
    }

    private void checkFile(final File file) {
        if (!file.exists()) {
            throw new FileNotFound();
        }
    }

    private void writeSortedContentToFile(final File file,
                final String newContent) {
        try {
            writeFile(file, newContent);
        } catch (final IOException e) {
            throw new SortError(e);
        }
    }

    private void writeFile(final File file, final String newContent)
            throws IOException {
        FileUtils.write(file, newContent);
    }

    private void logSortedContent(final String newContent) {
        System.out.println(newContent);
    }

    private String buildSortedContent(final Map<String, Integer>
                    values) {
        final StringBuilder content = new StringBuilder();

        final Set<Entry<String, Integer>> entrySet = values.entrySet();

        for (final Entry<String, Integer> entry : entrySet) {
            content.append(entry.getKey() + ":\t" +
                entry.getValue()).append("\n");
        }
```

```
        }
        return content.toString();
    }

    private void sortWords(final List<String> lines, final Map<String,
                Integer> values) {
        for (final String line : lines) {

            final String[] words = line.split(" ");
            handleWords(values, words);
        }
    }

    private void handleWords(final Map<String, Integer> values,
        final String[] words) {
        for (final String word : words) {

            handleWord(values, word);
        }
    }

    private void handleWord(final Map<String, Integer> values,
              final String word) {
        Integer value = values.get(word);
        if (valueNull(value)) {
            putWith(values, word, new Integer(1));
        } else {
            putWith(values, word, ++value);
        }
    }

    private boolean valueNull(final Integer value) {
        return value == null;
    }

    private void putWith(final Map<String, Integer> values,
            final String word, final Integer value) {
        values.put(word, value);
    }

    private boolean noLines(final List<String> lines) {
        return lines.size() == 0;
    }
}
```

Yeni yapılandırma kademesinde metotların 3-5 satırdan fazla olmamasına dikkat ediyorum. Bir metot bünyesinde ne kadar az satır varsa, kodu kavrama oranı o oranda artacaktır. sort() metodu şimdi okuyucusuna hangi işlemi gerçekleştirdiğini daha iyi anlatmaktadır.

Birinci kademede işletme mantığını çalışır hale getirmeye çalışırken, ikinci kademede kodu refactoring yöntemleriyle daha okunur hale getirmeye odaklanıyorum. Eğer birinci kademede kodu yazarken aynı zamanda okunur hale getirmeye çalışsaydım, o zaman tek sorumluluk prensibine ters düşmüş olurdum. Aynı zamanda sadece bir işlemle ilgilenmeliyim. Geviş getiren bir hayvan da geviş getirirken kalkıp, tekrar ot yemiyor. Hayvan geviş getirirken sadece bu işe

konsantre oluyor ve başka bir işle uğraşmıyor. Bu onun ikinci kademede yuttuğu otları verimli bir şekilde çiğnemesini sağlıyor ki bizde aynı şekilde her iki kademede yapılan işlemleri birbirlerinden ayırt ederek, belli safhalarda belli işleri yapmaya odaklanıyoruz. Bu bizim yaptığımız işte verimliliğimizi artırıcı bir durum.

Ben bu şekilde çalışmayı tercih ediyorum. İki kademeli çalışmak, çok hızlı bir şekilde işletme mantığını kodlamamı sağlıyor. İkinci kademede testlerin bana verdiği öz güvenle birlikte kodu istediğim şekilde yeniden yoğurabiliyor ve istediğim şekle sokabiliyorum. Buradan da anlaşılabileceği gibi yazılımda geviş getirme taktiğini uygulayabilmek için ikinci kademede yeniden yapılandırmak istediğim kodun tümünü kapsayan birim testlerine ihtiyaç duymaktayım. Eğer testlerim yoksa, ikinci kademede kodu yeniden yapılandırmaya cesaret etmem mümkün değil, çünkü elden yapılan yeniden yapılandırma işlemleri testler olmadan rus ruletinden farksızdır. Nasıl bir netice alacağınızdan emin olamazsınız ve yapılan değişikliklerin yan etkilerini ölçmeniz mümkün değildir. Bu yüzden geviş getirme taktiği test yazma zorunluluğunu beraberinde getirir.

Yazılımcılığın Ne Olduğunu Anlamamış Bilgisayar Mühendisi

http://www.kurumsaljava.com/2014/09/03/yazilimciligin-ne-oldugunu-anlamamis-bilgisayar-muhendisi/

Bu yazıyı okuduğumda, "yazılım kimlere kalmış" dedim ve bu yazıyı kaleme almaya karar verdim.

Şimdi bu yazıyı alıntılar vererek, analiz etmek istiyorum. Kendimi bunu yapmak zorunda hissediyorum, çünkü yazı tam bir kargaşa ve tutarsızlıklar abidesi.

Amacım kimseyi rencide etmek değil. Bu arkadaşımızın yazmış olduğu tezlerin mevcut ve müstakbel yazılımcılara ve yazılım sektörüne zarar verdiğini ya da verecegini düşünüyorum. Bu işi severek yapan birisi olarak, buna izin vermem düşünülemez. Bunun yanı sıra yazıyı okuyan birisi olarak, cevap verme ve yanlışlıkları düzeltme hakkımı kullanıyorum.

Yazının ana konusu mühendlislik okumadan, yazılımcı olunup, olunamayacağı şeklinde. Yazı sahibinin görüşü bu işi sadece bilgisayar ya da yazılım mühendisi olanların yapabileceği yönünde ve şöyle bir çelişki ihtiva ediyor:

```
Yazılım üretmek bir sanattır, aksi iddia edilemez. Sizin bu sanata
yatkın olup olmadığınızı ortaya çıkartacak çok kolay bir testi
aşağıda açıkladım.
```

Yazar ne yazık ki burada sanatı kastetmekle birlikte sanat ile zanaati karıştırmış durumda. Kurmuş olduğu cümle çelişkili, çünkü savunduğu mühendislik olgusu ile sanatın yakından ve uzaktan hiçbir ilgisi olamaz. Yazılım mühendislik işidir kanısını savunurken, yazılım sanattır diyor. Programcılık sanat mı, zanaat mı? başlıklı yazımda bu konuya değindim. Programcılık sanat değildir. Programcılığın bir kısmı mühendislik bir kısmı da zanaattir. Uygulamalar bir mühendis kafasıyla tasarlanır, bir zanaat ustası gibi koda dökülür.

Yazar bir yazılımcı olup, olamayacağımızı anlayabilmemiz için bir test hazırlamış. Testi inceleyelim:

```
Tek yapmanız gereken başlamak istediğiniz programlama veya uygulama
dilinin kitabını alıp okumaya ve uygulamaya başlamak. Şayet okuduğunuzu
anlayabiliyor ve uygulayabiliyorsanız, gayet güzel! Şayet okuduğunuzu
anlayamıyor, küçük uygulamalar geliştirirken dahi kitabı açıp bakma
ihtiyacı hissediyorsanız, bu durumda sizden yazılımcı olmaz.
```

Programcı olmaya hevesli ama bu konuda hiçbir bilgisi olmayan bir şahsın bu testi geçmesi imkansız, çünkü test çok absürd. Yeni bir alana ayak bastığınızda, o alana has dili bilmediğiniz için başlangıçta bocalamanız çok doğal. Her çalışma sahasının kendine has zorlukları olur ve bu zorluklar çalışma sahasına aşina olundukça aşılır. Eğer herkes yeni bir sahaya adım atarken böyle bir test yapmak zorunda kalsaydı, o zaman bu dünyada kalifiye bir eleman bulmak mümkün olmazdı.

İşte çok cesur ortaya atılmış bir tez daha...

```
Mühendis ile yazılımcı arasındaki fark, maaş konusunda keskinleşir.
```

Yazar burada ne yazık ki genel değil, yerel bir bakış sergilemektedir. Türkiye çok genç bir nüfusa sahip olan bir ülke. Bunun yanı sıra yazılım sektörü Türkiyemizde çocuk ayakkabılarıyla geziniyor. Yazılımın gerçek anlamda nasıl yapılması gerektiği konusunda bilgili insan sayısı parmakla sayılacak kadar az. Çoğu bilgisayar mühendisi de bu işi bilmiyor, sonradan bu işi öğrenmiş şahıslar da.

Bilgisayar ya da yazılım mühendisi ile alaylı yazılımcı arasındaki fark kesinlikle maaş olamaz. Türkiye küçük bir piyasa olduğundan ve çok sayıda yazılımcı bu piyasada iş bulmaya çalıştığı için firmalar seçici olma ayrıcalığına sahipler. Doğal olarak arz edilen iş gücünün talep edilenden fazla olması, firmaların mühendis sıfatını taşıyan yazılımcıları tercih etmesini kolaylaştırıyor. Almanya gibi bir piyasada Türkiye'ye nazaran arz talep dengesi tam tersine olduğundan, mühendis ile alaylı yazılımcı arasında hiçbir fark yok. Fark olsa bile bu farklılık tecrübe bazındadır. Kim daha tecrübeli ise, daha çok maaş alır. Hesap bu kadar kolay. Mühendis, mühendis olduğu için daha çok para kazanır ya da kazanmalıdır demek, abesle iştigalden başka bir şey olamaz. Ayrıca bu bazı kafaların nasıl çalıştığının da çok iyi bir ispatıdır. Bir iş gücünü kıymetli kılan, iş konusunda ne kadar kalifiye olduğudur. Üniversite bitirerek hiçbir kimse kalifiye elemanım diye ortalıkta gezinemez. İnsan iş başında, öğrendiklerini uygulayarak pekişir ve kalifiye bir eleman haline gelir. Bu açıdan bakıldığında bir bilgisayar mühendisi ile kendini çok iyi yetiştirmiş bir alaylı yazılımcı arasında hiçbir fark yoktur. Nice fizik ve kimya mühendisleri tanıyorum. Yıllarca programcı olarak çalışmışlar ve benim diyen birçok bilgisayar ya da yazılım mühendisini ceplerinden çıkartırlar.

Yazarın bir sonraki tezine bir göz atalım:

```
Mühendisler, kurumsal yapılı şirketlerde daima yazılımcıların
patronu olarak çalışır (başlangıçta böyle olmayabilir,
ama yazılımcının 5 yıldır beklediği terfiyi mühendis en çok
2 yılda alır.
```

Ben yazılımcı olsam ve bu şekilde hakkım yense, böyle bir firmada beş dakika bile durmazdım. Çalışanlarına sarf ettikleri emeğe göre değil de, sahip oldukları bir takim mühendislik sıfatlarına göre öncelik tanıyan firmalar, bu hallerini piyasada eleman bulamayacak hale geldiklerinde çok çabuk değiştirirler. Yazarın tezinden de anlaşıldığı gibi Türkiye'de yazılım sektörü yok kadar küçüktür ve bu durumu kendi avantajları yönünde kullanan kurumsal firmalar, şimdilik alaylı programcıların haklarını yemektedirler. Yarın, bir gün piyasa büyüdüğünde ve programcılara olan talep arttığında, kim daha çabuk terfi alıyormuş, göreceğiz. Kim daha çok çalışıyor ve tecrübe sahibi ise, terfiyi o almalı. Mühendisi alaylının önünde gören zihniyet, iş gücüne ihtiyaç duyduğunda, o alaylının önünde eğilir. Bu kaypaklık zaten mühendisler önceliklidir zihniyetinin ne kadar kokuşmuş olduğunun ve dikkate ve ciddiye alınmaması gerektiğinin ibaresidir.

Devam edelim:

```
Bazen kurumsal şirketler de "uzman" alabilir, ama bu durumda
"mühendisin" yanında takılan bir eleman gibi olursunuz, tavsiye etmem.
```

Yazar ne yazık ki mühendis dediği şahısları cebinden çıkaran alaylılarla çalışma fırsatı bulamamış. Alaylı konumu itibari ile zaten mühendisin bildiğinin iki katını, çalıştığının iki katını, öğrendiğinin iki katını öğrenme eğilimi gösterir, çünkü kendisini mühendis ünvanı olmadan ispatlamak zorundadır. Bu şartlar altında boynuz kulağı çok çabuk geçer.

```
İngilizce olmadan sıradan "programcı" olmanın ötesine geçemezsiniz.
```

Yazara bu konuda haklıdır. Türkçe kaynakların artması ile ingilizce bilmenin de önemini yitireceğini düşünmekle birlikte, ingilizce bilmenin yazılımda şimdilik bir gereklilik olduğunu düşünüyorum.

```
Üniversite okumadan, "yazılım uzmanı" olmaktan öteye geçemezsiniz...
Bu işin en doğru yolu mühendis olmaktır.
```

Eğer kendim de bir bilgisayar mühendisliği okumuş bir yazılımcı olmasaydım, bu üniversitede ne öğretildiğini gerçekten çok merak ederdim. Ben size üniversitede ne öğretildiğini kısaca özetleyeyim: biraz matematik, biraz algoritma, biraz ondan, biraz bundan ve dört sene sonra diplomasını alıp, mühendis oldum diye böbürlenen, ama gerçek yazılımın nasıl yapıldığından ışık yılları uzaklıkta şahıslar. Bu mühendislik olayını büyütmenin gerçekten bir anlamı yok. Makine ve elektronik mühendisleri bile biz bilgisayar mühendislerinden mezun olduklarında çok daha donanımlı mühendisler. Bunun yanı sıra en doğru yolu kavramı göreceli bir kavramdır. Yolların hepsi Roma'ya çıkar diye bir söz vardır. İyi bir yazılımcı olmak için bilgisayar ya da yazılım mühendisi olma zorunluluğu var diye bir şey olamaz.

```
Şu anda bir işveren olarak da "yazılım uzmanı" hiçbir arkadaşımızı
işe almadım, hep "yazılım veya bilgisayar mühendisi" işe aldım.
Çünkü bizim ürettiğimiz yazılımlarda, analiz ve matematiksel
yetenekler gerekiyor.
```

Eğer uzay mekiği ya da bilgisayar oyunları için program yazmıyorsanız, yazdığınız kodun %99'unda toplama, çıkarma gibi basit matematiksel işlemler yaparsınız. En son yaptığım matematiksek işlem x++ idi.

Programcılığın matematikle çok ilgisi yok. Soyutlayabilme yeteneğiniz varsa, programcı olmak için en önemli yeteneğe sahipsiniz demektir. Geriye kalan her şeyi kitaplardan ve bu işin ustalarına bakarak, öğrenebilirsiniz.

```
Bu işi kendi kendine veya kursta öğrenen hiç kimse de kalkıp,
"Yahu ben cebir, analiz, algoritma vb. de öğreneyim" demiyor.
```

Mühendislerin çoğu ilerde mühendis ünvanım ile daha iyi bir yere gelir, daha fazla para kazanırım diye mühendis olurlar. Alaylı programcıların çoğu bu işe gerçekten gönül verdikleri için programcı olurlar. Bu işe gönül veren bir insan Java gibi mainstream olan diller haricindeki dilleri de öğrenir, çünkü kendini geliştirme yolunda her türlü bilgiyi sünger gibi emer. Kim Daha

İyi Programcı? başlıklı yazımda bu konuya değindim. Verdiğim eğitimlerde bazı alaylı olan katılımcıların mühendislere nazaran daha geniş bir teknik altyapıya sahip olduklarına, kendi uzmanlık alanları haricindeki alanlarla da ilgilendiklerine ve yenilikleri yakından takip ettiklerine şahit oldum. Bunun sebebi bu işe gerçekten gönül vermiş olmaları ve severek yapıyor olmalarıdır. Ben bu tür adamları işe alırdım, çünkü projeyi uçuracak olanlar bu tür yazılımcılardır. Eğer bir yazılım mühendisi de bu tür özelliklere sahip ise, o zaman onun da başımızın üzerinde yeri var.

Hindistan'da insanları sınıflandırmak ve toplumu ayrıştırmak için kutu sistemi oluşturulmuş. Bir kutuya ait insanların başka bir kutuya geçmeleri mümkün değil. Yazar da yazılımcılar için böyle bir kutu sistemi oluşturmaya çalışmaktadır. Bunun kimseye bir faydası olmadığı ortadadır, çünkü oluşturulan bu kutu sistemi maddi olarak daha fazlasını hak eden alaylıların kötü şartlarda sömürülmelerine sebep vermektedir.

Bir işin üniversite eğitimini almış olmak şüphesiz iyi bir teknik altyapının oluşturulmasında faydalıdır. Lakin üniversitelerin ne kadar kolay ve uyduruk bir şekilde bitirilebileceğini hepimiz biliyoruz. Hele hele Türkiye gibi ezberciliğe dayanan bir eğitim sisteminin olduğu bir ülkede alınan üniversite eğitimini bir ayrıcalık olarak satmaya kalkmak, bu işin hakkını vererek çalışmaya çalışanlara yapılmış büyük haksızlıktır. Gerçek öğrenim süreci üniversite bittikten sonra başlar. Alaylı olsun, mühendis olsun, iyi bir programcı ile vasat bir programcının arasındaki farklılık da burada belirir. Kim her daim yeni bir şeyler öğrenme eğilimi gösterirse, öne geçer. Yazılım tutku işidir. O tutkuya sahip olmayanı mühendis ünvanı bile kurtaramaz. Benlik gütmeyelim.

Yazar bu yazıma şu yorumu bırakmıştır. Yazar 5.9.2015 tarihinde benim yazıma cevaben şu yazıyı kaleme almıştır.

Birim Testlerinde Beklentilerimi Daha Net Nasıl İfade Edebilirim?

http://www.pratikprogramci.com/2014/06/05/birim-testlerinde-beklentilerimi-daha-net-nasil-ifade-edebilirim/

Kodkata.com bünyesinde hazırladığım Koşullu Mantığın Komut İle Değiştirilmesi isimli katada aşağıda yer alan birim testlerinden yola çıkılarak, uygulama yeniden yapılandırılıyor.

```
package com.kodkata.kata.replaceConditionalDispatcherWithCommand.orig;
import junit.framework.Assert;
import org.junit.Test;
public class PriceCalculatorTest {

    private static final String LOCALE_TURKISH_REPUBLIC = "tr_TR";
    private static final int DEFAULT_NETTO_PRICE = 100;
    private static final String LOCALE_GERMANY = "de_DE";
    private static final String LOCALE_AUSTRIA = "at_AT";

    @Test
    public void shouldCalculatePriceForGermanLocale()
                throws Exception {
        Assert.assertTrue(new Order().calculatePrice(
                LOCALE_GERMANY, DEFAULT_NETTO_PRICE) == 119);
    }

    @Test
    public void shouldCalculatePriceForTurkishLocale()
                throws Exception {
        Assert.assertTrue(new Order().calculatePrice(
            LOCALE_TURKISH_REPUBLIC, DEFAULT_NETTO_PRICE) == 118);
    }

    @Test
    public void shouldCalculatePriceForAustrianLocale()
        throws Exception {
        Assert.assertTrue(new Order().calculatePrice(
            LOCALE_AUSTRIA, DEFAULT_NETTO_PRICE) == 120);
    }
}
```

Uygulamadan olan beklentilerimi ifade etmek için assertTrue() metodunu kullandım. İlk bakışta bu beklentilerin neyi ifade ettiğini anlamak kolay değil. Assertj çatısını kullanarak, DSL (Domain Specific Language) bazlı bir assert oluşturabiliriz. Bu DSL bazlı assert sınıfını kullandığımız taktirde testleri aşağıdaki şekilde yapılandırmak mümkün olacaktır.

```
package com.kodkata.kata.replaceConditionalDispatcherWithCommand.orig;
import static com.kodkata.kata.replaceConditionalDispatcherWithCommand.
            orig.OrderAssert.assertThat;
import org.junit.Test;

public class PriceCalculatorTest {

    private static final String LOCALE_TURKISH_REPUBLIC = "tr_TR";
    private static final int DEFAULT_NETTO_PRICE = 100;
    private static final String LOCALE_GERMANY = "de_DE";
    private static final String LOCALE_AUSTRIA = "at_AT";
    private final Order order = new Order();

    @Test
    public void shouldCalculatePriceForGermanLocale() throws Exception {

        assertThat(order)
            .inCountry(LOCALE_GERMANY)
            .withPrice(DEFAULT_NETTO_PRICE)
            .hasTotalPrice(119);
    }

    @Test
    public void shouldCalculatePriceForTurkishLocale() throws Exception {

        assertThat(order)
            .inCountry(LOCALE_TURKISH_REPUBLIC)
            .withPrice(DEFAULT_NETTO_PRICE)
            .hasTotalPrice(118);
    }

    @Test
    public void shouldCalculatePriceForAustrianLocale() throws Exception {

        assertThat(order)
            .inCountry(LOCALE_AUSTRIA)
            .withPrice(DEFAULT_NETTO_PRICE)
            .hasTotalPrice(120);
    }
}
```

Şimdi ilk bakışta testlerin daha kolay okunur yapıda olduğunu düşünüyorum. Oluşturduğum OrderAssert sınıfı şu yapıda:

```
package com.kodkata.kata.replaceConditionalDispatcherWithCommand.orig;
import org.assertj.core.api.AbstractAssert;
import org.assertj.core.api.Assertions;

public class OrderAssert extends AbstractAssert<OrderAssert, Order> {

    private String locale;
    private int price;

    private OrderAssert(final Order actual) {
        super(actual, OrderAssert.class);
    }

    public static OrderAssert assertThat(final Order actual) {
        return new OrderAssert(actual);
    }

    public OrderAssert hasTotalPrice(final int total) {
        isNotNull();

        Assertions.assertThat(actual.calculatePrice(locale, price)).isEqualTo(total);
        return this;
    }

    public OrderAssert inCountry(final String locale) {
        this.locale = locale;
        return this;
    }

    public OrderAssert withPrice(final int price) {
        this.price = price;
        return this;
    }
}
```

Başlama ve Bitirme Kriterleri

http://www.pratikprogramci.com/2014/08/08/baslama-ve-bitirme-kriterleri/

Bir yolculuğa çıkılmadan önce gerekli hazırlıklar yapılır. Bu hazırlıklar yolculuğa çıkabilmek için atılması gereken zorunlu adımlardır. Bu adımlar atılmadığı taktirde, yolculuğun başarısız olması, yolculuğun iptali ya da planlanan zamanda tamamlanamama rizikosu doğabilir.

Yolculuğa çıkıldıktan sonra nihai amaç varış noktasına planlanan zamanda erişmektir. Varış noktasına planlanan zaman diliminde erişildiğinde, yolculuk tamamlanmış olarak kabul edilir.

Bir yolculuğun başlaması ve bitişi hakkında özetle şunları söyleyebiliriz:

- Yolculuğa çıkabilmek için bazı koşulların yerine gelmesi gerekmektedir. Bu koşullara başlama kriterleri ismini verebiliriz.
- Yolculuğu tamamlanmış olarak kabul edebilmek için bazı koşulların yerine gelmesi gerekmektedir. Bu koşullara bitirme kriterleri ismini verebiliriz.

Bir yolculuk için tanımladığımız başlama ve bitirme kriterlerini yazılıma aktarabiliriz. Yazılımda yolculuk bir kullanıcı senaryosunu koda dökmekle başlar. Bu senaryo kodlandığında yolculuk tamamlanmış olur. Ama ne zaman bir kullanıcı senaryosunun kodlanmak için hazır olduğunu anlayabiliriz? Yani ne zaman yola çıkmak için hazırız? Aynı soruları kullanıcı senaryosunu tamamladığımızda da sorabiliriz: Ne zaman bir kullanıcı senaryosunu gerektiği şekilde koda döktüğümüzden emin olabiliriz? Yani ne zaman yolculuğumuzu tamamladık?

Çevik süreçlerde yolculuğa çıkabilmek ve yolculuğun tamamlandı olarak görülebilmesini sağlamak için başlama ve bitirme kriter tanımlamaları kullanılır. Yolculuğa çıkabilmek için kullanılan kriter tanımlamasına Definition Of Ready (DOR), yani hazır kriterleri tanımlaması, yolculuğu tamamlandı olarak adlandırmak için Definition Of Done (DOD), yani tamamlandı kriterleri tanımlaması oluşturulur. Şimdi DOR un ve DOD un ne olduklarını ve ne amaçla kullanıldıklarını yakından inceleyelim.

Hazır Kriterlerinin Tanımlanması (DOR)

Müşterimiz aşağıda yer alan kullanıcı senaryosunun tarafımızdan implemente edilmesini istemektedir:

```
"Kullanıcı e-posta adresini ve şifresini kullanarak, login işlemini gerçekleştirir."
```

Hangi şartlar altında bu kullanıcı senaryosunu koda dökmeye başlayabiliriz? Bu yolculuğa çıkabilmek için ihtiyaç duyduğumuz her şeye sahip miyiz? Her şeye sahip olduğumuzu nasıl kontrol edebiliriz?

Yazılım deterministik sonuçlar doğurmakla yükümlüdür. Bunlar müşterinin uygulamadan olan beklentileridir. Bu yüzden bir kullanıcı senaryosunun yazılımcı olarak sorumluluğunu üstlenerek, yola çıkmadan önce objektif bir şekilde yolculuğa hazır olup, olmadığımızı kestirebilmemiz gerekmektedir. Subjektif algılara dayalı bir analiz, yolculuğumuzun başarısız sonuç vermesine sebep olabilir. Sonuç itibari ile bir yerden başka bir yere giderken, kendimizi

körü, körüne yola vermiyor ve yolculuğa çıkabilmek için gerekli şartları göz önünde bulunduruyoruz. Aynı şey bir kullanıcı senaryosu için yola çıkarken de geçerlidir.

DOR bir listedir ve kullanıcı senaryolarının implementasyon için hazır kriterlerini ihtiva eder. Bir kullanıcı senaryosu sadece DOR uyumlu ise, implemente edilebilir konumundadır. DOR uyumlu olan bir kullanıcı senaryosu yolculuk için tüm şartların yerine geldiği ve yolculuğun başlayabileceği anlamına gelmektedir.

DOR uygulama bünyesinde yer alacak tüm kullanıcı senaryoları için geçerli bir tanımlamadır ve şu kriterleri ihtiva edebilir:

- Kullanıcı senaryosunun öncelik sırası ve önemliliği müşteri tarafından tanımlandı.
- Müşteri kullanıcı hikayesi hakkında detaylı bilgiyi ekip ile paylaştı.
- Kullanıcı senaryosu için gerekli implementasyon süresi tüm ekip tarafından tahmin edildi.
- Müşteri tarafından onay/kabul (acceptance) kriterleri tanımlandı.

Şimdi tekrar üzerinde çalıştığımız kullanıcı senaryosunu hatırlayalım:

```
"Kullanıcı e-posta adresini ve şifresini kullanarak, login işlemini gerçekleştirir."
```

Bu kullanıcı senaryosunu alıp, yola çıkabilmek için DOR listesinde yer alan kriterlerin yerine gelmiş olması gerekir. Öncelikle müşterinin bu kullanıcı senaryosuna ihtiyacı olduğundan emin olmamız gerekmektedir. Müşteri için ilk etapta önemli olmayan kullanıcı senaryolarının implemente edilmeleri, zaman ve kaynak kaybıdır. Zaman içinde müşteri gereksinimleri değişebileceğinden, hangi öneme sahip olduğunu bilmediğimiz bir kullanıcı senaryosu üzerinde çalışmanın nasıl bir netice doğuracağı malum. Bu yüzden DOR kriterlerinden ilki, müşteri tarafından kullanıcı senaryosuna öncelik sırası verilmesi gerektiğini söylemektedir. Müşteri oluşturduğu kullanıcı senaryoları için öncelik sırası vermekle yükümlüdür. Değişik planlama toplantılarında müşteri bu bilgiyi programcılarla paylaşır. Bu şekilde programcılar kullanıcı senaryolarının hangi önceliğe göre implemente etmeleri gerektiğini bilirler.

Müşteri tarafından oluşturulan kullanıcı senaryosu bir cümleden oluşmaktadır. Müşterinin ne istediğinin tam olarak anlaşılabilmesi için müşterinin ne istediğini ekiple paylaşması gerekir. Eğer ekip içinde kullanıcı senaryosu hakkında yeterinde bilgi mevcut değilse, kullanıcı senaryosu hazır kriterlerinden birisini ihlal ettiğinden dolayı implemente edilemez. Müşteri kullanıcı senaryoları hakkında detaylı bilgiyi ekip ile paylaşmakla yükümlüdür.

Diğer bir hazır kriteri, kullanıcı senaryosu için gerekli implementasyon süresinin tahmin edilmiş olmasıdır. Süre tahmini planlama toplantılarında ekip tarafından yapılır. Bu özelliğe sahip olmayan kullanıcı senaryoları implementasyon için hazır değildir. İterasyon bazlı çalışan ekipler, iterasyon sonunda müşteri için kullanılabilir bir uygulama sürümü oluşturmakla yükümlüdür. İterasyonlar belli bir zaman dilimini kapsar. İmplementasyon süresi tanımlı olmayan bir kullanıcı senaryosu, iterasyon gidişatını olumsuz etkileyebilir. En kötü ihtimalle kullanıcı senaryosu tamamlanamadığı için sürüm oluşturulamaz.

Onay/kabul kriterleri müşteri tarafından tanımlanan ve müşteri ve programcı arasında yapılacak iş hakkında detayları ihtiva eden bir sözleşmedir. Bu yazımda onay/kabul kriterlerinin

nasıl olmaları gerektiği konusuna değindim. Onay/kabul kriterleri müşterinin uygulamadan olan beklentilerini ifade ederler. Her kullanıcı senaryosunun tanımlı onay/kabul kriter listesi olmalıdır. Aksi taktirde müşterinin ne istediğinden emin olmamız mümkün olmayabilir. Programcı olarak onay/kabul kriterlerinden yola çıkarak, onay/kabul testleri oluşturabiliriz. Tanımlı olan tüm onay/kabul kriterlerini yansıtan ve çalışır durumdaki onay/kabul testleri, programcı olarak müşterinin beklentilerini tatmin eden bir kullanıcı senaryosu implementasyonu yaptığımız anlamına gelir. Onay/kabul kriterleri tanımlı olmayan bir kullanıcı senaryosu üzerinde çalışmak, senaryoyu tamamladığımızı düşünsek bile, müşterinin "istediğim şey bu değildi" söylemiyle karşı karşıya kalma rizikosunu artırır. Oysaki müşteri tarafından tanımlanmış olan onay/kabul kriterleri doğrultusunda yaptığımız çalışma, müşterinin yaptığımız çalışmaya itiraz etme ihtimalini düşürür, çünkü oluşturduğumuz ve çalışır durumda olan onay/kabul testleri kendisi tarafından tanımlanan onay/kabul kriterlerini baz almaktadır. Bu "programcı olarak müşteri ne istiyorsa, onu implemente ettim" anlamına gelmektedir.

DOR genel anlamda müşteri ile ekip arasında iş başlangıcını tanımlayan bir sözleşme metnidir. Bu sözleşmede yer alan kriterler yerine gelmediği sürece, programcı olarak kullanıcı senaryoları için sorumluluk altına girmeme hakkımız bulunuyor. Eğer DOR olmadan işe başlıyorsak, hem programcı olarak biz tutamayacağımız bir söz vermiş olmaktayız, hem de müşteri gereksinimlerini anlaşılır bir şekilde ifade etme zahmetine katlanmamıştır.

Birçok projenin en zorlandığı alanlardan birisi, yapılan planlamaya uygun bir şekilde sürümlerin oluşturulamamasıdır. Bunun başlıca sebeplerinden birisi DOR un eksikliğidir. DOR un eksikliği sürüm planlamasını zora sokar ve ekip içindeki programcıların deterministik netice oluşturmalarını engeller. Sürüm planında yer alan her kullanıcı senaryosunun (user story) DOR uyumlu olması demek, test edilebilir, açık ve net anlaşılır ve ölçülebilir yapıda olması demektir.

Bitirme Kriterlerinin Tanımlanması (DOD)

DOR yola ne zaman çıkabileceğimizi söyleyen doküman ise, DOD hedefe ne zaman vardığımızı anlamamızı sağlayan dokumandır. DOD projenin sağlıklı ilerlemesi için DOR kadar önemlidir. DOD bünyesinde bir kullanıcı senaryosunun tamamlanış kriterleri yer alır. Örnek bir DOD şu şekilde tanımlanabilir:

- Müşteri tarafından tanımlanan onay/kabul kriterleri için onay/kabul testleri oluşturuldu ve hepsi çalışır durumda.
- Kod hatasız derleniyor ve tüm testler çalışır durumda.
- Ekip içinden bir programcı ile kod inceleme (code review) seansı yapıldı. Bunun yanı sıra Sonar gibi araclar yardımı ile kodun statik analizi ve gerekli düzeltmeler yapıldı.
- Gerekli dokümentasyon oluşturuldu.
- Kullanıcı senaryosunu ihtiva eden kod versiyon kontrol sunucusuna eklendi.
- Kullanıcı senaryosunu ihtiva eden kod sürekli entegrasyon sunucusu (Jenkins) tarafından uygulamanın diğer bölümleri ile entegre edildi. Tüm testler çalışır durumda.

Çalıştığım bir projede kullanıcı senaryolarını mümkün olan en iyi standartlarda implemente ettiğime dair imza atmam istenmişti. İmza atmam, ama isterseniz yemin edeyim demiştim :) Şaka bir yana bunun gibi durumlar DOD eksikliği ve programcıya olan güvensizliğin ibareleridir. DOD oluşturulduğu taktirde, programcı yolculuğunu tamamlamak için hangi

adımları atması gerektiğini bilir ve çalışmalarını bu yönde sürdürür. DOD yolculuğun nerede son bulması gerektiğinin habercisidir.

DOD bünyesinde olması gereken en önemli kriter, onay/kabul kriterlerinin yerine getirildiğinin ispatını teşkil eden onay/kabul testlerinin varlığının mecburiyetidir. Aksi taktirde kullanıcı senaryosundan sorumlu olan programcı müşteri tarafından ifade edilen gereksinimleri tatmin eden kodu yazdığından emin olamaz. Bu durumda programcı müşterinin her türlü itirazına maruz kalabilir. Bunun önüne geçmenin ve müşterinin ifade ettiği gereksinimleri tatmin eden kodu oluşturmanın en sağlıklı yöntemi, onay/kabul kriterlerini baz alan onay/kabul testleri oluşturmaktır. Onay/kabul testleri olmayan bir kullanıcı senaryosunun tamamlandı olarak görülmesi imkansızdır.

Diğer önemli bir DOD kriteri de, kullanıcı senaryosunu geliştiren programcının yaptığı çalışmaları bir ekip arkadaşıyla paylaşmasıdır. Bu eşli programlama (pair programming) yöntemi ya da implementasyon sonunda kod inceleme seansı (code review) bünyesinde gerçekleşebilir. Bu şekilde programcının oluşturduğu çözüm ekip içinde paylaşılır ve varsa mevcut hataların giderilmesini sağlanır.

DOD un eksikliği müşteri gereksinimlerini tatmin etmeyen ve hata ihtiva eden kod birimlerinin oluşmasını hızlandırır.

DOR ve DOD statik dokümanlar değildirler. Zaman içinde ekibin ihtiyaçları doğrultusunda değişikliğe uğrayabilirler. Onay/kabul kriterlerinin programcılar tarafından tanımlandığı projeler de gördüm, müşteriler tarafından tanımlandığı projeler de. Gün sonunda kimin, neyi tanımladığı değil, çalışır ve müşteri gereksinimlerini tatmin eden bir uygulamanın oluşması önemlidir. DOR ve DOD da bu amaca hizmet eden araçlardır. Bu şekilde algılanmaları ve uygulanmaları projelerin gidişatlarını olumlu yönde etkilemektedir.

500 Beygir Gücünün Hazin Sonu

http://www.kurumsaljava.com/2014/03/08/500-beygir-gucunun-hazin-sonu/

İdeal şartlar altında bir programcının savaş verdiği tek bir cephe vardır, o da müşteri gereksinimlerini önemlilik sırasına göre kodlamak.

Çevik süreçlerde müşteriye 2-4 hafta süren çalışmalar ardından çalışır bir uygulama prototipi sunulur. Bu prototip müşteriye uygulamanın hangi seviyeye geldiğini, isteklerinin doğru uygulanıp, uygulanmadığını ve hangi değişikliklerin gerekli olduğunu anlama fırsatı verir. Buradan change request olarak bilinen ve müşteri gereksinimlerine daha yerinde cevap verebilmek için atılması gereken adımları tanımlayan değişiklikler doğabilir. Bu değişiklikler bir sonraki 2-4 haftalık çalışma sürecinde kullanıcı hikayesi (user story) olarak programcıya yansır. Bu değişikliklere rağmen programcının savaşı hala bir cephede devam etmektedir.

Eğer proje bünyesinde kodun kalitesine dikkat edilmiyorsa, programcı için ikinci bir savaş cephesi daha açılır: oluşan hataları gidermeye çalışma cephesi. Bu çok kısır bir döngünün oluşmasına sebep verebilir, çünkü müşteriye verilen prototiplerde hata sayısı çok yüksek olduğunda, programcı zamanının büyük bir bölümünü açılan ikinci cephede savaşarak harcar. İkinci cephe birim ve entegrasyon testleri yazılarak kapatılabilir ya da açılması engellenebilir.

Buraya kadar tanımladığım yazılım geliştirme süreci başın, ayakların ve kolların nerede olduğunu belli olan bir süreçtir. Ekip neyin, ne zaman yapılması gerektiğini bilir ve çalışmalarını ona göre yönlendirir. Başın, ayakların ve kolların nerede olduğu bilinmeyen ve en kötü ihtimalle bu uzuvların yerlerinin devamlı değiştiği projelere de rastlamak mümkündür. Bu tür projelerde programcı birçok cephede savaş verir. Programcının kendisini en çaresiz hissettiği ve cephanesinin ve motivasyonunun en çabuk azaldığı projeler bu tür projelerdir.

Bu tür projelerde change request çok başka bir kimliğe bürünerek programcılara musallat olur: acil iş!

Proje yönetimi acil iş olarak gelen talepleri hemen ekibe delege ederek, sonuç almak ister. Hadi gelin birlikte acil iş olarak gelen bir işin programcı için nelere sebep verdiğine bir göz atalım.

Acil işi öncelikle olağan üstü hal olarak tanımlayabiliriz. Normal proje akışı yanı sıra programcı bu olağan üstü hali normale dönüştürebilmek için birçok adım atmak zorundadır. Genel olarak acil değişiklik isteklerinin trunk olarak isimlendirilen ana kod dalında yapılması mümkün değildir. Bu yüzden programcı yan dal, yani branch oluşturmak zorundadır. Acil iş için gerekli kod değişikliği bu yan kod dalında yapılır ve değişiklikler daha sonra ana kod dalı ile birleştirilir. Bu işleme merge ismi verilmektedir. Her geçen gün ile merge işlemi daha komplike bir hal alır. İki hafta branch üzerinde çalıştıktan sonra muazzam emek ve dikkat sarf etmeden kodun ana dal ile birleştirilebildiğini hiçbir projede görmedim. Branch ve merge işlemleri başlı başına ağır işler olup, programcıyı çok yorarlar.

Bunun yanı sıra yapılan değişikliklerin staging olarak isimlendirilen değişik türdeki test sunucularında test edilmesi gerekmektedir. Bu da tüm ekibi rehin alabilecek türde bir işlemdir. Test sürecinden sonra yeni bir sürüm oluşturulup, bu sürümün müşteri için çalışır hale

getirilmesi gerekir.

Şimdi çok acil iş, çok acil iş diye devamlı ekibin çalışma sepetine dökülen bu incilerin, programcıları ruhen ve bedenen ne kadar çok tıkadıklarını anlayabiliyoruz sanırım.

Ben programcıları 500 beygir gücünde yarış arabaları ile kıyaslıyorum. Verin kullanıcı hikayelerini programcının eline ve bir zamanlar Formel 1 yarışlarının olduğu İstanbul Parka salın. Nasıl son speed turladığını göreceksiniz.

Temcit pilavı gibi programcının devamlı acil iş, acil iş diye başına ekşirseniz, o zaman 500 beygirlik bu gücü birinci viteste parkurda koşturuyorsunuz demektir. Bu programcı bir gün ikinci vitese takar ve İstanbul Park'ın arka çıkış kapısından kaçar gider.

Programcılar sahip oldukları potansiyeli kullanarak, işlerini düzenli bir gidişat çerçevesinde en iyi şekilde yapmak isterler. Rahatsız edildikleri bir ortamda tam randımanlı çalışamazlar. Doğrusunu söylemek gerekirse, canları çok sıkıldığında çekip giderler.

Bunun önüne geçmek için şu acil iş ismi altında deklare edilmiş kime ne faydası olduğu belli olmayan anormal durumların bir son bulması gerekiyor. Aksi taktirde 500 beygir gücündeki spor arabalar Edirme-Kars arasında birinci viteste gidip, gelmeye devam ederler. İstediğinizin bu olduğunu zannetmiyorum.

Dağın Ayağına Gelmesini Bekleyen Birisi

http://www.kurumsaljava.com/2014/04/09/dagin-ayagina-gelmesini-bekleyen-birisi/

Bir varmış, bir yokmuş. Evvel zaman içinde, kalbur saman içinde birisi varmış. Bu birisi çok inatçıymış. Birgün bir arkadaşı ile "ben dağa gitmem, dağı ayağıma getiririm" diye iddiaya girmiş.

Başlamış dağın ayağına gelmesini beklemeye. Bir zaman beklemiş, bakmış dağın geldiği falan yok. Bu eninde sonunda ayağıma gelecek diye beklemeye devam etmiş. Yine aradan bir zaman geçmiş. Bu durum dağın umurunda bile değilmiş. Birisi o zaman ben dağa biraz daha yaklaşayım, bakarsın dağ fikrini değiştirir ve ayağıma gelir demiş. Dağa doğru biraz ilerlemiş ve beklemeye başlamış. Gel zaman, git zaman dağdan ses yok, ne gelen varmış, ne giden. Galiba birisi dağı hala beklemeye devam ediyor…

Bu kısa hikayecikte dağın insanın ayağına gelmesini beklemeyi hayattan olan beklentilerimizi simgeleyen metafor olarak seçtim. İnsanlar hayattan her zaman büyük beklentiler içindedirler. Beklentinin fiil hali beklemektir. Beklemek pasif bir durumdur. Oturup pasif olarak bir şeyleri bekleriz, hikayedeki birisi misali.

Zamanla beklenti içinde olmakla hedef sahibi olmak durumsal olarak birbirine karışabilir. Birey beklentilerini düşünerek hedef sahibi olduğunu zanneder. Örneğin okulu bitirdikten sonra iş hayatına atılarak iyi bir gelecek sahibi olma beklentisine sahip oluruz. Bu beklenti zaman içinde hedef kılığına bürünür ve iyi bir gelecek sahibi olmayı hedeflediğimizi zannederiz. Oysaki pasif bir şekilde her şeyin beklediğimiz şekilde yolunda gitmesini ümit ederiz. Bu kendimize koyduğumuz bir hedef değil, bir beklentidir. Pasif bir şekilde bir şeylerin olmasını bekleriz. Pasif olmak faydalıdır, çünkü boşuna enerji harcanmasını engeller. Tek faydası da budur.

Dağı ayağına getirmeye çalışmak ya da iyi bir geleceğe sahip olmak beklentiden başka bir şey olamaz. Beklemek ve pasif olmak yerine hedef güdümlü olmak gerekir. Hedef bireyi mıknatıs gibi çekerken, beklenti hedefi mıknatıs gibi çekmeye çalışır. Hedefi kendine çekmenin ümitsiz bir vaka olduğunu hikayede gördük. Koca dağı kendine çekmeye çalışan birisi bu işte pek başarılı olamadı. Başarılı olmak için bir yerlere doğru, yani hedefe doğru çekilmemiz gerekir.

Hedef bireyi kendine doğru çekebilmek için bir şeye daha ihtiyaç duyar. Bu motivasyonun kendisi ya da motive edici sözler değildir. Bu daha ziyade hedefe doğru giderken nitrojen vazifesi gören, bir an önce hedefe ulaşılmayı isteyen bir şeydir. Bu şey tutkudur (passion).

Hedefe ulaşmak dağa tırmanmak gibi zahmetli bir uğraştır. Düşer, yara alırız. Tutku aldığımız yaralardan doğan acıları dindirir ve bizi tekrar ayağa kaldırır. Yola devam etmemizi sağlar. Tutku olmadan hedefe giden yollar kat edilemez, yollar yarıda bırakılır. Hedefe ulaşılsa bile yolda alınan darbeler o kadar büyüktür ki insan yola çıktığına bin pişman olur. Tutku ile yola çıkan ilacını yanına almıştır. Aldığı darbelerden gelen acıları hissetmez. Tutku bize bir zaman sonra bir şeyi daha gösterir. Bu şey bize bir zaman sonra gidilecek yerin aslında hedef değil, hedefe giden yolun kendisi olduğudur.

Beklenti motoru olmayan bir gemi gibidir. Buna karşın tutku insanı rotasında tutar. Yolculuğun

nereye gittiğine o karar verir, sahibini de pesinde sürükleyip götürür. Peşinde sürüklenme de pasif bir durumdur. Demek oluyor ki tutku gemisine binen belki büyük emekler sarf ederek hedefine ulaşır, ama yolculuk ona elini, kolunu sallayıp, ıslık çalarak yaptığı bir sabah yürüyüşü gibi gelir. İçinde tutkunun olduğu iş aslında iş değildir, eğlencedir. Bu yüzden hedefe ulaşmak kolaydır. Emeği eğlenceye dönüştüren tutkudur.

Demek ki hedefi keşfetmeden tutkuyu keşfetmek gerekiyor. Onu keşfetmek için kalbinizi dinleyin. En çok neyi yapmayı seviyorsanız, tutku onun içinde gizlidir. Onu keşfettikten sonra hedefleriniz sizi bulur.

En Basit Çözümü Oluşturma Yetisi Nasıl Kazanılır?

http://www.pratikprogramci.com/2014/06/01/en-basit-cozumu-olusturma-yetisi-nasil-kazanilir/

Yıllar içinde yazılımcının beyni karmaşık çözümler üretmek için programlanır. Soyutlama yetisi onun müşteri gereksinimi olarak tabir edilen o karmaşanın içinde çabukça şablonlar ve veri yapıları keşfetmesini sağlar. Bu yüksek derecede problem çözme yetisine sahip beyin, kendisinden basit çözümler beklendiğinde, tökezleyip kalır, çünkü programcının gözü yıllarca karmaşık yapılarla uğraşmaktan çok basit çözümleri göremez hale gelmiştir.

Programcı olarak en sık karşılaştığımız problemlerin başında karmaşa nehrinin akıntısında kendimizi kaybederek, doğrudan karşı kıyıya çıkmak yerine, akıntıyla birlikte kilometrelerce aşağıda karşı kıyaya çıkmak, yani beklenenden çok daha çetrefilli çözümler üretmek gelir. Hangi programcıya sorarsanız sorun, iki nokta arasındaki en kısa mesafe nedir sorusuna düz çizgi cevabını alırsınız. Bu çizginin bir ucunun müşteri gereksinimi, diğer ucunun ise programcı tarafından bu gereksinimi tatmin etmek için oluşturulan çözüm olduğunu düşünürsek, programcılar tarafından oluşturulan çoğu çözümün bir çizgi değil, poligon veri bir yapı olduğunu görebiliriz.

Bu madalyonun bir yüzü. Diğer bir yüzü de programcının müşteri gereksinimini anlamadığı durumda nasıl çözümler ürettiği ile ilgilidir.

Kulak misafiri olduğum bir beyin fırtınası seansından örnek vermek istiyorum. Üç senior programcı bir araya gelerek, muallakta kalmış bir müşteri gereksinimi hakkında çözüm üretmeye çalışıyorlar. Ortaya kapsamlı bir çözüm çıkartıyorlar. Akla gelebilecek bilumum senaryolar bu çözümün bir parçası. Kısa bir zaman sonra müşteri ile iletişimi koordine eden yönetici geliyor ve bu çözümün gereğinden fazlasını kapsadığını, müşteri gereksiniminin çok daha basit bir çözümle tatmin edilebileceğini söylüyor. Hatta if/else ihtiva eden sözde kod (pseudocode) oluşturarak, çözümün basitliğine işaret ediyor. Akabinde programcılardan gerçekten böyle basit miymiş gibi söylemler işitiyoruz.

Bu üç programcının müşterinin ne istediğini tam olarak bilmemeleri ve mümkün olan en kapsamlı çözümü oluşturmaya gayret etmeleri, onların gereğinden çok daha karmaşık bir çözümü ortaya koymalarına neden oldu.

İki boolean değişkeni parametre olarak kabul eden bir metot düşünelim. Programcı olarak bu metot gövdesinde dört değişik kombinasyonla baş etmek zorundayız. Parametre sayısı üçe çıktığında, kombinasyon sayısı sekize çıkar. Programcıların çözüm üretirken uyguladıkları düşünce tarzı ne yazık ki metot örneğinde parametre sayısını azaltmak yerine, metodun kabul ettiği parametre sayısını artırma yönündedir. İç güdüsel olarak mümkün olan en geniş kapsamlı çözüme kavuşabilmek için her türlü değişkene değer atayarak, kullanılırlar. Her değişken mümkün olan ama gerekli olup, olmadığı tam olarak bilinmeyen bir senaryo ya da senaryonun bir parçasıdır. Değişken sayısı arttıkça, oluşturulmaya çalışılan çözümün karmaşası ve kapsama alanı artar. Belli bir parametre sayısından sonra beynin o karmaşaya hakim olması ve çözüm üretmesi mümkün değildir. Daha az parametre kullanarak çözüm bulmak zaruri hale gelir. Daha az parametre demek, çok daha basit bir çözüm ortaya koymak demektir. Basit çözüm

demek KISS prensibini uygulamak demektir.

KISS prensibi mümkün olan en basit çözümü uygulamanın gerekliliğini ifade etmektedir. Bu çözümü oluştururken müşterinin yerine kafa yormak yerine, onunla birlikte çözüme ulaşmak için kafa yormak daha gerçekçi ve müşteri tarafından kabul gören bir çözümün oluşmasını kolaylaştıracaktır.

KISS prensibi en basit çözümü tercih etmemizi söylemektedir. Bu çözüme ulaşmak için kullanabileceğimiz iki yöntem bulunmaktadır. Bunlardan birincisi böl ve yönet yöntemedir. Büyük bir problemle uğraşmak yerine, o problemi daha küçük parçalara bölerek, alt problemler için çözümler oluşturabiliriz. Bu çözümler bir araya geldiğinde büyük problemin çözümünü oluştururlar.

Bir diğer yöntem ise çevik süreç olan Extreme Programming'in sahip olduğu Travel Light prensibidir. Az yükle yolculuğa çıkmak, seyahat esnasında yüke değil, yolculuğun kendisine konsantre olunmasını kolaylaştırır. Az yük daha hızlı yol katedilmesini sağlar. Hızlı yol alabilmek, neticelerin somutlaşmasını ve müşteri ile olan diyaloğun oluşturulan prototipler aracılığı ile kıymetli geri bildirimler doğurmasını sağlar. Hızlı yol alabilmek için mümkün olan en basit çözüm ile başlanarak, müşteriden gelen geri bildirim yardımı ile bu genel çözümü daha spesifik hale getirmeye çalışılmalıdır.

Genel bir çözümü müşteri gereksinimleri doğrultuşunda daha spesifik hale dönüştürürken, kod bakımı ve geliştirilmesinin her zaman kolay olması gerektiği konusundaki hassasiyet yitirilmemelidir. Tasarım prensipleri bu konuda yol gösterici olma niteliğini taşımaktadırlar. Örneğin Open Closed prensibi (OCP) kod değişikliği yapmadan uygulamaya yeni davranış biçimleri eklemeyi mümkün kılmaktadır. Stratejik kapama yöntemi uygulandığında uygulamanın gelecekte meydana gelebilecek değişikliklere karşı direnci artırılabilir.

Mümkün olan en basit çözümü oluşturmak için kullanabileceğimiz metotların başında test güdümlü yazılım gelir. Test güdümlü yazılım programcıdan her test için en basit implementasyonu oluşturmasını ister. Her yeni test ile implementasyon yeniden yapılandırma (refactoring) teknikleri kullanılarak yoğrulur ve test için gerekli yapının oluşması sağlanır. Oluşan yeni yapı da en basit çözüm prensibi ile uyumludur. Programcı sadece üzerinde çalıştığı testi tatmin edecek kadar kod yazar. Bu açıdan bakıldığında programcılar en basit çözümü oluşturmak için gerekli disiplini test güdümlü yazılım yaparak edinebilirler.

Sözde Lean!

http://www.pratikprogramci.com/2014/06/04/sozde-lean/

Super lean Grails Micro-Services diye bize sattıkları koda bir bakın:

```
@Grab("com.h2database:h2:1.3.173")
import grails.persistence.*

@Entity
@Resource(uri='/books')
class Book {
    String title
}
```

Bu lean değil, patlamak üzere olan saatli bir bomba. Tek bir sınıf bünyesinde nasıl başardılarsa, sınıfın hem veri tabanı için gerekli JDBC sürücüsü konfigürasyonunu, buradan bir java.sql.Connection nesnesi edinme işlemini, JPA anotasyonları ile bu sınıftan olan nesneyi bir veri tabanında kalıcı hale getirmeyi ve @Resource anotasyonu ile bu sınıfı /books adresi altında bir REST kaynağı olarak dış dünyaya açmayı başarmışlar. Sınıf hangi veri tabanı sisteminin kullanıldığını bile bilyor. Böyle bir şeyin çok serin (cool) olduğunu düşünenler olabilir. Ben aynı fikirde değilim! Neden mi?

- Bu sınıf tek sorumluluk prensibine çok aykırı bir yapıda. Bu sınıfın en az dört değişik sorumluluğu bulunuyor. Bu sınıf dört değişik sebepten dolayı kırılabilir. Yazılım sistemlerinin zamanla bakılamaz ve geliştirilemez hale gelmelerinin ana sebeplerinin başında tek sorumluluk prensibine uyumlu yapıların oluşturulmaması gelmektedir.
- Gerçek iş mantığına sahip kodla, bu koda olan erişim örneğin cephe ya da business delegate tasarım şablonu ile birbirinden ayrılmalıdır. İş mantığı kendisine nasıl erişildiğini bilmemelidir. Aksi halde yukarıda yer alan kod örneğinde olduğu gibi erişim teknolojisini kendisi yönetmek zorundadır ki bu da sınıfa ikinci bir sorumluluk yükler. Erişim teknolojisi değiştiğinde sınıfın buna göre adapte edilerek, yeniden derlenmesi gerekir.
- Uygulama bünyesinde yer alan alan nesneleri (domain objects) birebir dış dünyaya açılmamalıdır. Yukarıda yer alan örnekte bir alan sınıfı olan Book bir REST kaynağı olarak dış dünyanın kullanımına sunulmuştur. Bu yazılım sistemini kullananların yazılım bünyesinde olup, bitenler hakkında çok fazla detay bilgiye sahip olmalarına sebep verebilir. Bunun yani sıra uygulama bünyesindeki alan nesnelerinde meydana gelen değişikler dış dünyadaki kullanıcıları doğrudan etkiler. Bunun yerine gerekli verileri taşımak amacıyla request, response veri sınıfları tanımlanmalıdır. Bu sınıflar kullanıcılar ile uygulama arasında oluşturulan kullanım kontratının (contract) bir parçasıdır.
- Uygulama hangi veri tabanı sistemini kullandığını bilmemelidir. Bu tür bilgiler uygulamanın değişik ortamlara göre ihtiyaç duyduğu konfigürasyonun bir parçasıdır. Kod değişikliği yapmadan konfigürasyon değiştirilebilir yapıda olmalıdır. Örnegin Spring ve Dependency Injection kullanılarak uygulama değişik ortamlara göre konfigüre edilebilir.
- Sadece iş mantığını ihtiva eden bir servis ya da sınıfı test etmek çok daha kolaydır. Yukarıda yer alan sınıfı test edebilmek için veri tabanı bağlantısının oluşturulması, sınıfın beli bir uygulama sunucusuna konması gibi işlemler programcı olarak sadece entegrasyon testleri

yazmamıza izin verecektir. Birim testleri yazamadığım için yeniden yapılandırma (refactoring) yetimi kaybetmiş olabilirim. Sadece entegrasyon testleri kullanılarak kodu seri bir şekilde yeniden yapılandırmak mümkün değildir. Bu iş için birim testleri gerekmektedir. Birim testleri yazabilmek için de iş mantığının POJO (Plain Old Java Object) yapısında olması gerekmektedir.

Bu super lean gibi sıfatlarla kendilerini pazarlayan çatılarla ciddi anlamda yazılım sistemleri geliştirmek mümkün değildir. Bu çatılar sadece hızlı bir şekilde prototip yapmak için kullanılabilirler. Eğer maksat her şeyi monolitik bir sınıfta toplamaksa, bunu kendim de yapabilirim. Bunun için super lean bir çatıya ihtiyacım yok!

Programcının Hayatını Kolaylaştıran 18 Alışkanlık

http://www.pratikprogramci.com/2014/07/05/programcinin-hayatini-kolaylastiran-18-aliskanlik/

1. Her Bug İçin Bir Test Yazılması

Bir hatayı gidermiş olmak, o hatanın tekrar etmeyeceği anlamına gelmez. Tekrarı durumunda kullanıcıların değil, programcı olarak bizim kısa bir sürede tekrar eden hatayı keşfetmemiz gerekir. Bunun da en kolay yolu, keşfettiğimiz her hata için bir birim testi yazmaktır. Birim testleri hatanın tekrarı durumunda bize en kısa sürede geri bildirim sağlarlar.

Ne yapılmalı?
Keşfedilen her hata için bir test yazılmalıdır.

2. Commit Öncesi Testleri Çalıştırılması

Eğer sürekli entegrasyon yapmıyorsanız ve kod kırılmaları kimsenin umurunda değilse, bu bölümü geçebilirsiniz ;-) Lakin projenizde çok temel şeylerin yanlış gittiğini bilmelisiniz.

Sürekli entegrasyon mekanizmalarının kullanıldığı ortamlarda çalışmayan kodun commit edilesi tüm şimşekleri üzerinize çekebilir. Kodun çalıştığını kendimize ispat etmek için commit öncesinde mevcut testlerin yerel bilgisayarda çalıştırılması, mevcut hataların keşfedilmesi adına faydalı bir işlem olacaktır.

Ne yapılmalı?
Kodların versiyon kontrol sistemine eklenme işleminden önce, kırılmalar olup, olmadığını anlamak için tüm testler koşturulmalıdır.

3. Değişikliklerin Bir Seferde Versiyon Kontrol Sistemine Eklenmesi

Yerel testlerin çalışıyor olması, değişiklikler bir seferde commit edilmediği sürece bir şey ifade etmez, çünkü eksiklerden ötürü sürekli entegrasyon sunucusu uygulamayı yapılandıramayacağı için hata verecektir. Bu durumda şimşekleri yine üzerinize çekersiniz.

Ne yapılmalı?
Değişikliğe uğrayan tüm kaynak kod ve dosyaların bir seferde versiyon kontrol sistemine eklenmelidir.

4. Sürekli Entegrasyon Sunucusunun Kontrol Edilmesi

Yerel testlerin çalışıyor ve tüm değişikliklerin bir seferde versiyon kontrol sistemine eklenmiş olmaları, yaptığımız değişikliklerin uygulamanın diğer parçaları ile başarılı bir şekilde entegre edileceği garantisini vermez. Commit etmeyi unuttuğumuz mutlaka bir şeyler vardır ve bu bir sonraki entegrasyonun başarısız olmasına sebep vererek, tüm şimşekleri yine üzerimize çekmemizi sağlayacaktır. Şimşekleri seviyorsanız, bir sorun yok!

Ne yapılmalı?
Her commit işleminden sonra sürekli entegrasyon sunucusu kontrol edilerek, uygulamanın başarılı bir şekilde entegre edilip, edilmediği kontrol edilmelidir.

5. Sonar Verilerinin İncelenmesi

Yazdığımız kodun hangi durumda olduğunu anlamak için statik kod analizi yapan araçlardan faydalanabiliriz. Bu araçların başında checkstyle, findbugs, jdepend ve pmd gelir. Bu araçlardan edindiğimiz verileri Sonar sunucusu aracılığı ile inceleyebiliriz. Jenkins sürekli entegrasyon sunucusunun Sonar plugini bulunmaktadır. Bu plugin aracılığı ile her entegrasyon sonunda kodun statik analizi otomatik olarak yapılır. Verilerin incelenmesi de bize kalır.

Ne yapılmalı?
Her commit işleminden sonra Sonar aracılığı ile kodun yapısal durumu incelenmeli ve iyileştirmeye gidilmelidir.

6. Commit & Run Yapılmaması

Önümüzdeki cuma günü mesai bitimiyle iki haftalık tatile çıkacağınızı farzedelim. Yapmamanız gereken bir şey varsa, o da cuma günü mesai bitimine beş dakika kala üzerinde çalıştığınız dosyaları versiyon kontrol sistemine eklemektir. Murphy'nin kanunlarına göre ters gidebilecek her şey ters gider ve yaptığınız eklemelerle derleyemeyen bir uygulamayı arkanızda bırakırsınız. Artık arkanızdan neler konuşulduğunu siz düşünün. En kötü ihtimalle tatilde sizi patronunuz arayarak, bunun hesabını soracaktır.

Ne yapılmalı?
Tatile çıkılacaksa, yapılan işlerin devredilebileceği bir takım arkadaşı aranmalı ve yarım kalan işler birkaç gün öncesinden devredilmelidir.

7. Ne Yapılacağının Bilinmemesi

Yeni bir görev üstlendiğimizde, ilk etapta kafamızda neler yapacağımızı canlandırmaya çalışırız. Kafamızda bir zaman sonra bir takım sınıf isimleri uçuşmaya başlar. Bir yerlerden başlayıp, bu sınıfları implemente etmeye çalışırız. Müşteri isteklerini tatmin edecek uygulama parçasını geliştirmenin en verimsiz yöntemlerinden bir tanesi budur. Sistematik bir şekilde ilerlemediğimiz ve kafamızdaki bir takım hayallere bel bağladığımız için ne olduğu belli olmayan bir şeyler ortaya çıkar.

Bunun yerine bir birim testi yazarak uygulamayı geliştirmeye başlamalıyız. Bu test güdümlü yazılımda kullanılan bir yöntemdir. Testler uygulamadan olan beklentileri ifade etmek için kullanılabilecek en uygun araçlardır. Testler programcıya ne yapması gerektiği hakkında fikir verirler. Çoğu zaman testler uygulamayı kullanıcı gözünden görürler. Bu programcının yapılması gerekenleri daha çabuk kavramasını sağlar.

Ne yapılmalı?
Yeni bir göreve test yazarak başlanmalıdır. Test güdümlü yazılım yapılmasa da, testler bünyesinde ifade edilen beklentiler, programcının hangi program arayüzlerine (API – Application Programming Interface) ihtiyaç duyulduğunu anlamasını kolaylaştırır.

8. Uygulamanın Neden Çalışmadığının Anlamaya Çalışılması

Java gibi dillerde program modüllerini paralel koşturmak için threadler kullanılır. Reaksiyon göstermeyen uygulamalarda genelde thread bazında bir sorun vardır. Threadsel sorunları

anlamak için thread dump yapılmalıdır. Bir Java uygulamasından "kill -QUIT pid" komutu ile thread dump alınabilir.

Ne yapılmalı?
Uygulamalar çok değişik sebeplerden ötürü çalışmaz hale gelebilir. Uygulama neden çalışmıyor acaba sorusunu sormak yerine, bir thread dump alarak, uygulamanın neden çalışmadığı analiz edilmelidir.

9. İzci Kuralına Uyulması

Kötü olduğunu düşündüğümüz bir kod parçası hakkında şikayet etmek yerine, o kodu daha iyi hale getirmeye çalışmalıyız. Bir izci kuralına göre izciler kamp yaptıkları yeri, buldukları halinden daha iyi bir şekilde geride bırakırlar.

Ne yapılmalı?
İzci kuralı uygulanmalı ve denk gelinen ve kötü olduğu düşünülen kod parçaları iyileştirilmelidirler.

10. Eve İş Taşınmaması

Programcı olarak yazdığımız tüm testlerin çalışır durumda olduğuna gayret göstermeliyiz. Çalışan ve yeşil olan testler bize her zaman manevi rahatlık verir. Özellikle mesai bitmeden önce çalıştırılan ve yeşil oldukları görülen testler, öz güveni ve huzuru artırır.

Ne yapılmalı?
Mesai bitmeden önce tüm testler çalıştırılmalı ve yeşil oldukları görülmelidir. Yeşil olan tüm testler programcının yaptığı işleri arkasında bırakarak, huzur içinde evine gitmesini sağlar. Eve iş taşımamanın en güzel yolu, her zaman yeşil olması sağlanan testlerdir.

11. NullObject Tasarım Şablonunun Kullanılması

Programcı olarak başımızı ağrıtan hataların başında NullPointerException gibi hatalar gelir. Yazdığımız kodların büyük bir bölümü null check yapar, çünkü başka bir programcı tarafından yazılan metotların hangi değerleri geri verdiklerinden emin değilizdir. Object veri tipinde bir nesneyi geri veren bir metodun null değerini geri verme ihtimali çok yüksektir.

Ne yapılmalı?
Oluşturulan metotlarda null değeri yerine boş bir nesne geri verilmelidir. Örneğin bir metot List tipinde bir listeyi geri veriyorsa ve bu metot bünyesinde liste oluşturulamadı ise, Collections.EMPTY_LIST ya da new ArrayList() geri verilmelidir. Bu metot kullanıcılarının null check yapmasını engeller ve kodun daha okunur olmasını sağlar.

12. Tek Sorumluluk Prensibine Sadık Kalınması

Nesneye yönelik programlamada sınıf ve metotların çok uzun olmalarının başlıca sebebi, kodun tek sorumluluk prensibine sadık kalınarak, geliştirilmemiş olmasıdır.

Ne yapılmalı?
Oluşturulan her sınıf ve metot tek bir sorumluluğu yansıtacak yapıda olmalıdır. Tek bir

sorumluluğu olan bir sınıf ya da metot tek bir sebepten dolayı değişikliğe uğrayabilir. Birçok sorumluluğu olan bir sınıf ya da metot birçok sebepten dolayı değişikliğe uğrayacaktır. Birçok uygulamanın en ufak bir değişiklikte çalışmaz hale gelmelerinin başlıca sebebi, uygulamaların temelini oluşturan yapıların bünyelerinde birçok sorumluluğu barındırıyor olmalarında yatmaktadır. Genel olarak programcı kod yazarken SOLID prensiplerine sadık kalmaya çalışmalıdır.

13. En Basit Çözümü Oluşturmaya Gayret Edilmesi

Zaman içinde uygulamalar zaten karmaşık bir yapıya bürünürler. Bu karmaşanın önüne geçmenin en kolay yolu KISS prensibini uygulamaktır.

Ne yapılmalı?
Mümkün olan en basit çözüm tercih edilmelidir. En karmaşık çözümü ortaya koyan değil, aynı sorunu en basit şekilde çözen 1-0 öndedir.

14. Kod İnceleme Seanslarının Yapılması

Yazdığımız kodun hangi durumda olduğunu anlamak için başka bir çift gözün kodu incelemesini sağlamalıyız. Biz programcı olarak yazdığımız kodun iyi olduğunu düşünürüz. Bu subjektif bir algıdır. Bakalım takım içindeki arkadaşlarımız da aynı fikirde mi?

Ne yapılmalı?
Bir görevi tamamladıktan sonra takım içinden bir programcı seçilerek, birlikte kod inceleme seansı yapılmalıdır. Kod inceleme seansından alınan geri bildirim ile daha iyi bir programcı olma yolunda ilerlenir.

15. Test Yazarken Ekran Çıktılarının Göz Ardı Edilmesi

Test yazarken ekran çıktılarına güvenmek ve test metotlarını cılız bırakmak, testlerin zayıf ve uygulamanın kırılgan olmasını sebep verir.

Ne yapılmalı?
Test metotları test edilen senaryoyu en kapsamlı şekilde test etme eğilimi göstermelidir. Test edilmedikleri sürece ekran çıktıkları uygulamanın doğru çalışıyor olduğunun garantisi olamaz.

16. Temiz Kod Yazılması

Kodun bakımı ve genişletilebilirliği, okunabilirliği ile doğrudan orantılıdır. Programcı olarak yazdığımız metotların kısa olmasına dikkat etmemiz gerekir. Uzun olan metotları otomatik refactoring araçları ile küçültebilir ve okunabilir hale getirebiliriz. İdeal bir metot dört ya da en fazla beş satırdan oluşmalıdır.

Ne yapılmalı?
Kullanılan IDE'nin sunduğu otomatik refactoring yöntemleri ile metotlar daha okunabilir hale getirilmelidir.

17. Soyut Sınıfların Kullanılması

Tasarım aracı olarak soyut sınıflar kullanılmalıdır. Somut sınıflara nazaran soyut sınıflar daha

az değişikliğe uğrama eğilimi gösterirler.

Ne yapılmalı?
Daha iyi bir tasarım için DIP uygulanmalıdır.

18. Pratik Yapılması

Biz programcıların hiç ya da çok az pratik yaptığı bir gerçek. Oysaki pratik yapmak daha iyi programcı olabilmek için bir gerekliliktir.

Ne yapılmalı?
Kod kata yaparak, programcılık becerileri geliştirilmelidir

Müşteri Gereksinimlerini Anladığımızdan Nasıl Emin Olabiliriz?

http://www.pratikprogramci.com/2014/07/19/musteri-gereksinimlerini-anladigimizdan-nasil-emin-olabiliriz/

Tipik bir yazılımcı kendisine verilen bir işi görevlere (task) böler ve kod yazmaya başlar. Çoğu zaman bir işi parçalara bölme ve görevler oluşturma işlemi ekip içinde de yapılır. Bir işin parçası olan görevler ekip içinde paylaştırıldıktan sonra, herkes üzerinde düşeni yapmaya başlar.

Bir işi parçalara bölmek ve bu şekilde bir işi tamamlamaya çalışmak seçilebilecek en verimsiz yöntemlerden bir tanesidir. Bunun neden böyle olduğu bu yazımda açıklamaya çalışacağım.

Programcı olarak bize şöyle bir iş verilmiş olsun:

```
Kullanıcı e-posta ve şifresini kullanarak login yapabilir.
```

Şimdi bu işten hangi görevlerin çıkabileceğine bir göz atalım:

- Kullanıcı e-posta ve şifresini bir form aracılığı ile edin.
- Kullanıcının girmiş olduğu şifreyi MD5 yöntemi ile dönüştür, çünkü veri tabanında şifreler MD5 yönetimi kullanılarak tutuluyor.
- Kullanıcının IP adresini session isimli veri tablosunda tut.
- Bu tablo yoksa, tabloyu oluştur.
- Kullanıcının hatalı girişlerini login_failed tablosunda tut.
- Kullanıcı üç kez hatalı giriş yaptı ise, hesabını dondur.

Yazılımcılar böl ve yönet prensibi ile büyük problemleri parçalarına bölüp, parça çözümlerini bir araya getirerek, problemin genel çözümüne ulaşabileceklerini bilirler. Bunun yanı sıra yazılımcılar bir işi görevlere bölmeye eğilimlidirler, çünkü ne yapmaları gerektiğini hemen anlamak ve bilmek isterler. Bu görev listesine baktığımızda ne yapmamız gerektiğini hemen görebiliyoruz değil mi?

Lakin göremediğimiz bazı noktalar bulunmaktadır. Bunlar:

- Bu görev listesi bize işi bitirme konusunda ne kadar ilerleme kaydettiğimizi söyleyebilme kabiliyetine sahip mi?
- Görev listesine bakarak, müşterimizin bizden ne istediğini tam olarak anladığımızı söyleyebilir miyiz?
- Müşterimiz görev listesine baktığında, sahip olduğu gereksinimin bizim tarafımızdan doğru anlaşıldığını tespit edebilir mi?
- Bu görevleri tamamladığımızda ne oranda müşterimizin sahip olduğu gereksinimi tatmin ettiğimizi kestirebilir miyiz?

Görev listesine göz attığımızda, **“şu veri tabanı tablosunu oluştur”** gibi görev tanımlamalarının çok teknik şeyleri ifade ettiğini görmekteyiz. Bu görev listesi bize işin nasıl yapılacağı konusunda fikir verebilmektedir. Lakin bir görevi tamamladığımızda, genel anlamda

ne kadar ilerleme kaydettiğimizi kestirmemiz çok zordur. Görev listesinde yer alan görevlerin tamamlanma oranlarından yola çıkarak, ne kadar işimizin daha kaldığını söylememiz mümkün değildir, çünkü görev tanımlamaları bu bilgiyi yansıtma yetisine sahip değildirler. Yazılımcı olarak görev bazında çalıştığımızda, işin neresinde olduğumuzu ve daha ne kadar zamana ihtiyacımız olduğunu kestirmemiz mümkün olmayacaktır.

Bir müşteri gereksinimini tatmin edecek kodu yazmaya başlamadan önce bu gereksinimin ne olduğunu birçok yönüyle anlamış olmamız gerekmektedir. Yukarıda yer alan görev listesi, müşterinin sahip olduğu gereksinimi ne oranda anladığımızı yansıtmaktan acizdir. Görev listesi çok teknik bilgi ihtiva etmektedir. Oysaki müşterinin dile getirdiği gereksinimler, oluşturulmak istenen uygulamaya en tepeden bakar yapıdadırlar. Müşteri için önemli olan bir görevin nasıl yerine getirildiği değil, uygulamanın sergilediği davranış biçimlerinin sahip olduğu gereksinimleriyle ne oranda örtüştüğüdür. Bu yüzden müşterimiz bu görev listesine baktığında, bizim onu ne kadar iyi anladığımızı kestirmesi mümkün değildir. Aynı şey bizim için de geçerlidir. Görev listesine bakarak, müşteriyi ne kadar iyi anladığımızı kestiremeyiz, çünkü görevler bir takım teknik işlevleri ifade etmektedirler ve müşterinin ne istediğine dair bilgiden bihaberdirler.

Bir müşteri gereksinimini koda dönüştürmeden önce, onun ne istediğini ve bizim programcı olarak ne anladığımızı gösteren bir araca ihtiyaç duymaktayız. Bu aracın ismi onay/kabul (acceptance testing) testleridir.

Şimdi müsteri gereksinimini tekrar hatırlayalım ve görev listesi oluşturmak yerine onay/kabul test listesi oluşturalım ve aradaki farkı inceleyelim.

Bize verilen görev şu şekilde idi:

```
Kullanıcı e-posta ve şifresini kullanarak login yapabilir.
```

Şimdi bu gereksinimden yola çıkarak, onay/kabul testleri oluşturalım. Benim aklıma ilk etapta gelen testler şu şekildedir:

- Kullanıcı e-posta ve şifresini girmeden login butonuna tıklar. Bu durumda uygulama Kullanıcıya "Lütfen e-posta ve şifrenizi giriniz" hatasını gösterir.
- Kullanıcı sadece e-posta adresini girer ve login butonuna tıklar. Bu durumda uygulama kullanıcıya "Lütfen şifrenizi giriniz" hatasını gösterir.
- Kullanıcı e-posta ve şifresini girer ve login butonuna tıklar. Geçersiz bir e-posta adresi olması durumunda, kullanıcıya "Lütfen geçerli bir e-posta adresi giriniz" hata mesajı gösterilir.
- Kullanıcı e-posta ve şifresini girer ve login butonuna tıklar. Şifrenin geçersiz olması durumunda, Kullanıcıya "Giriş işlemi hatalı" hata mesajı gösterilir.
- Kullanıcı e-posta ve şifresini girer ve login butonuna tıklar. E-posta adresi ve şifre geçerlidir. Login işlemi gerçekleşir ve Kullanıcı ana sayfaya yönlendilir.

Bu listeye baktığımızda hemen dikkatimizi çeken birkaç nokta bulunuyor. Bunlar:

- Görevler teknik bazda değil, daha ziyada uygulama kullanıcısı gözünden bakılarak

oluşturulmuş.
- Hem programcı hem de müşteri bu listeye bakarak, ne yapılması gerektiğini anlayabilir.
- Çalışır hale getirdiğimiz her onay/kabul testi ile ne kadar ilerleme kaydettiğimizi görebiliriz. Bunun yani sıra ne kadar işimizin kaldığını kestirmemiz de kolaylaşmaktadır.

Biz programcılar ne yazık ki bize verilen görevleri teknik açıdan analiz etme eğilimi gösteriyoruz. Bir programcıya bir ev yap deseniz, programcı ev sahibi için önemli olan yaşam sahasının kalitesi ve genişliği gibi konularla ilgilenmek yerine, kullanılan betonun kaç derecede daha iyi kuruduğunu, kapı paspasının nereye konması gerektiği konularına öncelik verecektir. Burada programcı olarak gözden kaçırdığımız bir durum mevcut:

"Yaptığımız işle müşteriye sağladığımız katma değer nedir?"

Programcı olarak yaptığımız her işte, her daim bu sorunun cevabını aramalıyız. Bu sorunun cevabını bulmanın en kolay yolu, onay/kabul testleri oluşturmak ve kodu bu testler eşliğinde şekillendirmektir. Eğer bir müşteri gereksinimini koda dönüştürmeden önce görev listesi oluşturmak yerine onay/kabul test listesi oluştururursak, her iki tarafın da ne yapılması gerektiği ve ne yapıldığı konusunda fikir sahibi olmaları kolaylaşacaktır.

Onay/kabul testlerine nasıl tanımlanır sorusu aklınıza gelmiş olabilir. Onay/kabul testlerine gelmeden önce onay/kabul kriterlerinin tanımlanmış olması gerekmektedir. Onay/kabul kriterleri müşteri tarafından tanımlanırlar ve müşterinin uygulamadan olan beklentilerini ihtiva ederler. Normal şartlar altında uygulamanın nasıl bir davranış sergilemesi gerektiğini en iyi bilebilecek olan şahıs müşteridir. Bu yüzden onay/kabul kriterlerinin müşteri tarafından tanımlanması, yapılması gereken çalışmanın çerçevesini oluşturacaktır. Bu kriterlerden yola çıkarak, onay/kabul testlerini tanımlayabilir ve uygulamayı bu testler doğrultusunda şekillendirebiliriz.

Tanımlamış olduğumuz onay/kabul testlerini test güdümlü yazılım yapmak için kullanabiliriz. Bu tür test güdümlü yazılıma ATDD (Acceptance Test Driven Development) ismi verilmektedir.

Nasıl tasarım şablonları (design patterns) programcılar arasında ortak bir kelime hazinesi oluşturma özelliğine sahip ise, onay/kabul testleri de müşteri ile programcı arasında ortak bir dilin oluşmasını sağlama özelliğine sahiptirler. Ben şahsen onay/kabul testleri olmadan bir uygulama geliştirmeye çalışmanın harikiriden farksız olduğunu düşünüyorum. Onay/kabul testlerini müşterinin vizyonu olarak düşünmeliyiz. Yaptığımız işte takip edebileceğimiz bir vizyon olmadığı taktirde, rotamızın nereye doğru gittiği konusunda bir fikir sahibi olamayız. Rotası belli olmaya bir geminin de hangi limana gittiğini kestirmek mümkün değildir. Düşündüğünüz limana gitmeyeceğinden emin olabilirsiniz.

5 Adımda Daha Kaliteli Yazılım Testleri

http://www.pratikprogramci.com/2014/07/29/5-adimda-daha-kaliteli-yazilim-testleri/

Yazılım testleri can yeleği olma özelliğine sahip olmalarına rağmen, yazılımcılar tarafından göz ardı edilme eğilimi yaşayan en önemli yazılım disiplinlerinden birisidir. Çalıştığım projelerde testlerin hak ettikleri ilgi ve alakayı görmediklerini gözlemliyorum. Edindiğim izlenimler şu şekilde:

Bazı yazılımcılar test yazma konusuna hiç ilgi göstermezler. Bazı yazılımcılar test yazarlar, lakin test kodunu işletme mantığının yer aldığı koddan ayrı tutalar, yani test koduna üvey evlat muamelesi yaparlar. Bu test kodunun zaman içinde çok karmaşık hale gelmesi ya da zamanla gerçekleri yansıtmaması anlamına gelebilir. Bazı yazılımcılar test yazarlar. Test koduna üvey evlat muamelesi yapmamalarına rağmen, test kodunun okunabilirliğini artırmak için efor sarf etmezler. Bu tür testlerin anlaşılması, bakımı ve geliştirilmeleri zordur. Bazı yazılımcılar test kodunun ne kadar kıymetli olduğunu bilirler ve güncel ve sade kalmaları için ellerinden gelen her türlü çabayı sarf ederler. Testleri uygulamayı kullanan sanal kullanıcılar olarak düşünebiliriz. Testlere baktığımız zaman, işletme mantığının yer aldığı kodu incelemeden, uygulamanın nasıl çalıştığı hakkında çok kısa bir sürede bilgi sahibi olabiliriz. Bunun gerçekleşebilmesi için testlerin sade bir dilde yazılmış olmaları gerekmektedir. Bu yazımda bu amaca nasıl yaklaşabileceğimizi göstermek istiyorum.

Aşağıda Customer sınıfını test ettiğini zanneden, ama aslında ödeme işlemini test eden bir birim testi yer almaktadır.

```
// Kod 1

@Test
public void testCustomer() throws Exception {

    final Customer customer = new Customer();
    customer.setName("Acar");
    customer.setFirstname("Oezcan");

    final Order order = new Order();
    order.setOrderAmount(100);
    order.setCustomer(customer);

    final Payment payment = new Payment();

    final PaymentResult result = payment.pay(order);
    Assert.assertTrue(result.success);

}
```

Test metot isimleri metot bünyesinde yapılan işlemleri ifade edecek güçte olmalıdırlar. Metot ismine bakarak, neyin test edildiğini anlamak kolaydır ya da kolay olmalıdır. Bu yüzden seçilen ismin gerçekleri yansıtması gerekir. Bu yüzden metot ismini ilk etapta testPayment olarak değiştirmemizde fayda görüyorum. Daha sonra bu ismi değiştirerek, yapılan spesifik testin ne

olduğunu ifade etmesini sağlayacağız.

Kod 1 de yer alan test kodunun ilk bakışta karmaşık bir yapıda olduğunu görüyoruz. Bunun başlıca sebebi müşteri, sipariş ve ödeme nesnelerinin oluşturulmaları ve metot bünyesinde gerekli veriler ile donatılmalarıdır. Bu gereğinden fazla kodun oluşmasına sebep olmaktadır. Bir metot bünyesindeki kod sayısı arttıkça, kodun okunurluk seviyesi düşer. Bu yüzden bir metoda mümkün mertebe az kodun konuşlandırılmasını sağlamamız gerekmektedir.

Kod 1 de yer alan test metodunun okunurluk seviyesini nasıl artırabiliriz? Test metodunun merkezinde Order sınıfı yer almaktadır. Order sınıfı için bir fluent (akıcı arayüz) interface oluşturarak, nesne oluşturma ve işlem yapma sürecini şu şekilde daha okunur hale getirebiliriz.

```
// Kod 2

Customer customer;

@Before
public void setup() {
    customer = new Customer("Oezcan", "Acar");
}

@Test
public void testPayment() throws Exception {

    final Order order = OrderBuilder.order().withCustomer(customer).withOrderAmount(100)
                            .returnOrderObject();

    final Payment payment = new Payment();

    final PaymentResult result = payment.pay(order);
    Assert.assertTrue(result.success);
}

public static class OrderBuilder {

        Customer customer;
        Order order;

        public PaymentResult pay(final Order order) {
            return new PaymentResult(true);
        }

        public Order returnOrderObject() {
            if (order == null) {
                throw new IllegalStateException("lütfen daha önce order nesnesini olusturur
            }
            return this.order;
        }

        public OrderBuilder withOrderAmount(final int amount) {
            order = new Order();
            order.setOrderAmount(amount);
            if (customer != null) {
                order.setCustomer(customer);
            }
```

```
            }
            return this;

        }

        public OrderBuilder withCustomer(final Customer customer) {
            this.customer = customer;
            return this;

        }

        public static OrderBuilder order() {
            return new OrderBuilder();

        }
    }
```

OrderBuilder sınıfını fluent interface türünde oluşturdum. Kod 1 e baktığımızda, müşteri ve sipariş oluşturma işleminin altı satırı kapsadığını görmekteyiz. Kod 2 de fluent interface yardımı ile tek bir satırda sipariş nesnesini oluşturduk.

Aynı şeyi Payment sınıfı için de şu şekilde yapabiliriz:

```
// Kod 3

Customer customer;

@Before
public void setup() {
    customer = new Customer("Oezcan", "Acar");
}

@Test
public void testPayment() throws Exception {

    final Order order = OrderBuilder.order().withCustomer(customer).withOrderAmount(100)
                                .returnOrderObject();

    final PaymentResult paymentResult = PaymentBuilder.payment().withOrder(order).pay();

    Assert.assertTrue(paymentResult.success);
}

public static class PaymentBuilder {

        Order order;

        public static PaymentBuilder payment() {
            return new PaymentBuilder();
        }

        public PaymentResult pay() {
            final Payment payment = new Payment();
            return payment.pay(this.order);

        }

        public PaymentBuilder withOrder(final Order order) {
            this.order = order;
            return this;
        }

}
```

Her test metodu bünyesinde test etme süreci şu adımlardan oluşur ya da oluşmalıdır:

- Birinci adımda test edilen nesneler oluşturulur. Bu işlemi test için gerekli ortamın oluşturulması olarak düşünebiliriz.
- İkinci adımda test edilmek istenen metot koşturulur.
- Üçüncü adımda elde edilen netice sahip olunan beklentilerle kıyaslanır.

Davranış güdümlü yazılımda (BDD – Behavior Driven Development) metotlar ya da birimler değil, uygulamanın sahip oldugu davranışlar test edilir. Bu tür testler uygulamayı bir kara kutu olarak görürler. Bu şekilde detaylarda kaybolmadan, uygulamanın sahip olması gereken davranış biçimlerini test etmek mümkündür. BDD dünyasında bir davranışı test etmek için Given/When/Then yapısı kullanılır. Given testin çıkış noktasıdır. Bunu test edilen davranış için

gerekli ortamın hazırlanması olarak düşünebiliriz. When bölümünde uygulamanın davranışı tetiklenir. Then bölümünde uygulamanın nasıl bir davranış sergilediği test edilir. Bizim uygulamamızda Given/When/Then yapısını şu şekilde uygulayabiliriz:

```
// Kod 4

@Test
public void testPayment() throws Exception {

    // Given
    final Order order = OrderBuilder.order().withCustomer(customer).withOrderAmount(100)
                            .returnOrderObject();

    // When
    final PaymentResult paymentResult = PaymentBuilder.payment().withOrder(order).pay();

    // Then
    Assert.assertTrue(paymentResult.success);
}
```

Kod 4 bünyesinde BDD tarzı aralar kullanmasak da, testimizi ve beklentilerimiz BDD tarzı yapılandırdık. Bu şekilde hangi işlemlerin yapıldığını bir bakışta anlamak daha kolay bir hale gelmektedir. Buna rağmen OrderBuilder ve PaymentBuilder sınıfları çok fazla detayı ihtiva ettiklerinden, ilk bakışta ne yaptıklarını anlamak zor olabilir. O halde kod 4 de yer alan test metodunu daha sadeleştirmemiz gerekmektedir. Bunun bir örneği şu şekilde olabilirdi:

```
// Kod 5

@Test
public void testPayment() throws Exception {

    // Given
    final Order order = createOrderWithAnAmountOf(100);

    // When
    final PaymentResult paymentResult = payNow(order);

    // Then
    assertPaymentIsSuccessfull(paymentResult);
}

private Order createOrderWithAnAmountOf(final double value) {
    final Order order = OrderBuilder.order().withCustomer(customer).withOrderAmount(value)
                            .returnOrderObject();
    return order;
}

private PaymentResult payNow(final Order order) {
    final PaymentResult paymentResult = PaymentBuilder.payment().withOrder(order).pay();
    return paymentResult;
}

private void assertPaymentIsSuccessfull(final PaymentResult paymentResult) {
    Assert.assertTrue(paymentResult.success);
}
```

Then bölümünde testten olan beklentilerimiz yer almaktadır. Bu yazımda birim testlerinden olan beklentilerimizi daha net nasıl ifade edebileceğimiz konusuna değinmiştim. Yine burada fluent interface yardımı ile beklentilerimizi daha okunaklı hale getirebiliriz.

Kod 5 de yer alan testi herhangi bir BDD aracı kullanmadan oluşturduk. Sadece BDD prensiplerini uygulamaya çalıştık. BDD yapmak için kullanabileceğiniz test araçları bulunmaktadır. Bunlardan birisi easyb. Easyb Groovy dilini kullandığından dolayı, testlerimizin ifade gücünü artırmak için gerçek anlamda DSL (Domain Specific Languages) yapıları oluşturabiliyor ve kullanabiliyoruz. Kod 4 de yer alan testi easyb ile şu şekilde tanımlayabiliriz:

```
Kod 6

scenario "Müsterinin 100 TL'lik yaptigi alisverisin ödeme bölümünü test ediyoruz.", {

  given "Müsteri 100 TL'lik alisveris yapar",{

    order = OrderBuilder.order().withCustomer(customer).withOrderAmount(100)
                    .returnOrderObject();

  }

  when "Müsteri kasaya gidip, ödeme butonuna tikladiginda", {

    paymentResult = PaymentBuilder.payment().withOrder(order).pay();

  }

  then "Ödeme basariyla tamamlanir", {
    Assert.assertTrue(paymentResult.success);
  }

}
```

Test ettiğimiz uygulama davranışı ödeme işlemidir. Easyb yardımı ile kod 6 da yer alan kodu bir test senaryosu haline getirdik. Kod 5 de yer alan kodu da bir test senaryosu olarak düşünebiliriz. Test senaryoları oluşturmak kodun daha okunur ve neyin test edildiğinin daha anlaşılır olmasını sağlamaktadırlar.

Kod 6 da kullandığımız

```
scenario
  given
  when
  then
```

bir alana has dildir (DSL – Domain Specific Language). Burada içinde olduğumuz alan test alanıdır. Easyb aracılığı ile kullandığımız bilgisayar dili 4 direktif ihtiva etmekle birlikte, testlerimiz bünyesinde ifade etmek istediklerimiz için yeterli yapıdadırlar. Küçük olan bu alan dilleri ile yapılan işlemleri daha net ifade etmek mümkündür.

Son olarak test metodunun ismine tekrar bir göz atalım. testPayment çok genel bir isim gibi görünmekte. Test bünyesinde olup, bitenleri ifade etmek için ifade gücü yüksek bir metot ismi seçmemiz faydalı olacaktır. Ben aşağıda yer alan ismi öneriyorum.

```
@Test
public void when_an_order_is_triggered_then_payment_should_be_successfull() throws Exceptio

    // Given
    final Order order = createOrderWithAnAmountOf(100);

    // When
    final PaymentResult paymentResult = payNow(order);

    // Then
    assertPaymentIsSuccessfull(paymentResult);
}
```

Metot isimlerini oluştururken given/when/then yapılarını kullanabiliriz. Bu metot bünyesinde yapılan işlemi birebir dışa yansıtmak için en verimli neticeyi verecektir.

Özetleyecek olursak:

- Test metotları given/when/then şeklinde üç bölüme ayrılarak, testin okunurluğu artırılabilir.
- Fluent interface yapıları hem given bölümünde yer alan test nesneleri yapılandırmak hem de then bölümündeki beklentilerin ifadesi için kullanılabilirler. Fluent interface kullanımı test okunurluk oranını artırır, çünkü insanların kullandığı dile yakın yapıdadırlar.
- Metot isimleri metot bünyesinde yapılan işlemleri ifade edecek güce sahip olmalıdırlar.
- Testlerin uygulama senaryoları olarak hazırlanmaları, neyin test edildiğinin netleşmesine ışık tutarlar.

Bu tarz test yazmayı bir deneyin ve tecrübelerinizi bizimle paylaşın.

Ne Zaman Başımız Göğe Erer?

http://www.kurumsaljava.com/2014/02/21/ne-zaman-basimiz-goge-erer/

Geçenlerde öğrenci bir arkadaş fikrimi almak için bana bir soru sordu. Belli bir meblağ için iki ay boyunca fulltime bir yazılım evi için çalışmasının doğru olup, olmayacağı hakkında fikrimi sordu. Bu okulunu aksatır mı diye sordum. Cevabı evet oldu. Benim de cevabım belliydi.

Yazılımda tek sorumluluk ismini taşıyan bir yazılım prensibi var. Birden fazla sorumluluğu olan sınıflar er ya da geç devrilirler. Aynı şey öğrenciler için de geçerlidir. Okulu yanı sıra iki ay gibi uzun bir süre iş peşinde koşturan iki karpuzu tek kolunda taşıyamaz.

Ben 1996 yılından beri freelancer olarak çalışıyorum. Üniversite yıllarında freelance programcı olarak çalışmaya başladım. İki binli yılların dotcom furyasını hatırlayanlar çok iyi bilirler. O zamanlar Java kelimesini yazanlar bile programcı olarak işe alınırdı. Para kazanmanın tadına varanların çoğu üniversiteyi bitiremediler. Ha bugün, ha yarın derken öğrenim yolda kaldı, business ve para kazanmak öne çıktı.

Paranın en büyük özelliği devamlı harcanmak isteyişidir. Para kazanan harcar ve belli bir zaman sonra belli bir hayat standardına kavuşur ve bu standardı korumak ister. Bu yüzden iki binli yıllarda programcı olarak çalışmaya başlayan bilgisayar mühendisi öğrencilerin çoğu bugün diplomasız hayatlarını sürdürüyorlar.

Ayakta kalabildilerse ne mutlu onlara. 2005 senesinden sonra gördüğüm iş ilanlarının hepsinde istinasız bilgisayar ya da benzeri bir mühendislikten mezun programcılar arandığına şahit oluyorum. Ben şahsen diplomanın şahsın yetenekleri hakkında çok fazla bir şey ifade etmediği kanaatindeyim. İnsan üniversiteden sonra kendi yaptığı işi öğrenmeye başlıyor. Lakin iş verenlerde ne yazık ki bir diploma takıntısı var. Bu sebepten dolayı insan eğer okumak üzere yola çıktı ise, o zaman bu işe diploma alarak bir nokta koymalıdır. Sonuç itibari ile çalışan ve tüketen toplumun ödediği vergilerle devlet üniversiteleri işliyor. Öğrenci bir yerde diplomasını alarak topluma da kullandığı kaynakların hesabını vermek zorunda. İşi yarıda bırakmak olmaz. Bu topluma ihanet olur. O yüzden diploma alınacak, bundan kaçış yok.

Şimdi tekrar yazımın başında bahsettiğim öğrenci arkadaşa geri dönmek istiyorum. Bu arkadaş iki aylık fulltime iş için 700 TL talep ettiğini söyledi. Öncelikle şunu söyleyeyim. Bu rakamın freelance bir iş için 700 değil, 7000 TL olması gerekirdi. Sahip oldukları değeri kestiremedikleri için çoğu öğrenci arkadaş böyle kendisini kullandırıyor. Eğer size teklif edilen işin hakkından geleceğinizden eminseniz, o zaman değeriniz ne ise, onu talep edin.

İkinci husus ise kazanılacak 700 TL için koca bir gelecek vaat eden öğrencilik statüsünün rizikoya sokulmasıdır. 700 TL insanın başını göğe erdirmez, ama diploma erdirir. Öğrenci arkadaşlar bunu unutmamalı. Para her zaman kazanılır, diploma bir kere.

Kitap Okumanın Önemi

http://www.kurumsaljava.com/2013/12/20/kitap-okumanin-onemi/

Şimdi size sorsam, en çok sevdiğiniz üç yazılım kitabını yazarları ile sayabilir misiniz? Sayabilmeniz lehinize olurdu, çünkü bir sonraki iş görüşmenizde bu soruyla karşılaşma şansınız yüksek. Başıma geldiği için söylüyorum :)

Bana son on beş yıllık yazılımcı iş hayatımda sorulan en ilginç soru buydu. Cevabım şu şekilde olabilir.

Bu sorunun altında yatan mentalite çok başka türden. Beni şaşırttı açıkçası. Artık eskisi gibi anlatın bakalım, şimdiye kadar neler yaptınız demiyorlar. Size belli bir zamanda çözmeniz gereken bir problem de sunmuyorlar. Bilginizin hangi temellere dayandığını anlamak için derin sondaj çekiyorlar. Bu soru benim için neden bu kadar ilginç, açıklamaya çalışayım.

Google, Stackoverflow ya da BTSoru.com gibi sistemlerin var olduğu bir dünyada yazılımcılar belli bir sorunun çözümüne birkaç tık sonra ulaşabiliyorlar. Copy/Paste Programcı başlıklı yazımda bunu dile getirmeye çalışmıştım. Çoğu zaman copy/paste yaparak bir takım sorunları çözüp, işi geçiştiriyoruz. Çoğu zaman ne yazık ki çözümün temelinde yatan konseptleri anlamadan...

Eğri oturalım, doğru konuşalım. Programcılık bilim ve mühendislikle alakalı bir meslek. Bu işin temelindeki bilimi kavrayabilmek için bu işin eğitimini almak gerekiyor. Ben üniversitedeki eğitimden bahsetmiyorum. O da önemli, lakin üniversiteden sonra bu bilim dalında kendimizi ne kadar ve nasıl geliştirdiğimiz çok daha önemli.

Programcılık bir ömür boyu öğrenim görmeyi gerektiren bir meslek. Bu eğitimi bize başkaları veremez. Kendi, kendimizi eğitmemiz gerekiyor. Bunun en başlıca yolu da bol, bol kitap okumaktan geçer.

Günümüzde programcı olarak bizim eğitimimize en büyük sekteyi yazılım çatıları (framework) vuruyorlar, çünkü sundukları imkanlarla çalışma sahalarındaki zorlukları ve bilgileri maskeliyorlar. Bu çatıları kullanmak programcı olarak bizim işimizi kolaylaştırmakla birlikte, temelde ne olup, bittiğini tam olarak anlamadığımız sürece yeni bilgi edinmek güçleşiyor. Bizim ana sorumluluğumuz sorun çözmek değil, sorunları çözebilmek için bilgi ve bilim taşıyıcısı olmaktır. İnternetten birkaç satırlık kodu alıp, sorun çözmek matah değildir. Asıl önemli olan kısa yolu değil, uzun ve acılı yolu seçip, belli bir öğrenim sürecinden sonra gerekli temel prensipleri kavramaktır.

Kendimizi ciddi anlamda geliştirmek istiyorsak, ilgi duyduğumuz alanlarda elimize geçen her türlü yazılı kaynağı sistematik bir şekilde tüketmeliyiz. İçimizde işin temellerine indiğimiz hissi uyanmadıysa, yılmadan daha çok kaynak kitap tükerek, bu hisse sahip olana kadar devam etmeliyiz.

Yılmamalıyız! İlim ve bilim, ilim ve bilim peşinde koşturanındır.

En son okuduğunuz üç kitabı yorumlarda görmek ümidiyle...

Agile Türleri

http://www.kurumsaljava.com/2014/01/03/agile-turleri/

Coca Cola'nın kaç türü var, bilirsiniz... Cola light, Cola zero, Cola classic.... Çevik süreçler için de aynı şey geçerli. Ben çevik süreçleri agile zero, agile light ve hardcore agile ya da classic agile olarak üç bölüme ayırıyorum.

AGİLE ZERO

Çalışma ortamında çevikliğe dair hiçbir ibare yoktur.

Belirtileri

- Projede hiçbir birim, entegrasyon ya da onay/kabul testi yoktur.
- Proje kesinlikle zamanında yetişmez.
- Yeni müşteri gereksinimlerinin uygulamaya eklenmeleri çok zaman alır, çünkü uygulama mimarisi esnek ve değiştirilebilir yapıda değildir. Öyle olsa bile testlerin olmaması, kodun ve mimarinin yeniden yapılandırılması engeller.
- Uygulamada çok bug vardır ve çoğu keşfedilmemiştir.
- Uygulama elden test edildiği için hem çok zaman kaybedilir hem de test geniş kapsamlı yapılamaz.
- Entegrasyon çok zaman alır ya da yer yer mümkün değildir, çünkü uygulama sürekli entegre edilmemiştir.
- Sürüm belli bir yazılımcının bilgisayarında alınır. O yazılımcı tatile gittiğinde başka bir kurban yazılımcı seçilir. Buna kısaca releaser hopping ismini veriyorum.
- Müşteriye belirli aralıklarla çalışan bir prototip sunulamaz, çünkü uygulama entegre edilemediğinden çalışır halde değildir.
- Yazılımcılar stres altındadır ve fazla mesai yapmaya zorlanırlar.
- Her şey önceden planlanmaya çalışılır. Buna müşteri isteklerinin tam teşekküllü yazılım öncesi tespiti de dahildir.
- Yazılım öncesinde uygulama mimarisi bir mimar tarafından tespit edilmiştir. Yazılımcılar bu mimaride öngörülen şartlara uymak zorundadırlar. Yeni müşteri istekleri ile mimarinin adapte edilmesi söz konusu iken, kod birim testleri eksikliğinden dolayı yeniden yapılandırılamaz. Böylece uygulama mimarisi yeni müşteri isteklerini taşıyacak şekilde adapte edilemez. Zaten mimar böyle bir şeye içgüdüsel olarak karşıdır. Kısaca yazılımcının uygulama mimarisini değiştirme konusunda inisiyatifi yoktur.
- Müşteri için piyasadaki rekabet şartlarının değişmiş olabileceği bilindiği halde, sürüm zamanları uzun tutulur. Bu uzun bir zaman diliminden sonra müşteriye sunulan sürümün müşterinin işine yaramayacağı anlamına gelebilir. Kısa zamanlı sürümlerle müşterinin fikri alınmış olsa idi, bu sorunun önüne geçilebilirdi.
- Son kullanıcılar tonlarca bug bulup, yazılımcıların uzun bir süre bu bugları temizlemek için uğraş vermelerine sebep olurlar. Böylece yeni müşteri gereksinimlerinin implementasyonu geriye atılır.

AGİLE LİGHT

Scrum gibi bir çevik süreç kullanılıyordur.

Belirtileri

- Ekip sadece Scrum kullanarak gerçekten çevik olduğunu düşünür.
- Projede hiçbir birim ve entegrasyon testi olmayabilir.
- Çeviklik sadece proje yönetiminden (sprint planlaması ve uygulaması) ibarettir.
- Sprint sonunda müşteriye çalışır bir sürüm/prototip sunulabilir.
- Yeni müşteri gereksinimlerinin uygulamaya entegrasyonu çok zaman alır, çünkü test güdümlü yazılım ya da eşli programlama metotları uygulanmaz. Uygulamaya yeni gereksinimlerin eklenebilmesi için uygulamanın yeniden yapılandırmaya ihtiyacı olabilir. Test güdümlü çalışma neticesi olarak geniş kapsamlı birim test seti olmadığı taktirde, uygulamayı yeniden yapılandırmak harikiridir.
- Ekip yazılımcılar ve testçiler olarak iki guruba ayrılır.
- Uygulamayı test etmek için geniş çaplı onay/kabul test setleri hazırlanır. Onay/Kabul testlerini hazırlayan testçi ekibidir. Onay/kabul testleri yazılımcılar tarafından geliştirilen birim testlerin yerine hem geçemezler hem de yazılımcılar tarafından kodu yeniden yapılandırmak (refactoring) için kullanılamazlar, çünkü koşturulmaları saatler alır.
- Uygulama bir sürekli entegrasyon sunucusu kullanılarak entegre ediliyor olabilir.
- Not agile'de uygulama mimarisi için söylediklerim agile light için de geçerlidir.

AGİLE CLASSİC

Ekip katıksız olarak çevik süreç metotlarını uygular.

Belirtileri

- Ekibin parçası olan her yazılımcı test güdümlü yazılım yaparak geniş kapsamlı birim testi seti oluşturur. Test güdümlü çalışma zorunluluğu yoktur. Önemli olan testlerin gerçekleri yansıtacak şekilde kodu kapsamalarıdır (code coverage).
- Proje yönetimi Scrum'da olduğu gibi iterasyon bazlı yapılır. İterasyon uzunluğu 2-4 hafta arasıdır.
- Jenkins ya da Bamboo gibi bir sürekli entegrasyon sunucu kullanılarak kod devamlı entegre edilir.
- Kodu yaptığı değişiklikle kıran yazılımcının masasına bunu gösteren bir oyuncak ayı bırakılır. Bu oyuncak ayı kodu kıran bir yazılımcıdan diğerine el değiştirir. Her yazılımcı kodu kullandığı versiyon kontrol sistemine eklemeden önce (commit) tüm birim testlerini koşturarak, kodun içinde bulunduğu durumu kontrol eder. Çalışmayan ya da kırık kod eklenmez.
- Yeni müşteri istekleri zorluk çekilmeden uygulamaya dahil edilir, çünkü testler sayesinde kodun yeniden yapılandırılması (refactoring) kolaydır.
- İterasyon sonunda müşteriye çalışan bir prototip sunulur. Müşteri isterse bu prototipi aktif olarak günlük işinde kullanmaya başlar.
- Yazılımcının sorularını cevaplamak için ya müşteri ekibin yakınlarındadır ya da müşteriyi temsil eden bir vekil vardır. Yazılımcı sorularını doğrudan müşteriye ya da vekiline yöneltir. Müşteri ile yazılımcı arasına analist, proje yöneticisi, firma sahibi ya da başka bir şahıs giremez.

- Ekipte Scrum Master ya da Product Owner gibi roller yoktur. Ekibin dadıya ihtiyacı yoktur. Ekip içindeki yazılımcılar her şeyden sorumludur.
- Ekip uygulamayı müşteri isteklerinin öncelik sırasına göre geliştirir. Uygulamaya hangi özelliğin ekleneceğini her zaman müşteri belirler.
- Yazılımcılar mesailerini sekiz saat ile sınırlı tutarlar. Daha fazlasına gerek yoktur, çünkü yazılım geliştirme süreci planlandığı şekilde ilerler.
- Her iterasyon sonunda yazılımcılar bir araya gelerek, geçmiş iterasyonu analiz ederler (retrospective). Her yazılımcı düşündüğü artı ve eksiler hakkında düşünce beyan eder. Maksat başkası hakkında olumsuz fikir beyan etmek ya da sorumlu bulmak değildir. Bu da bir geri bildirim türü olduğu için ekip içinde bulunduğu durumu daha iyi kavrar ve tespit edilen olumsuzlukların bir sonraki iterasyonda meydana gelmelerini engeller.
- Uygulama mimarisi yazılım sistemi ile geliştirilir. Her yazılımcı inisiyatif kullanarak, uygulama mimarisini güncel implemente edilen müşteri gereksimini taşıyacak şekilde adapte edebilir.

Ne zaman çevik oluruz? Kodu hamur gibi yoğurabildiğimizde!

Yazılımcıların Gerçek Ekmek Teknesi

http://www.mikrodevre.com/2014/08/14/yazilimcilarin-gercek-ekmek-teknesi/

Yazılımcılar kodu müşteri için yazıyoruz derler ya, bu doğru değil! Kod her zaman bir merkezi işlem birimi (CPU – Central Processing Unit) namı diğer mikroişlemci için yazılır. Mikroişlemci olmadan kod çalışmaz. Bu yüzden ilk etapta programcı olarak ekmek teknemiz mikroişlemcidir. Sanal makinelerde (VM – Virtual Machine) çalışan kodlar için bu geçerli değil diyebilirsiniz, çünkü sanal makine bir mikroişlemci değildir. Doğru, lakin sanal makineler de kodu mikroişlemcilerin anlayacağı şekle dönüştürürler. Java dünyasında örneğin bunu yapan JVM JIT Hotspot derleyicisidir. Kod eninde sonunda mikroişlemcinin hafıza alanlarından (register) birisine düşer ve işlem görür.

Modern bir mikroişlemcinin nasıl çalıştığını bir büyüteç altında inceleseydik, ne olup, bittiğini anlayamazdık, çünkü birkaç santimetre karelik bir kutucuk içinde kalbi saniyede birkaç milyar kez atan birden fazla canavar yatmakta.

Ana işimiz CPU için kod yazmak, ama nasıl çalıştığından bihaberiz. Modern mikroişlemcileri çok yüksek bir perspektiften anlatan birkaç kitap sayfası karıştırıp, mikroişlemcilerin nasıl çalıştıklarını anladığımızı düşünüyoruz. Eğer mikroişlemcilerin gerçek anlamda nasıl çalıştıklarını anlasaydık, yazılımcı olarak en büyük problemimiz paralel program yazmada dünya şampiyonu olurduk. Paralel program yazmayı bile doğru, dürüst beceremeyen, becerdiğini zannedip saniyede milyar kalp atışını musluktan akan su damlası misali frenleyip, paralel program yazdığını düşünen bizlerin, bu canavarı nasıl ehlileştirebileceğimiz bile aklına gelmiyor. Ama biz hep böyle değildik. Sonradan böyle olduk. Bizi terbiye edip, bu hale getirdiler. Kullandıkları tek argüman da, programcının mutlaka yüksek dilleri kullanarak, daha soyut düşünmeye çalışmak zorunda olması gerektiğiymiş. Her yeni yüksek dil ile mikroişlemciden bir adım daha uzaklaştık ve etrafımız günümüzde bir derleyicinin bile ne yaptığını anlamayan programcılarla doldu. Bunun en güzel örneğini kendisine full stack J2EE/JEE developer ismi veren programcılar teşkil etmekte.

Programcı olarak bir mikroişlemcinin nasıl çalışması gerektiğini bilmek zorunda değilim diyebilirsiniz. Sonuç itibariyle bir arabayı kullanırken, motorun da nasıl çalıştığını bilmiyorum ve bilmek zorunda değilim, öyle değil mi? Lakin arada çok ince bir fark var. Arabayı kullandığım sürece nasıl çalıştığını bilmek zorunda değilim. Lakin araba için bir şeyler üretiyorsam, o zaman arabanın nasıl çalıştığını tam olarak bilmem gerekir. Aynı şey yazdığım kodlar için de geçerli. Programcı olarak mikroişlemci için kod yazıyorsam, mikroişlemcinin nasıl çalıştığını tam olarak bilmem gerekiyor. Yüksek dillerde yazılan kodun %99 u sorunsuz herhangi bir mikroişlemci üzerinde çalışırken, %1 lik bölümü çalışmadığında, programcıların %99 u afallayıp, kalırlar, çünkü mikroişlemci bünyesinde olup, bitenlerden bihaberdirler. Bu işin gerçek ustaları bu yüzden bu kadar azdırlar, çünkü bu %1 kesimin içindedirler.

Buradan tek bir sonuç çıkarabiliriz: Yazılımcıların donanıma ve özellikle kodlarının üzerinde çalıştığı mikroişlemcilere hükmetmeleri gerekiyor. İyi bir yazılımcı olmak için bu bir şart. Yüksek seviyede soyutlama yetisini bu detay bilgisi ile kombine ettiğimizde, her yönüyle işine hakim yazılımcılar haline gelebiliriz.

Günümüzde 64 bit genişliğinde ve birden fazla çekirdeğe sahip mikroişlemciler kullanılmakta. Bu donanım mühendislerinin çabalarıyla gelinen bir nokta. Onlar da 4 ve 8 bit genişliğindeki mikroişlemcilerle başladılar. Elli sene önce ortaya çıkan bu mikroişlemciler günümüzün modern mikroişlemcileri ile hala birçok ortak özelliğe sahipler. Prensipte bir mikroişlemcinin çalışma tarzı elli sene öncesine kıyasla fazla değişmedi.

64 bitlik bir mikroişlemciyi tüm özellikleri ile anlamak kolay olmayabilir. Lakin 8 bitlik bir mikroişlemcinin nasıl çalıştığını görerek, bilgi seviyemizi 64 bite doğru yükseltebiliriz.

8 bitlik bir mikroişlemcinin nasıl çalıştığını tam anlamıyla kavrayabilmek için 8 bitlik bir mikroişlemciyi kendi çabalarımla oluşturmamız faydalı olacaktır. Böyle bir mikroişlemciyi TTL bazlı mantık kapıları ihtiva eden entegre devrelerle yapabiliriz.

Mikrodevre.com bünyesinde 8 bitlik bir mikroişlemcinin bu entegre devreler yardımı ile nasıl oluşturulabileceğini önümüzdeki blog yazılarım ve videolog kayıtlarımla göstermek istiyorum.

8 bitlik bir mikroişlemci oluşturma fikri sadece dijital elektronikle sınırlı değil. Eğer öyle olsaydı, büyük bir ihtimalle bir yazılımcı olarak bu konuya bu kadar derin bir ilgi duymazdım. Bir mikroişlemcinin anlamlı bir işlem yapabilmesi için bu mikroişlemci için program yazmak gerekir. Böyle bir projeyi benim için ilginç kılan, donamımla yazılımın bu kesişme noktası. Bu aslında bize yolculuğumuzun entegre devreler yardımı ile bir mikroişlemci oluşturmakla son bulmadığının müjdesini vermekte. Yolculuğumuz mikroişlemcimiz için bir komut kümesi oluşturup, bu komut kümesi yardımı ile assembler seviyesinde programlar yazarak devam edecek. Bu amaçla kendi derleyicimizi oluşturacağız.

Belki oluşturduğumuz mikroişlemci ilkel bir hesap makinesinden ileri gidemeyecek. İşletim sistemi kurulumu gibi daha sofistike işlemler için 16 bitlik bir mikroişlemciye ihtiyaç duyacağız. Lakin 8 bitlik bir mikroişlemci oluşturma projesi bize analog elektronik, dijital elektronik, mikroişlemci mimarileri ve kod derleyicileri alanlarında ihtisas yapma fırsatı verecektir. Çalışmalarımızın tatmin edici neticeler vermesi ümidiyle.

Kod A.Ş. – Kod Anonim Şirketi

http://www.kurumsaljava.com/2012/07/20/kod-a-s-kod-anonim-sirketi/

Programcılar tarafından yazılan metotların ve kullanılan değişken isimlerinin çoğu anonim, yani adı sanı belirsiz. Kullanılan isimler ilk bakışta kod bünyesinde olup bitenleri ifade gücünden aciz. Bu kodun okunabilirliğini düşüren bir faktör. Zamanımızın büyük bir kısmını kod okuyarak geçirdiğimizi düşündüğümüzde, seçilen isimlerin ne kadar önemli olduğunu ortaya çıkmakta.

Anonim kod sadece metot ve değişken isimleri ile sınırlı değil. Dün katıldığım bir kod inceleme oturumunda (peer code review) kodun büyük bir kısmını oluşturan koşullu mantık yapılarının (if/else) da anonimlikten nasiplerini aldıklarına tekrar şahit oldum. Oturum esnasında aşağidaki tarzda bir if yapısa denk gelince, burada ne tür bir işlemin yapıldığını sormadan edemedim, çünkü birkaç sefer okumama rağmen, kodun ne yaptığını anlayamadım.

```
// bla bla bla bla
if(pos.length() > 0 && indexAccess.isLucene() && blabla && blabla .....){
}
```

Sağolsun çalışma arkadaşım kodun ne yaptığını bana açıkladı. Ben ne zaman bu kod parçasını okumak istesem, yanımda tercüman olarak çalışma arkadaşımı da bulundurmam gerekiyor. Ne kadar pratik degil mi!

Bu kod parçasında gözüme çarpan iki sorun var. Birincisi if bünyesinde olup bitenleri anlamak hemen hemen imkansız. İkincisi kodun ne yaptığını açıklamak için //bla bla seklinde koda yorum eklenmiş. Bu kadar zaman harcayip, koda yorum eklemek yerine, if bünyesindeki kodu bir metoda dünüştürüp, bu metoda anlamlı bir isim versek nasıl olurdu?

```
if(searchable()){
}

private boolean searchable(){
    return pos.length() > 0 && indexAccess.isLucene() && blabla && blabla .....
}
```

Şimdi if bünyesin olup bitenleri anlamak benim için daha kolaylaştı. Anonim olan if içindeki kodu bir metoda dönüştürerek, ona bir kişilik kazandırdım. Bu metot ismini gören programcı metot bünyesinde olup bitenleri aşağı yukarı kestirebilir. Bunun için kodu yazan programcıyı tercüman olarak kullanmak zorunda değil.

Eğer anonim kodu nasıl anlarım derseniz, kod içinde bırakılan açıklayıcı yorumların izini sürün derim. Eğer bir değişken ismini ya da if bünyesinde olup bitenleri açıklayıcı bir yorum bırakılmışsa, o zaman o kod anonim koddur.

Buraya kadar olan kodun teknik olarak nasıl daha okunabilir hale getirilebilir olması ile ilgiliydi. Şimdi gelelim programcının neden anonim yapıları tercih ettiğine. Bunu öğrenmek için programcıya şu soruyu sordum: “Bana if bünyesinde olup bitenleri ifade edebilecek güçte bir metot ismi söyleyebilir misin?“. 10 saniye… 20 saniye… 30 saniye geçti ve programcı hiç te fena

olmayan bir öneride bulundu. Ama akabinde yorumu şu oldu: "Bu ismi düşünene kadar ben if için gerekli kodu yazmıştım bile!". Ama marifet kod yazmak değil ki canım kardeşim. Marifet okunabilen kod yazmaktır.

Programcılar bu tarz kod yazmaya alışmışlar ve farkında olmadan her gün anonim, yani okunması zor kod üretiyorlar. Bunu değiştirmek için tavsiyem programcının kod kataları yapmasıdır. Kod Kata ve Pratik Yapmanın Önemi başlıklı yazımda pratik yapmanın programcı için ne kadar önemli olduğunun altını çizmiştim. Yaptığım kataları programcılarla paylaşmak için KodKata.com isminde bir websayfa hazırladım. Vakit bulduğunuzda bir göz atmanızı tavsiye ederim. Bu konudaki geri bildirimleriniz beni sevindirecektir.

Kod A.Ş. iflasa mahkum. Bu batan gemiden ayrılmanın zamanıdır.

Sorumluluk Sahibi Olmak

Yazılım yapmayı zorlaştıran her zaman kod birimleri arasındaki bağimlılıklar ve bu bağımlılıkların yönetimi olmuştur. Bu bağımlılıkları tamamen yok etmek yazılım sistemini anlamsız kılarken, kontrolden çıkmalarına göz yummak yazılım sisteminin ölüm fermanı olabilir. Yazılım mühendisi bunu bilir ve gerekli gördüğü yerlerde DIP, ISP ve SRP gibi tasarım prensiplerini kullanarak kodu dokur.

Yazılımcının kod yazarken devamlı uygulaması gerektiği bir tasarım prensibi varsa, bu da SRP (Single Responsibility Principle) tek sorumluluk prensibidir. Bu prensibe göre her kod biriminin sadece ve sadece bir sorumluluk alanı, yani yaptığı tek bir iş olmalıdır. Bu kod birimi bir paket (package), bir sınıf, metot ya da bir değişken olabilir. SRP uygulanmadığı taktirde yazılım sistemi kontrol edilemez, kırılgan bir bağımlılıklar yumağı haline gelebilir.

Genelde bir bakışta bir kod biriminin SRP uyumlu olup, olmadığını anlamak zor değildir. Eğer bir sınıf iki bin satırdan oluşuyorsa, bu sınıfın birden fazla işle meşgul olduğu söylenebilir, aksi taktirde bu kadar büyümesi mümkün olmazdı.

Büyük yazılım sistemlerini sınıf sınıf gezip, kim SRP uyumlu, kim değil diye kontrol etmek mümkün değil. Yazılımcı olarak daha ziyade anlık resmi görmemizi sağlayacak bir araç oluşturmamız lazım. Bu aracı geliştirmeden önce, sınıfların SRP uyumluluklarını ölçmek için bir yönteme ihtiyacımız var. Bu yöntem örneğin bir sınıfın versiyon kontrol sistemi bünyesinde ne kadar değişikliğe uğradığını ölçebilir. Eğer bir sınıf sıkça değişikliğe uğruyorsa, bu bu sınıfın birden fazla sorumluluk sahibi olduğu anlamına gelebilir. Gerçekten de binlerce satırdan oluşan kod birimlerinin versiyon kontrol sistemindeki geçmişleri incelendiğinde, çok sık değişikliğe ugradıklarını görmek mümkündür. Değişik sorumluk sahibi bir sınıf, her bir sorumluluk için değişikliğe maruz kalabileceğinden, bu gibi sınıfların sicilleri kabarıktır. O zaman bu sınıfları, kaç satır ihtiva ettiklerini ve kaç sefer değişikliğe uğradıklarını tespit edebilecek bir uygulama geliştirelim. Böyle bir uygulamanın kodu aşağıda yer almaktadır.

```
import java.io.BufferedReader;
import java.io.DataInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.HashMap;
import java.util.Map;

public final class SvnHistory {

    private static final String PROJECT_ROOT = "C:/xxx";
    private static Map<String, ClassStats> STATS_MAP =
            new HashMap<String, ClassStats>();
    private static StringBuilder STATS = new StringBuilder();
    private static final String SVN_EXE = "C:/Program
                Files/SlikSvn/bin/svn.exe";
    private static final String SVN_LOG_COMMAND = "log";

    public static void main(final String[] args) {
```

```
        makeHistory(new File(PROJECT_ROOT));
        makeHeader();
        makeLines();
        printStats();
    }

    private static void makeHeader() {
        STATS.append("Filename;revisionCount;loc").append("\n");
    }

    private static void makeLines() {
        for (final String key : STATS_MAP.keySet()) {
            makeLine(key, STATS_MAP.get(key));
        }
    }

    private static void printStats() {
        System.out.println(STATS.toString());
    }

    private static void makeLine(final String key, final
        ClassStats s) {
        STATS.append(key).append(";").append(
            s.revCount).append(";")
                .append(s.loc).append("\n");
    }

    private static void makeHistory(final File root) {
        for (final File file : root.listFiles()) {
            if (isJavaFile(file)) {
                history(file);
            }
            if (file.isDirectory()) {
                makeHistory(file);
            }
        }
    }

    private static boolean isJavaFile(final File file) {
        return file.getName().endsWith(".java");
    }

    private static void history(final File file) {
        try {
            final int revCounter = getRevisionCount(file);
            STATS_MAP.put(file.getName(), ClassStats.make(
            file.getName(), revCounter,
                    getLinesOfCode(file)));
        } catch (final Exception err) {
            err.printStackTrace();
        }
    }

    private static int getRevisionCount(final File file)
                throws IOException,
            InterruptedException {
        String line = null;
```

```
        String line = null;
        int revCounter = 0;
        final Process p = Runtime.getRuntime().exec(
                SVN_EXE + " " + SVN_LOG_COMMAND + " " +
                     file.getAbsolutePath());
        final BufferedReader bri = new BufferedReader(
                new InputStreamReader(
                p.getInputStream()));
        final BufferedReader bre = new BufferedReader(
                new InputStreamReader(
                p.getErrorStream()));
        while ((line = bri.readLine()) != null) {
            if (line.startsWith("r"))
                revCounter++;
        }
        bri.close();
        bre.close();
        p.waitFor();
        return revCounter;
    }

    private static int getLinesOfCode(final File file) {
        int loc = 0;
        try {
            final FileInputStream fstream = new FileInputStream(file);
            final DataInputStream in = new DataInputStream(fstream);
            final BufferedReader br = new BufferedReader(
            new InputStreamReader(in));
            while (br.readLine() != null) {
                loc++;
            }
            in.close();
        } catch (final Exception e) {
            System.err.println("Error: " + e.getMessage());
        }
        return loc;
    }

    private static class ClassStats {
        private String fileName;
        private int revCount;
        private int loc;

        private ClassStats() {

        }

        public static ClassStats make(final String name,
            final int revCounter, final int loc) {
            final ClassStats classStats = new ClassStats();
            classStats.fileName = name;
            classStats.revCount = revCounter;
            classStats.loc = loc;
            return classStats;
        }
    }
}
```

Bu uygulama kodun Subversion tabanlı bir versiyon kontrol sisteminde tutulduğunu varsaymaktadır. PROJECT_ROOT değişkeni versiyon kontrol sisteminden edinilmiş projenin (working copy) ana dizinine işaret ediyor. Uygulama ana dizinde bulunan tüm dizinleri taradıktan sonra bulduğu her Java dosyası için versiyon kontrol sicilini kontrol ediyor ve dosya üzerinde yapılan değişiklik adedini tespit ediyor. Akabinde her Java dosyanın satır adedi tespit edildikten sonra, aşağıdaki şekilde bir ekran çıktısı oluşturuluyor:

```
Filename;revisionCount;loc
AAA.java;13;55
BBB.java;21;325
CCCjava;3;292
DDD.java;2;367
EEE.java;6;384
FFF.java;1;156
GGG.java;1;77
HHH.java;11;47
III.java;10;81
JJJ.java;3;132
KKK.java;1;111
LLL.java;9;29
MMM.java;32;433
NNN.java;45;1301
OOO.java;3;88
PPP.java;11;47
QQQ.java;60;212
```

Ekran çıktısı bir csv (comma seperated value) dosyası. Bu dosyayı örneğin Excel ile import edip, aşağıdaki şekilde bir diagram oluşturmak mümkün.

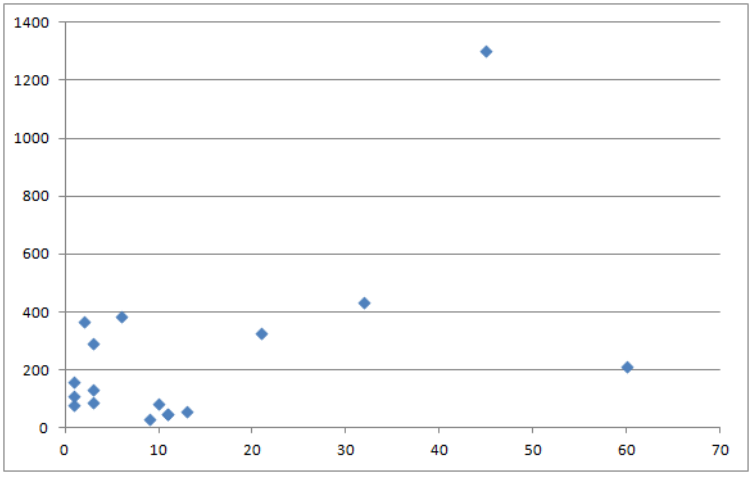


Diagram üzerindeki her nokta bir Java dosyasına işaret etmektedir. X ekseninde her dosyanın maruz kaldığı değişiklik adedi (revision count), Y ekseninde dosyanın sahip olduğu satır adedi yer almaktadır. Bu diagrama bakıldığında tek sorumluluk prensibine ters düşen kod birimleri

hangileridir?

SRP uyumlu olmayan sınıflar için sağ üst alana bakmak yeterli. Bu sınıfları lokalize ettikten sonra tek bir iş yapacak hale getirmek biz yazılımcıların asli görevlerinden birisi. İzcilerden kendimize örnek alalım :)

Acı Çekmeden Üstad Olunmaz

http://www.kurumsaljava.com/2012/08/12/aci-cekmeden-ustad-olunmaz/

Her sektörün kendi guru, üstad, pir olarak görülen zatları var. Yazılım sektöründe de durum farklı değil. Şöyle başımızı kaldırıp devleşmiş bir üstada baktığımızda, bu adam nasıl bu kadar yetenek, kabiliyet, bilgi ve beceri sahibi olabildi diye bir soru aklımıza gelir. Aynı zamanda kendimizin bu devin yanında ne kadar küçük kaldığımızı görür, saklanacak bir yer aramamıza bile gerek kalmadan onun yanında kaybolup gideriz. Örnek mi ver diyorsunuz: Robert C. Martin, Kent Beck, Peter Norvig. Ne milletten oldukları önemli değil. Bu listede birgün mutlaka Türk yazılımcılarının da ismi yer alacak. Önemli olan nasıl üstadlaşabildikleri.

Bir yazılımcı üstad olarak mı doğdular? Yoksa ultra zekiler mi? Yaratan onlara daha mı özen gösterdi? Eğer öyle ise, neden ben ya da siz değil de, onlar? Bu soruların cevabı düşündüğümüzden daha kolay. Sizden ve benden bir farkları yok. Aynı fizyolojik özelliklere sahibiz. Onları farklı kılan bir sey var: üstad olmanın bedelini ödediler. Bu bedeli nasıl ödediklerini inceleyelim.

Noel Tichy'nin kişisel gelişim için tanımladığı üçlü çemberi tanıyor musunuz?

Noel Tichy'ye göre kişisel gelişimin sağlandığı saha Learning Zone olarak isimlendirdiği ikinci çember. Bu çember bireyin yetenek, bilge ve becerilerinin yetmediği alanı tanımlıyor. Birey bu sahaya geldiğinde, daha önce tanımadığı şeylerle karşılaştığı için bunları öğrenmek zorunda kalıyor. Comfort zone olarak tanımlanan en içteki cember bireyin kendisini geliştirmesinin mümkün olmadığı saha, çünkü birey bu sahada hakim olduğu aktiviteleri tekrarlıyor. Bildiği şeyleri tekrarlaması ona bir şey katmıyor. En dışta bulunan çember, bireyin olup bitene anlam veremediği, gidişatı kavrayamadığı, kısaca karşılaştığı durumun onu panik ettiği sahayı tanımlıyor.

Kendimizi geliştirebilmemiz için Comfort Zone'dan çıkıp, Learning Zone'a girmemiz gerekiyor.

Bu ne yazık ki o kadar da kolay bir şey değil. Birincisi Comfort Zone'dan çıkmak demek, bilinmeyene doğru gitmek demek. Buna birey içgüdüsel olarak karşı koyabilir, çünkü bu geçiş bireyin içinde huzursuzluk ve tedirginlik yaratabilir. Ayrıca bu geçis bireyi akli ve ruhi açıdan çok zorlayabilir. İkincisi Learning Zone'un yeri devamlı değişiyor. Ne zaman Learning Zone'da, ne zaman Panic Zone'da olunduğunu kestirmek kolay olmayabiliyor.

Bireyin kendisini devamlı zorlayarak Learning Zone içinde kalmaya çalışması kısaca acı çekmesi anlamına gelmektedir. Kendi kişisel gelişimi için icat ettiği tüm aktiviteleri Learning Zone'da kalacak şekilde şekillendirmesi gerekmektedir. Bu bireyi zihinsel ve ruhsal olarak çok yorar. Devamlı sınırları zorlamak kolay değildir. Ama sınırları devamlı zorlamadan kişisel gelişimde yol kat etmek mümkün değildir.

Beşer ile üstad arasındaki fark burada yatmaktadır. Müstakbel üstat emin adımlarla her daim Learning Zone ve Panic Zone'da cirit atarken ve bunu yıllar boyunca hiç yılmadan yaparken, beşer yer, içer, yatar ve üstadlara hayran olur, ama kendisinin bu yüksekliklere ulaşmasının imkansız olduğunu düşünür, çünkü ya Learning Zone'un varlığından bihardir ya da sınırları devamlı zorlayacak cesareti yoktur. Beşer olarak hayatını sürdürür, üstad gibi ölümsüzleşmenin sadece rüyasını görebilir. Ama bir üstad olmak düşündüğü kadar zor değildir.

Bir örnek üzerinde bir programcının Learning Zone'a nasıl geçip, orada kalabileceğine değinmek istiyorum. Yazılımda usta ya da üstad olmanın bir yolu da iş haricinde pratik yapmaktan geçer. Bunun en iyi örneğini kod kataları teşkil eder. Programcı bir katayı ilk defa uyguladığında hemen Learning Zone'a geçer. Katayı her tekrarlayışında tekrar tekrar Learning Zone'a geçtiğine şahit olur, çünkü her tekrar ona yeni bir şeyler öğretir. Belli bir zaman sonra programcı öğrenebileceği birçok şeyi öğrenir ve içinde bulundugu Learning Zone onun için Comfort Zone haline gelir. Artık Learning Zone'un yeri değişmiştir. Programcının katayı harfiyen tekrar tekrar uygulaması ona yeni bir şeyler katmaz. Artık tekrar katayı değiştirerek yeniden Learning Zone'a geçmesi gerekmektedir, aksi taktirde olduğu yerde saymaya başlayacaktır. Programcı örneğin katayı başka bir programlama paradigması ile yeniden şekillendirebilir. Bu programcıyı tekrar Learning Zone'a katapult eder. Programcı yepyeni bir dünyayı keşfetmeye ve kişisel sınırlarını zorlayıp, aşmaya başlar.

Kendimden bir örnek vereyim isterseniz. Bu satırları yazarken Learning Zone içindeyim, çünkü düşüncelerimi ifade edebilmek için muazzam zorlanıyorum. Bunu hissediyorum. İçimde devamlı bir güç yazıyı yarım bırakarak yazmamı engellemeye çalışıyor, çünkü gidişattan hoşnut değil. Tanımadığı bir şeyler olup bittiği için zorlanıyor ve beni yazmaktan alıkoymaya çalışıyor. Bu güç benim Comfort Zone'da geride bıraktığım yazı yazma yetilerim. Onları geliştirmek için bu ve bunun gibi bir milyon yetmiş altı bin sekiz yüz doksan yedi yazı daha yazmam gerekiyor. İşte bu güç bunun bilincinde olduğu için, neden acı çekip Learning Zone ya da Panic Zone'da kalmak istiyorsun diyor ve beni devamlı Comfort Zona'a geri çekmeye çalışıyor.

Üstad olarak gördüğümüz her şahıs bu hale gelebilmek için yeterince acı çekip, bedelini ödedi. Bu acılar sadece bir iki gün, bir iki hafta, bir iki ay ya da bir iki yıl değil, onlarca yıl çekildi. Peter Norvig "Kendine on senede programlamayı öğret başlıklı yazısında her hangi bir dalda ustalaşmanın on sene gerektirdiğini söylüyor. Altını çizdiği ve bu yazının konusu olan bir şey daha söylüyor:

"The key is deliberative practice: not just doing it again and again, but challenging yourself with a task that is just beyond your current ability, trying it, analyzing your performance while and after doing it, and correcting any mistakes. Then repeat. And repeat again."

Kabaca tercüme edecek olursak: "Tartarak pratık yapmak işin sırrı: sadece tekrarlamak değil, mevcut sınırları zorlayacak bir aktivite seçip denemek, analiz etmek ve hataları düzeltmek ve akabinde tekrar etmek ve tekrar etmek."

Üstat olmak imkansız değil. Bunun için yüksek IQ ya da yetenek gerekli değil. Ama acı çekip bedelini ödemeden kimse olamaz.

EOF (End Of Fun) Özcan Acar

Çevikliğin Böylesi

http://www.kurumsaljava.com/2012/04/05/cevikligin-boylesi/

Son zamanlarda yazılımla yakından ya da uzaktan ilişkisi olan herkesin ağzında olan kelime; çeviklikten bahsediyorum. İngilizce de agile, lean gibi kavramlar kullanılıyor ve artık her şey için kullanılmaya başlandı. Gören de zannederki artık her proje çevik yazılım yöntemleri ile yapılıyor, her şey yolunda.

2010 senesinde Londra'da yapılan Domain Driven Design exChange konferansında Eric Evans, yazılımda çevikliğin anlamının kaybolduğunu çünkü artık her şeyin çevik olarak isimlendirildiğini söylemişti. Ne kadar haklı. Aşağıdaki fotografı bir trende çektim. Artık iş verenler bile çevik kelimesini kullanarak yazılımcıları oltaya düşürmeye çalışıyor. Çevik bir fare, çevik bir klavye belki de çevik bulut (agile cloud) bile kullanıyorken bulabiliriz kendimizi yakında. Yaşasın hayatın her safhasında karşılaştığımız çeviklik.

Son günlerin en moda çevik terimlerinden birisi Scrum. Avrupa'da artık birçok proje Scrum ile yapılıyor. Proje yöneticileri Scrum'la oturuyor, Scrum'la kalkıyor. Scrum yazılımda yeni bir dönemin başlangıcı. Diğer çevik süreçler on, on beş yıldır piyasada olmalarına rağmen, Scrum iyi pazarlandığı için çeviklik deyince ilk akla gelen süreç oluyor. Şahsen Scrum yöntemleri beni elektrik gibi çarpmamış olsa bile (çok fazla etkilenmediğim anlamında), Scrum doğru uygulandığında bir projede çevikliğin başlangıcı olabilir. Bugüne kadar birçok Scrum projesinde çalıştım ve doğru düzgün uygulandığını görmedim. Eski proje yöneticileri şimdilerde Scrum Master olmuşlar ve eski yöntemleri ile projeleri sürdürmeye devam ediyorlar. Kafalarda değişen fazla bir şey olmamış. Scrum'ı elbise olarak almışlar, kendi bedenlerine uydurmaya çalışmışlar. Bu bir yere kadar doğru olabilir, ama daha doğrusu Scrum elbisesini alıp, bedeni ona göre uydurmaktır. Scrum harfiyen ne istiyorsa proje ve çalışmalar ona göre şekillendirilmelidir. Ama mevcut kafalar değişmedikçe Scrum ve diğer çevik süreçlerin başarılı olmaları mümkün değil.

Çeviklik, eski kafaların eski yöntemlerle yaptıkları işleri kendi yöneticilerine satmak için kullandıkları bir kavram haline geldi. Scrum projelerinde edindiğim tecrübeleri size aktararak çevikliğin nasıl yarı yolda kaldığını ve gerçek çevikliğin ne anlama geldiğini aktarmaya çalışayım.

Katıldığım ilk Scrum projesini hatırlıyorum. Sıkı bir mülakatın ardından projeye alındım. Mülakat esnasında bana bir sürü Java bulmacası soruldu. Maksat ne kadar derin Java bilgisine sahip olduğunu anlamaktı. Ama test güdümlü yazılım ya da JUnit hakkında bir kelime bile edilmedi. Aynı tarzda mülakatlar ile başka programcılar da projeye dahil edildi. Onlarda da durum farkli değildi. Ben programlarımı test güdümlü yazmaya gayret gösterdim. Ama bazı programcılar kırık kodları versiyon kontrol sistemine ekledikleri için testlerim çoğu zaman çalışmıyordu. Bu beni çok rahatsız eden bir durumdu. Test yazmaya önem vermeyen bu tür programcılarla çok ağız dalaşımız oldu. Sürekli entegrasyon serveri kullanılmıyordu, test güdümlü çalışılmıyordu, müşteri gereksinimleri kullanıcı hikayeleri olarak hazırlanmamıştı. Hangi kullanıcı hikayesini ne kadar zaman diliminde tamamlarız diye tahmin etme oturumları yapılmıyordu. Ama bir Scrum ekibi ve projesiydik. Her sabah on beş dakika asker gibi sıraya girip, Scrum toplantımızı yapardık. Bunun neresi çeviklik, neresi Scrum. Günde on beş dakika toplantı yaparak çevik olunmaz! Zaman kaybından başka bir şey değildir, göz boyamacadır.

Çalıştığım diğer bir Scrum projesinde Scrum'ın doğru adapte edilmesi için gayret gösterilmişti. Bir sürekli entegrasyon serveri ve yazılım metriklerini takip etmek için Sonar sunucusu kullanıyorduk. Zaman yetersizliğinden dolayı programcılar test yazmaya önem vermıyorlardı. Ayrıca uygulama bir uygulama sunucusu içinde çalışmak zorunda olduğu için test yazmak kolay değildi. Mevcut testlerin hepsi entegrasyon testi tarzındaydı. En ufak bir değişiklikte bile yeniden bir EAR paketi oluşturup, tüm uygulamayı uygulama sunucusuna çekmek gerekiyordu. Uygulama sunucusunun ayağa kalkması beş dakika sürüyordu. Yazılım geliştirme tamamen uygulama sunucusu ile iç içe geçmiş durumdaydı. Uygulama sunucusunu kullanmadan yazilim yapmak mümkün değildi. Test eksikliğinden ve uygulamanın bir uygulama sunucusuna çekilmesi gerektiğinden, uygulamayı yeniden yapılandırmak çok zordu. Uygulamayı uygulama sunucusundan bağımsız bir hale getirmek için çaba sarfetmemiz gerektiği konusunda çok dil dökmeme rağmen bu gerçekleşmedi. Sonuç olarak çevik değildik, çevik olamadık. Her yeni bir ekleme ile uygulamayı değiştirmek daha da zora girdi. Eğer programcılar ölmedilerse hala yaşıyorlar.

Çevik Nasıl Olunur?

Çevik olmak öncelikle cesaret ister. Gelenekleri bir kenara itip, yeniden başlamayı gerektirir. Felç geçirmiş bir insanın yeniden konuşmayı ya da yürümeyi öğrenmesi gibi çeviklikte her şeyi unutup, yeniden öğrenmeyi gerektirir. Çevik olmayı zor kılan da budur. İnsanlar alışkanlıklarından kolay kolay vaz geçemezler. Radikal değişimleri sevmezler. Yeniliklere kolay kolay adapte olamazlar. Dünya yerinde dursun isterler. Ama yazılım yapmak gibi dinamik bir ortamda değişmeyen tek şey değişikliğin kendisidir. Bununla yaşamak her babayiğidin harcı değildir. Cesur olanların, yeniliklere açık olanların işidir. Buraya kadar olan çevikliğin edebi tanımıydı. Şimdi gelelim teknik tanımlamasına.

Çevik olmayı ben hamur yoğurma ile kıyaslarım. Hamuru, içinde yeterince sıvı olduğu sürece

yoğurabilirsiniz. Sıvı azaldıkça hamuru yoğurmak zorlaşır. İçinde sıvı kalmamış hamuru yoğuramassınız. Bu analogiden yola çıkarak çevikliğin yazılımdakı tanımlamasını yapalım. Hamur oluşturduğumuz yazılım sistemi, sıvı birim testleri, yoğurma ise yeniden yapılandırmadır (refactoring). Birim testleri olmadan yazılım sistemini yoğuramassınız (yeniden yapılandıramassınız), ama her yeni müşteri gereksinimi ile sizden yazılım sistemini yoğurmanız beklenir. Oluşturduğunuz uygulamaya yeni müşteri gereksinimlerini eklemek, yani uygulamayı yoğurmak zorundasınız. Bu işi yapmak yani programcı olarak çalışmak istiyorsanız başka alternatifiniz yok. Patronunuz ve müşteriniz sizden uygulamayı yoğurmanızı yanı kendi isteklerine cevap verecek şekilde geliştirilmenizi bekler. Birim, entegrasyon ya da onay/kabul testleri yazmıyorsanız uygulamayı yoğururken sıvı buharlaşacak ve siz belli bir noktadan itibaren uygulamayı yoğuramama rizikosuyla karşı karşıya kalacaksınız. Ama sizden yoğurmaya devam etmeniz beklenecektir. Birçok proje ya da geliştirilen uygulama bu sebepten dolayı başarısız olmaktadır. Devam etmek istiyorsanız uygulamaya sıvı katmanız gerekir. Oluşturacağınız yeni testler hamurunuz için yeni sıvı olacaktır. Bu testler hamuru tekrar yumuşatır ve yoğrulmasını kolaylaştırır. Programcı testleri kullanarak uygulamayı yeniden yapılandırabilir (refactoring). Bu hamuru yeniden yoğurabilme yeteneğini kazanmak demektir. Testler yoksa yeniden yapılandıramaz. Yeniden yapılandıramassa uygulama sıvı eksikliğinden katılaşır ve yoğrulamaz hale gelir. Çevikliğin ve esnekliğin gizli anahtarları testler ve yeniden yapılandırmadır. Bu iki anahtarı kullanmayı bilen programcı çeviktir. Tüm çevik olma hevesinin temelinde bu iki element yatar. Bu iki elementin olmadığı yerde çeviklik olamaz. Uygulamayı istekler doğrultusunda yoğuramadıktan sonra ne proje yönetiminin, ne Scrum'ın ne de çok iyi programcılardan oluşan ekibin bir anlamı kalır. Proje yöneticisi istediği kadar tavana zıplasın; katılaşmış bir uygulayı yeni kod yazarak yumuşatamayız, sadece test yazarak ve devamlı refactoring yaparak bu amaca ulaşabiliriz. Yazılımda işin özü budur: hamuru devamlı yoğrulacak kıvamda tutmak. Hamuru yoğurabilmek demek müşteri gereksinimlerine cevap verebilmek demektir, yani anında görüntü. Anında görüntüyü katılaşmış bir hamurla, yani yazılım sistemi ile yapamayız. Çıkarılması gereken sonuç çok basit aslında: sadece çevik süreç uygulayan projeler yoğrulabilir hamur üretir ve müşterinin yeni gereksinimlerine cevap verebilir.

Çevik olabilmek için çevik yazılım metotlarına hakim olmak gerekir. Her sabah on beş dakika toplantı yaparak elde edilebilecek bir durum değildir bu. Çevikliği anlayanlarla, anlamayanlar arasındaki fark budur. Çevikliği anlamayanlar çeviklik buzulunun (eisberg) su üstündeki kısmını görürler. Su üstünde görünen kısım, su altında kalan kısımdan çok küçüktür. Asıl çeviklikle haşır, neşir olma su altında olur. Her şeye Scrum, lean, agile diye isim verenler su üstünde yaşarlar, suyun altında olup bitenlerden bihaberdirler, çünkü derine dalmaya korkarlar. Onlar değişikliği zaten sevmezler. Eski çalışma tarzlarına çeviklik etiketini yapıştırıp hayatlarına devam ederler. Çeviğiz diye kendilerini ve etraflarındaki insanları kandırırlar. Hamuru yoğururken sıvı tükenmeye yakın panik olurlar. Ne yapacaklarını şaşırırlar, neden böyle oldu diye kara kara düşünürler. Çevik olduklarına kendileri de o kadar inanmıştır ki, suçu çeviklikte bulurlar. Çevikliği sevmemeye başlarlar. Onlar için suçlu olan çevik olma paradigmasıdır. Bilmezler ki çevikliğin hakkını verememişlerdir, çünkü ne olduğunu tam olarak kavrayamamışlardır. Değişikliğe açık olmadıkları için kavramaları da kolay değildir.

Gelelim çevik olmayı anlayanlara. Onlar bahsettiğim iki anahtarın sahibidir. Test güdümlü kod

yazmayı bilirler ve her fırsatta bunu uygularlar. Bunun yanısıra refactoring konusunda uzmandırlar. Her yeni müşteri gereksimi ile daha önce verdikleri tasarım kararlarını revide ederler. Uygulamanın katılaşmaya başladığında mevcut tasarımın yetersiz olduğunu anlarlar ve bu tasarımı bozup, yeni bir tasarım yapmaktan çekinmezler. Cesurdurlar. Bilirler ki ellerindeki testler yeniden yapılandırma işlemini mümkün kılar. Buradan çıkardığımız ilk sonuç şudur: çevik olabilmek için test güdümlü yazılım yapmak önem taşımaktadır. Test güdümlü yazılım yapılamıyorsa, o zaman uygulamanın büyük bir kısmını kapsayacak şekilde test kodu yazılmalıdır. Oluşturulan testlerin otomatik olarak çalışması gerekmektedir. Yeniden yapılandırma (refactoring) işlemlerinin birçok yan etkisi olabilir. Onları hızlı bir şekilde lokalize edebilmek için otomatik çalışan testlere ihtiyaç duyulmaktadır. Testlerin olmadığı ya da yetersiz olduğu projelerde programcılar yeniden yapılandırma işlemine cesaret edemezler. Cesaret edemedikleri için yazılım sistemi her gün biraz daha katılaşır ve bir zaman sonra yoğrulamaz hale gelir. Eğlencenin bittiği an budur. Bu noktadan itibaren programcılar için cehennem azabı başlar. Fazla mesailer yapılır, beraber ağıtlar yakılır, kollektif çığlıklar atılır, sende mi Brütüs, sana o kadar emek verdik, yaptıkların bize caiz mi diye yazılım sistemine yüklenilir. Ama Brütüs suçsuzdur. O beni yoğurmayın dememiştir ki. Yoğurma cesaretini programcılar gösterememiştir.

Gerçek çevikliğin temelinde gerçek yazılım mühendisliği metotları yatar. Birincisi test güdümlü yazılım; ikincisi yeniden yapılandırma (refactoring); üçüncüsü eşli programlama; dördüncüsü sürekli entegrasyon; beşincisi basit tasarım; altıncısı iteratif sürüm oluşturma; yedincisi ….

Bu metotların hepsi insanların bilgisayarları icadı ve program yazmaları ile birlikte yer yer uygulanmış metotlardır. Hiç biri yeni icat edilmemiştir Ama bundan on, on beş sene önce Extreme Programming (XP) olarak topluca karşımıza çıktılar. XP bünyesinde proje yönetimi için de metotlar barındırmaktadır. XP bu metotları Scrum'dan almıştır. Ama Scrum bünyesinde XP'nin sahip olduğu çevik metotlar yoktur. Bu yüzden içinde çalıştığım hemen hemen her Scrum projesi sadece kağıt üzerinde çevik olabilmiştir. Saha'ya inildiğinde çevikliğin bir tane atomuna bile rastlamak mümkün olmamıştır.

Sadece proje yönetmek ve programcıların ne yaptığını kontrol etmek için icat edilmiş sözde çevik süreçleri kullananlara sesleniyorum buradan. Gerçek çevik yazılım metotlarını kullanmadığınız sürece çevik olmanız bir hayal olarak kalacaktır. Bu yüzden sürdürdüğünüz birçok proje 130 km/h ile duvara tosluyor. Cesaret edip işin temeline inmeniz gerekiyor. Daha fazla mühendis olup, çevik metotlara hakim olmanız gerekiyor. Her gün on beş dakikalık toplantılarınızdan vaz geçin demiyorum. Bu yerinde bir aktivite. Ama zaman ayırıp bir mühendis kafasıyla sistematik olarak çevik yazılım metotları ile ilgilenin, onları öğrenin, onlara hakim olun, onlari uygulayın. Her Scrum projesini testlerin yine en son safhada yazıldığı ya da zaman yetersizliğinden dolayı yazılamadığı bir ortama çevirmeyin. Profilime ben de kulağa hoş gelen Scrum Master ünvanını koyarım. Ama bu benim ne kadar çevik metotlara hakim olduğumu yansıtmaz. Ünvanlardan çok, hakim olduğunuz gerçek çevik yazılım metotları ile övünün. Hamuru yoğuran ünvan değildir. Hamuru yoguran tecrübeli beyin ve sistemli ve sonuç getiren metotlar kullanmaya alışmış ellerdir. Gerçek yazılım mühendisleri ile düzmece mühendislerin arasındaki fark işte budur. Parayla satın alınan sertifikalar programcıyı programcı yapmaz. Bu sevdadan vazgeçin. Bu sözde çevik yöntemlerle ne dünyayı kurtarırsınız ne de yazılım dünyasına artı bir değer katarsınız.

Çekirdekte gerçek meselenin uygumalayı devamlı yoğurmak olduğunu göz ardı eden bu sözde çevik yöntemler, çevik teriminin her halt için kullanılması ve enflasyona uğramış olması, çevikliğin içinden çıkılamaz bir hale gelmesine sebep olmuştur. Kafalar iyice karışmıştır. Hangi yöntem çeviktir, nasıl çevik olunur, bu sorulara artık kimse net olarak cevap verememektir. Çeviklik artık ifade gücünü yitirmiştir. Bu gerçekten üzücü bir durum. Bir nevi komplo. Sanki eski şelale (waterfall) yöntemleriyle büyümüş bir zümre çevikliği ortadan kaldırmak için iş birliği içindedir. Böyle birsey yok doğal olarak. Fantazim yine kanatlandı.

Beni bir Scrum düşmanı olarak görmeyin. Doğru uygulandığında iyi bir başlangıç olabilir. Ne yazik ki Scrum'ın doğasından kaynaklanan bir sorun bu; Scrum'ı doğru uygulamak hemen hemen imkansız; her türlü uygulama tarzına açık; somut değil; isteyen istediği tarafa çekiyor, bu yüzden ortaya çok komik görüntüler çıkıyor. Scrum doğru uygulandığı taktirde ve proje bünyesinde XP vari yazılım metotları kullanıldığında bir proje tam anlamıyla çevik olabilir. Onun haricinde bu imkansız.

Çeviklikte her şeyin başı çevik mühendislik metotlarıdır. Bunun başında test güdümlü yazılım ve refactoring gelir. İçinde bu elementleri barındırmayan bir süreç çevik yazılım süreci olamaz. Para basmak ve dandik sertifikalar dağıtmak için oluşturulmuş sözde çevik süreçlerden uzak durmakta fayda var. Cesaret gösterip işin özüne inelim. Yeniliklere açık olalım. Yazılım mühendisi isek o zaman bir mühendis gibi çalışalım, sistemli ve metotlu. Buna izin verilmediği yerde durmayalım, baş kaldıralım. Yazılımda çeviklik sadece çevikliği kavramış mühendislerle mümkündür. Eski kafa bir proje yöneticisini Scrum Master yaparak proje çevikleştirilemez. Projede çevikliği yazılım mühendisleri ateşler, meşaleyi onlar ileri götürür, Scrum Master olmuş birisi değil. Onlar en fazla seyirci olabilirler. *Sahada maçı oynayan çevik yazılım mühendisleridir*. Çeviklik konusunda bunun harici her şey hikayedir.

Battı Balık Yan Gider

http://www.kurumsaljava.com/2012/04/14/batti-balik-yan-gider/

Batan projeleri ben batan gemilere benzetiyorum. Batmaya meğil gösteren bir gemi nasıl bunu belli ediyorsa, batmaya meğilli bir yazılım projesi de parmak kaldırıp, dikkat ben batıyorum der. Bir projenin batacağı nasıl anlaşılır? Şöyle:

- Big design up front olarak isimlendirilen, kahin vari, bugünden yarının nasıl olacağını bilme kibiri ile tüm yazılım mimarisi ve tasarımının yazılım öncesi yapılması. Kutsallaşan bu tasarımı sorgulamak ve değiştirmek imkansızdır. Yazılımın ana kurallarından birisi: değişmeyen değişikliğin kendisidir. Madem bunu biliyoruz, neden proje başlamadan herşeyi ön görmeye çalışıp, acayip acayip tasarımlar yapıyoruz? Bu mimari ya da tasarımın müşteri gereksinimlerini bir zaman sonra taşımayacağı ortada değil midir? Müşterinin neye ihtiyacı olduğunu nasıl tahmin edebiliriz? Kendisi bile bunu bilemez. Piyasa koşulları bugünden yarına öyle bir değişir ki, proje müşteri için anında tüm anlamanı yitirebilir. Projelere pragmatik bir tasarım ile başlanmalıdır. Gerekli olduğu zamanlarda bu tasarım revide edilerek, güncelleştirilir.
- Birim, entegrasyon, regresyon, fonksiyon ya da onay/kabul testlerinin olmaması. Sıfır birim test kodunun olduğu projeler bilirim. Birim testleri olmayan bir projeyi yeniden yapılandırmak (refactoring) imkansızdır. Ya bazı kafalar bunu bilmiyor ya da umursamıyorlar. Test konseptlerini bilmeyen programcılar olabilir. Bu bir ayıp değil. Hemen oturup öğrenmeliler. Ama asıl problem ayağı yere basmayan zaman tahminleri ile proje planlaması yapan proje yöneticileridir. Baktılar proje yetişmeyecek, askerlerine emir vererek, testlerin yapılmamasını sağlarlar, çünkü test yazmak zaman kaybıdır onlar için. Böyle yapmaya devam edin!
- Programcıların DRY ve KISS prensiplerine sadık olmamaları. Bu kodun bakımını, değiştirilmesini ve geliştirilmesini zora sokan bir durumdur. Bir noktadan sonra (point of no return) kod tabanı o kadar katılaşır ki, yeni müşteri gereksinimleri için yoğrulamaz hale gelir. Çöpe at, yeniden yap!
- Sürekli entegrasyon yapılmaması. Bakımın ve geliştirmenin kolay olabilmesi için yazılım sistemlerinin modüler olması gereklidir. Modüler bir yapı entegrasyon işlemlerini gerekli kılar. Altı ayda bir entegre edilen modüler bir sistemin adam akıllı entegre edilemeyeceği ortadadır, çünkü entegrasyonun doğası sorunludur. Bu sorunu aşmak için altı ayda bir değil, yazılım sistemi her gün entegre edilmelidir. Bunun için Jenkins ya da Cruise Control gibi entegrasyon serverleri kullanılabilir. Ya sürekli entegre et, ya da bu diyarı terk et!
- Projelerde maliyeti düşürmek için tecrübesiz programcıların çalıştırılması. Üniversite mezunu bir gençten bir usta kıvamında kod yazması beklenemez. Usta programcılar ağaçta yetişmiyor. Zaman içinde yeni üniversite mezunu yazılımcı gençlerde tecrübe kazanıp, ustalaşacaklardır. Lakin bir projeye çok sayıda genç yazılımcı dahil edip, projenin zamanında tamamlanmasını beklemek utopiktir. Eğer projede hiç usta programcı yoksa, iş daha da kötü olacaktır. Eğer projede usta programcılar varsa, bunlar zamanlarının büyük bir kısmını gençleri eğitmek için harcayacaklardır. Bir projede genç yazılımcı olmasın demiyorum. Ama mutlaka iş bitirici usta programcılar olmalı ve bu programcıların kendi işlerine odaklanmaları saglanmalıdır.

- Mimari kararların firma bünyesindeki merkezi bir noktada alınması. Çoğu büyük firmada kendilerini mimar grubu olarak tasvir eden bir yapıya rastlamak mümkündür. Firmada yazılım alanında olup biten herşeye burunlarını sokma alışkanlığı vardır bu grubun. Herşeyden haberdar olmak isterler, aldıkları tüm mimari kararların uygulandığını titizlikle takip ederler. Oysaki merkezden birileri ne yaptığımıza bir göz atacak diye zaman kaybetmek anlamsızdır. Böyle kararların merkezi bir yerden, sahada ne olup bittiğini bilmeden alınması proje gidişatını zora sokabililr. Elini kirleten proje ekibindeki programcılardır. Projeyi ilgilendiren tasarım kararlarını onlar almalı ve uygulamalıdırlar. Merkez mimar grubu yine genel gidişatı yönlendirici kararlar alsın, ona birşey demiyoruz, ama lokal projelere karışmasınlar, karışmak istiyorlarsa sahaya inip, ellerini kirletsinler. Bakalım o kararları yine öyle rahatlıkla alabilecekler mi!

Listemize müşterinin ne istediğini anlamamak, yanlış teknolojik araçların seçimi, yazılım ekibinin yeteneklerine güvenmemek, iteratif bir çevik süreci kullanmamak gibi daha birçok neden ekleyebiliriz.

Yazılım genç bir sektör değil. Genç bir sektörde çalışıyoruz argümanının arkasına saklanarak, batan projelerin sorumluluğundan kendimizi kurtarmaya çalışmamız doğru değil. 1950, belki de daha öncesinden beri insanlar programlar yazıyor. Donanımcılara bir baksanıza. Almışlar başlarını, gidiyorlar. Bilmem kaç çekirdekli işlemciler yapıyorlar. Biz yazılımcıların onları takip etmesi bile hayal oldu. Donanımda muazzam gelişmeler olurken, biz yazılımcılar olduğumuz yerde sayıyoruz. Yazdığımız programlar birden fazla çekirdek üzerinde koşmakta zorlanıyor. Daha bu işe bile hakim değiliz. Lakin bunların hepsi yazılımdan anlamıyoruz anlamina da gelmez. Yazılım mühendisi olarak bilmemiz ve uygulamamız gereken prensip ve pratikler var. Bunlar:

- Extreme Programming gibi bir çevik sürecin adapte edilmesi
- Test Güdümlü Yazılım
- Sürekli Entegrasyon
- Yeniden Yapılandırma
- Tasarım Prensipleri
- Yazılım Testleri
- Tasarım Şablonları

Projelerin batmasında biz yazılımcılar da rol oynuyoruz. Kendimizi geliştirip, yazılım pratik ve prensiplerine daha çok hakım olmamız gerekiyor. Bu konulara işimize olan saygımızdan dolayı daha fazla odaklanmalıyız. Bu konulara hakim olan üstatların peşlerini bırakmamalıyız. Onlardan öğrenebildiğimiz herşeyi öğrenip, işimizi daha kaliteli yapmaya çaba sarf etmeliyiz.

Ama tek suçlu biz değiliz. Proje yöneticileri herşeyi doğru yapıyor denemez. Firma bünyesinde verilen yanlış politik kararlar, yöneticilerin programcıları insan olarak değil de, kaynak olarak görme eğilimleri, ayakları yere basmayan proje planlaları, maliyeti düşürme güdümlü işçi alımları, genel olarak kötü yönetim, projelerin başarısızlıkla sonuçlanması çabuklaştırmaktadır.

Biz programcılar işimizi doğru yaparsak en azından projeyi siz batırdınız diye bize yüklenemezler :)

Kod Kata ve Pratik Yapmanın Önemi

http://www.kurumsaljava.com/2012/04/07/kod-kata-ve-pratik-yapmanin-onemi/

Bizim ailede müzisyen geni var, babamdan bana geçmiş olsa gerek. Babam çok iyi bir ses sanatçısıdır. Keşke benim de onun kadar güzel sesim olsa diye düşünmüşümdür her zaman. Ama ne yazık ki yaratıcının benimle olan planları başka türdenmiş. İyi bir dinleyici olduğumu düşünüyorum. TSM parçalarını seslendirmeye çalıştığımda babam, detone olmadığımı söyler. Ama sesimin ne kadar kötü olduğunu ben bilirim. Keşke babam gibi güzel bir sesim olsaydı. Ufakken hatırlıyorum: TV yok; internet yok; radyo var; arkası yarın ve TSM var. TRT radyo yayınlarını dinleyerek büyüdüm; ne güzel günlerdi…

Bendeki müzisyen genini aktif hale getirmek için 2002 senesinde bir elektro gitar aldım. Aradan tam on sene geçmiş ve ben hala gitar çalamıyorum. Bir müzik aletini çalmaya çalışmak büyük fedakarlık istiyor. Eğer motivasyon yoksa o zaman başarı sağlamak hemen hemen imkansız. Motivasyonun yanısıra düzenli olarak pratik yapmak gerekiyor. Ben bunları ne yazik ki beceremedim ve gitar çalmak benim için her zaman bir hayal olarak kalacak. Şimdi bunun programcılıkla ne alakası var diyeceksiniz. Anlatmaya çalışayım.

Müzisyenler performans ve pratik yapma arasında ayrım yaparlar. Biz programcılar yapmayız! Müzisyenler için sahnede olmak performans yapmaktır. Programcılar için de iş yerine giderek kod yazmak performanstır. Müzisyenler düzenli olarak sahne harici pratik yaparlar. Biz programcılar yapmayız. İş yerinde yazdığımız kodları pratik yapmak olarak algılarız. Çok iyi müzisyenler pratik yapmaya çok önem verirler ve daima saatler boyu bu konuda çalışırlar. Bu yüzden sahnede çok muazzam bir performans ortaya koyarlar, çünkü yaptıkları pratikler onların sular seller gibi müzik yapmalarını sağlar. Biz programcılar pratik eksikliğinden dolayı zaman zaman kod yazmakta zorlanırız. Sular seller gibi program yazamayız, çünkü pratik yapma özürlüyüzdür. Pratik yapmanın ne olduğunu tam olarak bilmeyiz ve performansla pratiği birbirine karıştırırız. Bu yüzden ustalaşma sürecimiz zora girer. Orta halli idare edip gideriz. İçinde çalıştığımız projeler bizi sanki orta halde tutmak için ağız birliği yapmış gibidir. Proje yöneticilerinin ya da iş verenlerin, programcıların orta halden ustalığa geçmesi için çaba sarfettikleri enderdir. Doğruyu söylemek gerekirse biz onlar için kağıt üzerinden sağa sola kaydırdıkları kaynaklardan başka birşey değillizdir. Belki de çok iyi olmamızı bile istemezler, çünkü daha fazla maaş isteme ya da daha iyi bir iş bulma fikri aklımıza gelebilir. Bu yüzden ustalaşma sürecinde onlardan medet ummak doğru değildir. Kendi başımızın çaresine bakmamız gerekir.

Nasıl pratik yapmayan bir müzisyenin sonu sahnede performans yaparken vahim olacaksa, pratik yapmayan programcının da sonu bundan farklı olmayacaktır. Pratik yapmayan programcı bir şeyleri çok kısa bir zaman diliminde bitirmek zorunda kalıp, strese girdiğinde yıllardan beri farkında olmadan geliştirdiği ve öyle pekte verimli olmayan yöntem ve mekanizmaları kullanmaya başlayacaktır (bu konuya daha sonra detaylı olarak değineceğim). Bunun önüne geçmek için programcının iş haricinde pratik yapması gerekmektedir.

Karate yapanlar katanın ne olduğunu bilirler. Kata bir karate ögrencisinin tekrar tekrar uyguladığı belli beden hareketlerinden oluşan şemadır. Sahip olduğu kuşağa göre karate

ögrencisinin yaptığı katalar değişir. Karate ögrencisi katalarını yaparak karate yapma yeteneğini pekiştirir. Tekrar tekrar aynı kataları yapar, bıkmadan, sıkılmadan, usanmadan. Zamanı geldiğinde, örneğin kuşak imtihanlarında katalarını sular seller gibi sergiler ve bir üst kuşağa geçer. İmtihan olma ve katalarını ustalarına gösterme bir karate ögrencisi için performantır. Başarı sağlayabilmek için imtihan gününe kadar daima kataları üzerinde çalışır, yani pratik yapar. Katalarını yapmayan bir karate ögrencisinin imtihan günündeki hali malumur. Ama burada kata yapmanın ana amacı imtihanan geçmek ve bir üst kuşağı kazanmak değildir. Maksat usta bir karateci olmaktır. Bunun yolu da katadan geçer yani pratik yapmaktan.

Kata, programcıların kullanabileceği bir metafordur. Programcının uyguladığı kod katasıdır. Performans harici zamanlarımızda her gün on beş dakika, yarım saat ya da bir saat kod kata pratiği yapabiliriz. Örneğin benim sık yaptığım katalardan birisi roma rakamları kod katasıdır. Her gün ya da gün aşırı bu ve buna benzer kod kataları yaparım. Burada amaç çalışır bir program geliştirmek değildir. Çok basit bir kod katası bile işinizi görecektir, örneğin FizBuz kod katası.

Uyguladığımız kod kataları yaptığımız bazı işlemlerin, kod yazarken verdiğimiz tasarım kararlarının otomatikleşmesini sağlar. Örneğin ben yazılım geliştirme aracı olarak Eclipse kullanıyorum. Yaptığım kod kataları Eclipse bünyesinde bulunan ve fare kullanmandan yazılım yapmayı sağlayan kısa yol tuşlarını – shortcut key (örneğin STRG+SHIFT+F otomatik kod formatlaması için kullanılır) aklımda pekiştirmemi sağladı. Artık kod yazarken hiç düşünmeden neredeyse otomatikleşmiş bir seviyede kısa yol tuşlarını kullanıyorum ve çok daha hızlı programlayabiliyorum. Öğrenmem gereken daha çok kısa yol tuş kombinasyonu var, ama nasıl verimli bir şekilde öğrenebileceğimi biliyorum. Kod katası yapmadan önce birçok kısa yol tuşunu performansım (iş yerinde kod yazma) esnasında kullanmışımdır. Bazen internetten araştırma yapmışım ve yeni tuş kombinasyonları öğrenmişimdir. Lakin birkaç temel kısa yol tuş kombinasyonu harici bunları devamlı aklımda tutmam mümkün olmamıştır. Çok egzotik tuş kombinasyonları var. Bunları akılda tutmak için her gün kod pratiği yapmam gerekiyor. Kataları yaparken tekrar tekrar aynı kıya yol tuşlarını kullandığım için bunları akılda tutmam ve ezberlemem kolaylaşıyor. Ezberimde olan kısa yol tuşlarını da performansım esnasında kullanmam da çok kolay oluyor.

Kısa yol tuşlarına hakimiyet verebileceğim en basit örneklerden sadece bir tanesi. Kod kataları yaparak bir programcı yazılım esnasında gerek duyduğu yetenekleri geliştirebilir. Örneğin test güdümlü kod kataları yapmaya alışmış bir programcı, iş yerinde kod yazarken ister istemez bir test sınıfı oluşturarak işe başlayacaktır, çünkü bunu yapmaya alışmıştır. Test yazsam mı yazmasam mı diye kara kara düşünmez ve hemen test kodunu oluşturmaya başlar. Kafasındaki çalışma kalıbı artık bu şekildedir (doğru olan çalışma tarzı zaten budur). Test sınıfı ve metotları parmaklarından adeta akar gider. Bu durumu gitar çalan bir müzisyenle kıyaslayalım. Gitarist çalmak istediği parçayı devamlı çalıştığı için performansı esnasında beyninde oluşan kalıplar devreye girer ve müzisyen farkında bile olmadan parmakları perdeler üzerinde gider gelir. O esnada gitarist hangi parmakları ile hangi perdeye gidip hangi notayı basması gerektiğini düşünmez bile. Eğer düşünürse o zaman pratik eksikliği var demektir ve çaldığı parça bu durumun göstergesi olacaktır. Yeniden yapılandırma (refactoring) kataları yapan bir programcı bir switch komutunu ya da bir for düngüsünü nasıl yok edeceğini ya da yeniden yapılandıracağını bilir. Bu konuda tecrübesiz bir programcı genelde switch komutunun

kullanılmasını bir sorun olarak bile algılamayabilir. Bunu sorun olarak algılasa bile bir switch komutunu nesneye yönelik programlama teknikleri kullanarak ortadan kaldırmada zorlanabilir. Buna karşın bu konuda pratik yapmış bir programcı hiç gözünün yaşına bakmadan switch komutunu dakikalar içinde yok edecek ve kodu bakımı ve geliştirilmesi daha kolay bir hale getirecektir. Pratik yapmış programcı için başka bir alternatif yoktur; switch komutunu görür görmez beyni ne yapması gerektiğini bilir; programcı hemen harekete geçer.

Buradan yola çıkarak yıllar içinde farkında olmadan geliştirdiğimiz bazı yöntem ve kalıpların neden verimsiz program yazmamıza sebep olduklarına değinmek istiyorum. Hiç zaman yetersizliğinden dolayı çok hızlı bir şekilde program yazmak zorunda kaldınız mı? Böyle bir durumda nasıl kod yazıyorsunuz? Kullandığınız yöntem ve kalıplar nelerdir? Burada tasarım kalıplarından (design pattern) bahsetmiyorum. Değinmek istediğim daha çok beyninizde hangi mekanizmaların çalıştığı ve ellerindenizden hangi kod satırlarının nasıl döküldüğü. Ben kendimdem örnek vereyim. Kata, refactoring ve diğer yazılım tekniklerine kullanmadan önce oluşturduğum sınıflar ve metotlar binlerce ya da yüzlerce satırdan oluşmakta idi. Özellikle sürüm zamanı yaklaşmaya başlayınca panik olur, saçma sapan, ama en azından çalışan kod yazardım. İki gün sonra yazdığım kodları okuduğumda bu kadar kötü mü yazılır kod diye hayretlere düşerdim. Test sınıfları oluşturmamam da cabasıydı. Programcı bu gibi durumlarda ister istemez yıllarca farkında olmadan geliştirdiği ve öyle pekte verimli olmayan yöntemlerine geri döner. Normal şartlar altında özenle seçilen sınıf, metot ve değişken isimleri, zor şartlar altında artık pek önemsenmez, çünkü stresten dolayı ve günü kurtarmamız gerektiği için otomatik olarak beynimizdeki verimli olmayan mekanizmalar devreye girer. Bu davranışı hayatı tehlikede olan bir canlı ile kıyaslayabiliriz. Canlı kendini tehlikede hissettiği anda iç güdüleri ile haraket edip, tehlikeden uzaklaşmaya çalışacaktır. İnsanlarda da durum farklı değildir. Kendimizi tehlikede hissettiğimiz zaman ya da hayatımızı kaybetme rizikosuyla karşılaştığımızda beynimiz adrenalin hormonunu salgılar. Panik oluruz, ama iç güdüsel davranarak tehlikeye karşı koymaya çalışırız. Bu esnada beynimizde çalışan programlara hükmetmemiz hemen hemen imkansızdır. Onlar otomatik olarak çalışır ve bizim teklikeye karşı koymamızı sağlarlar. Aynı şekilde stres altında olan programcı da iç güdüsel hareket edecek ve daha önce geliştirdiği ve kullandığı tehlikeden kurtulma rutinleri devreye girecektir. Programcı bir an önce çalışan bir kod oluşturmak istiyecek ve bunun için sahip olduğu tüm prensipleri ayaklar altına alacaktır. Böylece okunması çok güç yüzlerce, binlerce satırdan oluşan metotlar, sınıflar oluşacaktır. Bu programcının gücüne gitmez, çünkü kendisi de farkında olmadan böyle çalışmak zorunda kalmıştır. Oysaki programcı sık pratik yapmış olsa böyle yapıların oluşması mümkün değildir. Neden? Kod kataları programcının beynini bir nevi yeniden programlar. Kod kataları programcının beyninde yeni kalıpların ve mekanizmaların oluşmasını sağlar. Programcı strese girdiğinde bu yeni mekanizmalar devreye girer. Aynı gitar çalan usta bir gitaristin parmaklarını farkında olmadan perdeler üzerinde oynatması gibi programcı da edindiği yeni mekanizmaları otomatik olarak kullanmaya başlar. Katacı programcı metotların uzun olmasına izin vermez, metot, sınıf ve değişken isimlerini yine özenle seçer, beyni programcıyı devamlı bu yönde çalışmaya zorlar, çünkü programcının beyninde yaptığı katalar ile yeni bir repertuar oluşmuştur ve beyin bunları otomatik olarak kullanmaya başlar. Kullanılan eski yöntemler kaybolur ve programcı stres altında bile çok verimli olabilir. Beyinde bu repertuarı oluşturmak için kod katası yapmak gerekiyor. İş yerinde program yazarak bu repertuarı oluşturmak hemen hemen imkansız. Usta programcılar ağaçta yetişmiyor ya da gökten düşmüyor. Usta programcı

olmak isteyen kişinin büyük fedakarlıklar yapması lazim. Bunun başlangıcı da kod kataları.

Özetle usta bir programcı olma yolunda kod katası yapmak çok büyük önem taşıyor. Bunun yanısıra kataların düzenli olarak yapılması gerekli, aksi taktirde beklenen neticeleri elde etmek zor olabilir. Ben kataları mecbur olduğum için yapmıyorum, işimi ve programcılığı sevdigim ve kod katası yapmayı zevkli bulduğum için yapıyorum. Bir hobi olarak değerlendirebilirsiniz. Kimse mecbur olduğu için karate kursuna gitmiyor, bu işten zevk aldığı ve karateyi öğrenmek istediği için gidiyor. Aynı şekilde bende daha iyi bir programcı olabilmek için kod katalarımı yapıyorum. Her yaptığım katayla yeni birşeyler öğreniyorum. Her kata seansında bir sorunu çözmek için daha kestirme yollar keşfedebiliyorum, başka tasarım kararları vererek başka çözümler üretebiliyorum. Bir zaman sonra artık bazı işlemleri yapmak otomatikleşiyor sanki. İş hayatımda, yani performans yaparken bunun faydasını görüyorum. Programcı olarak kendime olan güvenim artıyor.

Siz daha iyi bir programcı olmak için ne yapıyorsunuz? Kod katası yapmıyorsanız mutlaka denemelisiniz :)

Ustayı usta yapan pratikleridir.

Not: Yaptığım kataları KodKata.com'da sizinle paylaşıyorum. Bir göz atmanızı tavsiye ederim.

Yazılım Maketleri

http://www.kurumsaljava.com/2011/04/25/yazilim-maketleri/

Dün kızıma lego parçalarından oluşan bir set aldım. Bu sabah beraber lego parçalarından kaleler yaparken birşeyin farkına vardım. Yazılım mühendisleri olarak çok soyut şeylerle ugraşıyoruz. Artık soyutluk seviyesi öyle bir hal almış ki, geçenlerde kendimi CPU içide yer alan registerlerin Assembler kullanılarak programlanmasından bahseden bir programcı hakkında "bu kadar low level işlerle uğraşılır mı ya" gibisinden düşünürken yakaladım. Her defasında soyutluk çıtasını bir kademe daha yukarıya çekmeye alışmış ben, somut olan ve ele alınıp, bir mikroskop altında görülebilecek olan CPU registerlerine ne kadar yabancılaşmışım! Bu verebileceğim örneklerden sadece bir tanesi.

İşler soyutlaşdıkça tüm resmi algılamak ve sistemleri modellemek belki daha da kolaylaşıyor, lakin temelde olup bitenleri anlamadığımız sürece, dünyanın en büyük finans krizine sebep olan bankacılardan bir farkımız kalmıyor. Onlarda ne yazik ki temelde çürük kredilerden oluşan yeni finans ürünleri oluşturarak, bilgisiz insanlara pazarladılar. Bu ürünleri kullanarak yeni finans ürünleri soyutladılar. Bu işlem, kimsenin içinde ne olduğunu anlamadığı finans ürünleri oluşana kadar devam etti. Sonuç malum!

Bunların lego ile ne alakası var diye düşünebilirsiniz. Yeni yetişen yazılımcıların OOP (Object Oriented Programming), modüler yazılım sistemleri, komponent tabanlı yazılım, kodun tekrar kullanımı (code reuse) gibi kavramları teorik ve pratik olarak kavramaları çok güç olabilir, çünkü bunlar ilk etapta elle tutulur, gözle görülür olmayan soyut kavramlar. Bu kavramları elle tutulur, gözle görülür hale getirebilseydik nasıl olurdu? Bunun için lego parçalarından faydalanabiliriz.

Siz daha önce bir yazılım sisteminin maketini elinizde tuttunuz mu? Bizler yazılım mimarilerini yazılım mimarlarının kağıtlar üzerinde çizdikleri kutucuklardan tanıyoruz. Bu iki boyutlu resimler örneğin kodun tekrar kullanımını bize ne kadar ifade edebiliyor?

Nasıl mimarlar tasarladıkları evlerin küçük maketlerini yapıyorlarsa, biz yazılım mühendisleri de legoları kullanarak oluşturmak istediğimiz yazılım sistemlerinin maketlerini oluşturabiliriz. Örneğin bir lego parçası tekrar kullanılabilir bir modülü simgeliyor olabilir. Yazılımcı bu lego parçasını eline alıp, sistemin değişik bölümlerinde tekrar kullanabilir. Bunu yaptığı esnada kodun tekrar kullanımının ne anlama geldiğini daha iyi anlayabilir, kodun tekrar kullamının getirdiği avantajları daha iyi görebilir. İnsan elinde tuttuğu üç boyutlu bir nesneyi daha kolay algılama eğilimi gösteriyor. Ben de dün kızımla beraber oynarken bunun farkına vardım. Lego parçalarını bir an için tekrar kullanılabilir yazılım modülleri olarak hayal ettim. Aynı renkte ve boyuttaki lego parçalarını değişik kalelerin (yazılım sistemi olarak düşünün) yapımında kullandım. Lego parçaları birer modül ve modülleri bir araya getirerek, değişik sistemler oluşturmak mümkün. Her defasında temel bir lego modülünü icat etmek zorunda kalmadan tekrar tekrar yeni bir sistemin inşasında kullanabiliyorum. İşte oluşturduğumuz yazılım sistemleri de bu prensibe göre inşa edilmeli. Yazılımda işin sırrı tekrar kullanılabilir modüllerde yatıyor.

Kanımca genç bir yazılımcıya bahsettiğim kavramları somutlaştırıp, eline alabileceği üç boyutlu nesneler olarak sunduğumuz taktirde, bu kavramların algılanması daha da somutlaşıyor. Belki yazılım maketleri oluşturmak yazılım mimarilerinin esnekliğini artırabilir. Ne dersiniz? Bu konu hakkında biraz kafa yormamız lazım!

Koddan Korkan Programcı

http://www.kurumsaljava.com/2013/09/09/koddan-korkan-programci/

Bir senior ve bir junior arasında yapılan konuşmaya kulak misafiri olalım:

Senior: Sakın başkasının kodunu değiştirme! Ufak bir değişiklik ummadığın hataların oluşmasına sebep olabilir. Yaptığın değişiklik sonucu bir hata oluşmadı ise, kimse seni övmez. Ama hata olursa, herkes başına üşüşür. Bunu istediğini zannetmiyorum. **Junior**: Ama agile diye bir şey var, öyle kodu yeniden yapılandırmadan olmazki! Kodu okunabilir hale getirmek lazım. Bu devamlı yapılmassa, bir zaman sonra kodun bakımı zorlaşacaktır. **Senior**: Sen bilirsin! Ben söyleyeceğimi söyledim, kodu değiştirdiğin zaman olacaklardan her zaman sen sorumlu olursun.

Bahsi geçen projenin kod tabanı çok geniş. Gerçekten de en ufak bir değişiklik beklenmeyen neticeler doğurabiliyor. Bu sebepten dolayı ekipte genelde "do not touch a running system (çalışan bir sistemi değiştirme)" mentalitesi oturmuş durumda. Hepsinin temelinde korku var: programcılar koddan korkuyor.

Tecrübeli bir kuaför saçtan korksa idi, saça form verebilir miydi? Usta bir fırıncı hamurdan korksaydı ekmek yapabilir miydi? İyi bir futbolcu toptan korksaydı gol atabilir miydi? Peki koddan korkan bir programcı iyi kod yazabilir mi?

Koddan korkan, kodun hakkını veremez! Koddan korkmanın tek nedeni, yapılan değişikliklerin meydana getirebileceği yan etkileri kestirememektir. Ama programcının yaptığı her türlü değişikliğin yan etkilerini ölçmek için bir aracı olsa korkusu azalır mıydı? Evet, azalırdı ya da tamamen yok olurdu. Böyle bir araç var mı? Evet, var. Peki bu aracın ismi nedir? Birim testi, entegrasyon testi, onay/kabul testi. Her türlü test makbuldür, yeterki değişikliklerin yan etkilerini tespit etmekte yardımcı olabilsiler.

Test yazılmayan projelerde er ya da geç **kod korku oranı** artar. Belki de projelerin gidişatını tahmin etmekte bu bahsettiğim korku oranı bir metrik olarak kullanılabilir. Bu metrik örneğin şu şekilde tespit edilebilir:

Yönetici: Bah, burada bir parça kod var. Bah baalım! Bakınca ne kadar korktun, süle bakim? *Programcı*: Anaaammmm, göstermeyin bana böyle kodlar, öcü gibi, çok korktum! *Yönetici*: Tüh la, proje yakında duvara toslar, biliyodum zateng!

Şu sunumumda çevikliğin temelinde sürekli değişim için cesaretin olduğundan bahsetmiştim. Test yazmak yazılımcının cesaretini ve koda karşı özgüvenini artırır. Test yazılmadığında bunun tersi olur. Veresiye satan yazılımcı olmak ya da olmamak yazılımcının kendi elinde.

Alışkanlıkların Gücü

http://www.kurumsaljava.com/2013/06/25/aliskanliklarin-gucu/

Hiç diyet yaptınız mı? O zaman her diyetten sonra tekrar kiloları fazlasıyla geriye aldığınıza diyet sayınız adedince şahit olmuşsunuzdur. Yeme alışkanlıkları değiştirilmeden hiçbir diyetin başarılı olduğunu görmedim. Bu konuda epeyce bir tecrübeliyim diyebilirim. Tipik bir iş gününün yüzde %90 ından fazlasını masa başında geçiren birisi olarak, kilolarımı dengede tutmak için yapmadığım diyet ve spor türü kalmadı. Ama bu konuda Karatay diyeti ile tanışana kadar kesinlikle başarılı olamadım. Verdiğim kiloları her zaman faylasıyla geriye aldım. Kilo vermek değil, kiloyu tutabilmek marifetmiş, bunu yaptığım bunca diyetten sonra çok iyi anladım.

Bir diyetin kısa bir analizini yapıp, neden başarılı olmadığını inceleyelim isterseniz. Her diyetin başında niyet vardır: kilo verme niyeti. Bu niyet olmadan diyetin başarılı olma ihtimali %0 dır. Hadi niyet sahibiyiz diyelim. İkinci adım hangi diyetin yapılacağının tespitidir. Hadi Diyet A yı yapmaya karar verdiğimizi düsünelim. Diyet A müthiş bir diyet ve bir ay içinde 10 kilo verdirebiliyor. Vay canına deyip, bu diyet ile sağlığımızı ne kadar büyük bir rizikoya soktuğumuzu göz ardı ederek diyete başlıyoruz. 10 kilo muazzam bir rakam. Ne yapıp, edip bu kilolardan kurtulmamız gerekiyor. Bu yüzden vücudumuzu darp edecek bu diyete başlıyoruz….

Bir ay sonra gerçekten 10 kilo verdiğimizi düşünelim. Bu utopik rakamı seçtim, çünkü insanların diyetlerden olan uçuk beklentilerini ifade etmek istiyorum. On senede 10 kilo alıp, bu kiloları bir ayda vermeyi istemek biraz absürd bir düşünme tarzı degil mi? Diyet yapanların %99.9999 kısmı diyet sona erdikten sonra eski yeme alışkanlıklarına geri dönerler. Bu onların altı ay içinde 20 kilo geri almaları anlamına gelir. Eğer sağlıklı yeme alışkanlarına sahip olsalardı, zaten diyet yapmalarına gerek kalmazdı.

Kiloların geri alınmasının tek sebebi, diyet yapanın sahip olduğu yeme alışkanlıklarıdır. Yapılan diyet bireyin yeme alışkanlıklarını değiştirmez. Sadece kilo kaybına sebep olur. İnsanlar kolay kolay sahip oldukları alışkanlıkları terk edemezler, çünkü onlar beyinde yerleşmiş aksiyon şablonlarıdır ve beyin tanıdığı şablonların dışına çıkmayı sevmez. Bu sebepten dolayı hiçbir diyet başarılı olamaz, ta ki diyet sonrası yeme alışkanlarıkları gözden geçirilip, adapte edilene kadar.

İnsan yaradılışı gereği her zaman kendi alışkanlıklarına sadıktır. Yeni alışkanları farkına varmadan edinir, mevcut alışkanlıklarının üzerine titrer. İnsanı comfort zone da tutan ve ilerlemesini engelleyen mevcut alışkanlıklarıdır. Beyinde çalışan ve alışkanlık olarak ifade ettiğim bu subrutinler değişikliğe karşı çok dayanıklıdır. Modifikasyona izin vermezler. Tek amaçları bir döngü içinde görevlerini yerine getirmektir. Bunun böyle kalması için beyin onları devamlı korur ve onlara runtime hizmetleri sunar. Onlara ne kadar çok etki etmeye çalışsak bile bunda çogu zaman başarılı olamayız. Bu yüzden silinmeleri gerekir. Bu onları değiştirmeye çalışmaktan daha zor bir iştir. Ama bunu yapmadığımız taktirde alışkanlıklarımızın rehini olmaya devam ederiz.

Alışkanlıklar organizma için otomatik pilot vazifesi görürler. Beyin onların yardımı ile zor olan “düşün-karar ver-uygula” döngüsüne girmeden aksiyon alabilir. Bu günlük hayata adapte

olmamızı kolaylaştırır. Bu şekilde hiç farkında bile olmadan debriyaja basıp, vites değiştirerek, şehrin bir ucundan diğer ucuna araba ile gidebiliriz. Her vites değiştirmek istediğimizde bu işlemi nasıl yapacağımızı uzun, uzun düşünmek zorunda kalsaydık, araba sürmenin ne kadar zahmetli bir iş olacağını düşünebilirsiniz. Bu açıdan bakıldığında alışkanlıkların olumlu tarafları da yok değil, çünkü günlük hayatımızı kolaylaştırabiliyorlar.

Yazılımcılar da gün içinde, sahip oldukları birçok alışkanlığa maruz kalırlar. Örneğin uzun metotlar yazmak, metot, değişken, sınıf isimlerini özenle seçmemek, birim testi yazmamak, mevcut kod birimlerini tekrar kullanılabilir hale getirmeden copy/paste yapmak gibi. Bu şekilde çalışanlar, devamlı bu şekilde çalışmaya devam ederler, çünkü bu şekilde çalışmaya alışmışlardır. Bu şablonun neden dışına çıkmaları gerektigini sorgulamazlar. Bunu sorgulayanlar belli bir zaman dilimi için bu kötü alışkanlarıklarından vaz geçmeye çalışırlar. Kısa metotlar oluştururlar, kullandıkları isimlere özen gösterirler, belki birim testi bile yazmaya başlarlar, ta ki sürüm zamanı yaklaşıp, günü kurtarmak zorunda kalıncaya kadar. O zaman ne olur? Edinmeye çalıştıkları yeni alışkanları top yekün terk ederek, eski alışkanlıklarına hemen geri dönüverirler, çünkü en hızlı o modda çalışabilirler ya da çalıştıklarını düşünürler. Bu aslinda programcının verdiği bir karar değildir. Daha ziyada beyin otomatik olarak eski şablonları çekmeceden çıkartarak, uygulamaya başlar. Beyin için içinde bulunulan stresli durumdan çıkmanın en kolay yolu mevcut alışkanlıklardan faydalanmaktır, yani işleri otomatiğe bağlamaktır.

Aynı şey Organizasyonel Değişim başlıklı yazımda değindiğim gibi yazılım ekiplerine yeni davranış biçimleri kazandırmak için verilen ugraşılar için de geçerlidir. Yeni bir eğitimden geçmiş bir ekibin, eğitim sonrasında yeni öğrenilen yetilerle yola devam etmesi beklenemez, çünkü ilk fırsatta ekip çalışanları kendi alışkanlıklarına geri dönerler.

Sahip olduğumuz alışkanları bir torbanın içinde toplanmış olarak hayal edelim. Beynimiz herhangi bir işlem gerçekleştirmek istediğinde bu torbaya elini sokup doğru olduğunu düşündüğü alışkanlığımızı seçerek, işletmeye başlayacaktır. Şimdi bu torbayı boşaltıp, yerine yeni alışkanlılar koyduğumuzu farz edelim. Bu durumda yeni alışkanlıklar devreye girecektir, çünkü torbada eski alışkanlıklar bulunmamaktadır. Beyin aslında yeni ya da eski alışkanlıklar arasında ayrım yapmadan, uygun bulduğu ilk alışkanlığı koşturacaktır. Demek oluyor ki torbadaki alışkanlıkları boşaltıp, yerine yeni alışkanlıklar koymamız gerekiyor.

Bize faydası olmayan alışkanlıklardan nasıl kurtulabiliriz yani torbayı nasıl boşaltabiliriz? Yeni alışkanlıklar edinerek. Bilinçli olarak yeni alışkanlık nasıl edinebiliriz? Pratik yaparak. Kod Kata ve Pratik Yapmanın Önemi başlıklı yazımda bu konuyu açıklamaya çalıştım.

Bu arada yazımın başında bahsettiğim Karatay diyetini merak ediyor olabilirsiniz. Bu bir diyet değil, sayın Prof. Doktor Canan Efendigil Karatay hanımefendi tarafından geliştirilmiş bir sağlıklı beslenme metodudur. Diyet olarak da uygulanabilmektedir, lakin ana amacı uzun vadede sağlıklı beslemek ve sağlıklı beslenme alışkanlıkları edindirmektir. Sağlıklı beslenmek isteyenlere şiddetle tavdiye edilir.

Kim Senior Programcıdır?

http://www.kurumsaljava.com/2013/06/21/kim-senior-programcidir/

İlk bakışta bir programcıyı senior yapan teknik bilgisidir. Yüksek seviyede teknik bilgiye sahip olmak için çok tecrübe sahibi olmak gerekir. Yüksek seviyede teknik bilgiye sahip bir şahsın senior olarak algılandığını düşünebiliriz. Lakin teknik bilgi senior olmanın sadece bir boyutudur. Senior mozaiğinin tamamlanması için birçok parçanın bir araya gelmesi gerekir.

Kendisini senior olarak tanımlayan birçok programcı gördüm. Benim açımdan bu programcıların çok büyük bir kısmı senior değil. Şu sebeplerden dolayı:

- Üç ya da beş sene programcı olarak çalıştıktan sonra kıdem alma zorunluluğu varmış gibi kendine durup, dururken senior ünvanını uygun görürler. Böyle programcılar senior ünvanından en uzak olan programcı adaylarıdır.
- Bir ya da birden fazla çatıyı (framework) iyi kullanabildiklerinden dolayı senior olduklarını düşünürler. Bu tip programcılar senior olma yolunda değil, daha ziyada uzman olma yolunda ilerlemektedirler.
- Yıllarca programcı olarak çalışmış olmalarına rağmen ekibin parçası olmaktan acizdirler. Kendi dünyalarında yaşayıp, ekipten kopuk olarak çalışmaya devam ederler. İletişim kurmakta zorluk çekerler.
- Çoğunun müşteriyle iletişimi yoktur ya da çok azdır. Müşterinin tam olarak ne istediğini bilmeden kod yazarlar. Müşterinin ne istediğini bilmediklerinden dolayı nerede durmaları gerektigini bilmezler. Onu da ekleyelim, bu da olsun derken gereksiz zaman ve para kaybına neden olan kod yazarlar.
- Kitap okuyup, kendilerini devamlı geliştirmezler. Geçen yılların onları otomatik olarak senior yaptığını düşünürler. İlk sene bir şeyler öğrentikten sonra dokuz sene aynı şeyleri tekrar etmeyi on senelik tecrübe sahibi olmak olarak algılarlar. Bu onların açısından senior olmak için yeterlidir.
- Test yazmayı zaman kaybı olarak görürler. Başlarını en çok ağrıtan durumun bu olduğunu anlamakta zorlanırlar.
- Teknik olarak belki çok iyidirler ama detaylarda kaybolduklarından veya lüzumsuz şeylerle uğraşmayı sevdiklerinden gereğinden fazla kod yazarlar. Bu hem işverene zarar verir hem de müşteriyi daha çok memnun edecek bir netice ortaya koymaz. Bu listeyi sayfalar dolduracak sekilde genişletmek zor değil. Benim senior olabilmek için iki kriterim var. Yukarıda saydığım durumları zaman içinde aşıp, senior olmak zor değil. Lakin aşağıda sıraladığım iki kriter olmadan senior olmak hemen hemen imkansız. Bu kriterler:

- Programcının müşteri ile olan iletişimde ne kadar başarılı olduğu.

- Programcının başarıyla kaç proje tamamladığı.

Program yazmak çok maliyetli bir iş. Yazılan her satır kod para demek. Eğer bu satır gereksiz bir satır ise, o zaman bu para pencereden dışarıya atılmış para demektir. Programcı yazdığı her satırın gerekli olup, olmadığını nasıl anlayabilir? Müşteriye ne istediğini sorarak. Müşterinin ne istediğini anlamadan yazdığı kodun gerekliliğini bilemez. Müşteri ne istediğini tam olarak ifade

eder ya da etmek zorundadır. Ben şunu istiyorum dediği zaman, programcının müşterinin şu olarak ifade ettiği şeyi koda dökmesi gerekir. Bundan fazlası zarardır, maddi kayıptır. Ne yazık ki kendisini senior olarak gören birçok programcı müşterinin ne istediğini anlamadan oturup, günlerce, haftalarca ya da aylarca program yazabiliyor. Müşterinin ne istediğini tam olarak anlamadan kod yazan, senior programcı olamaz, çünkü kendisini işverenine karşı sorumlu olarak gören bir programcının gönlü kendi yaptığı hatanın ya da keyfi çalışmalarının parasal zarar olarak işverenine yansımasına razı gelmez. Senior programcı bu dengeleri hep kafasında kendisiyle birlikte taşıyan programcıdır.

Başarıyla birçok projeyi tamamlayarak sınır koymanın, sadece gerekeni yapmanın, müşteriyi memnun etmenin, işvereni mutlu etmenin ne demek olduğunu bilmeyen programcı senior ünvanını hak etmez. Başarıyla tamamlanan her proje programcıyı senior olma yolunda ilerlerken pekiştirir, tecrübelendirir. Başarıyla bir proje tamamlamak demek, müşterinin isteklerini tatmin eden bir ürün ortaya koymak demektir. Bu başarıyı birkaç kere tatmadan gerçek anlamda senior olmak çok güçtür ya da imkansızdır.

Senior olmayı bazıları sadece bilgi küpü olmak ve daha fazla maaş almak olarak tanımlıyor demiştim geçenlerde attığım bir tweetde. Senior olmayı sadece bilgili olmak ve fazla maaş almak gibi iki zayıf sıfat ile tanımlamaya kalkmak gerçek senior programcıların hakkını yemek olur. Bu iki şey gerçek senior programcılar için çok öncelikli olmayan şeylerdir. Onlar müşteriyi dinlemeyi, onun isteklerini tatmin etmeyi yeğlerler.

TeletAPI'nin API'si

http://www.kurumsaljava.com/2013/04/11/teletapinin-apisi/

Yazılımcılar detaylara olan sevdaları ile tanınırlar. Bir yazılımcı kullanılmak üzere yeni bir API (Application Programming Interface) geliştirdi ise ve kendisinden bu yeni API'nin nasıl kullanıldığının açıklanması istenirse, size detaylar içinde kaybolmanız ve oturum sonunda API'nin nasıl kullanıldığını anlamamanız garantisini veririm.

Bir API'yi kullanmak nedir, önce bunun tanımlamasını yaparak başlamak istiyorum. Bir API ideal bir dünyada bir yazılım modülünün, kendisinin dış dünyadan kullamını mümkün kılmak için dış dünyaya açtığı pencere ya da pencerelerdir. Biliçli olarak bu tanımlamada kapı kelimesini kullanmadım, çünkü pencereyi kullanarak modülün iç dünyasında olup, bitenleri anlamak mümkün değildir. Ama kapı metaforunda kapıyı kullanarak içeri girmek ve olup bitenleri görmek ve anlamak mümkündür. Bu demek oluyor ki bir API kesinlikle modül içinde olup bitenlerin dış dünyaya sızmasına izin vermez ya da vermemelidir. Modülün nasıl çalıştığını API'ye bakarak anlamak mümkün değildir. API sadece modül ile bir kara kutuymuşcasına interaksiyona girmek için kullanılır. Bu API aracılığı ile sadece modülün sunmak istediği servislerin, izin verdiği ve kendisinin tanımladığı şekilde kullanılabilmesi anlamına gelmektedir. Modülün bu servis ya da servisleri kendi iç dünyasında nasıl hazırladığını anlamak, görmek ya da değiştirmeye kalkmak API aracılığı ile mümkün değildir. Tabi modül buna yine API'si aracılığıyla izin veriyorsa, durum değişik olacaktır.

Bu tanımlamaya göre bir API kullanıcısı API'nin arkasındaki gizli dünyayı bir kara kutu olarak görür. Bu kara kutuyu programlamış programcı için ise durum farklıdır. Programcı tüm detayları bilir ve API'nin nasıl kullanıldığını tanıttığı bir oturumda bu detaylarda kaybolur gider.

Bunun başlıca sebebi yazılımcının kendi geliştirdiği modüle kullanıcı gözlüğünü takarak bakamamasıdır. O her zaman modülü bir beyaz kutu (white box) olarak görür. Bu gözlüğü takarak modülü geliştirir. Bu aslında bir noktaya kadar yapması gereken bir seydir, lakin belli zamanlarda kullanıcı gözlüğünü takabilmelidir. Takamadığı taktirde kullanıcının anlamakta ve kullanmakta güçlük çektiği API'ler oluşabilir.

Yazılımcı modül API'sini mutlaka kullanıcı gözlüğü ile tasarlamalıdır. Bunun için en ideal zaman modül için kod geliştirmeye başlamadan öncesidir. Ortada herhangi bir modül yokken, API'si vardır. Bu şekilde düşünmek bile birçok yazılımcıyı zorlar. Yazılımcı hemen oturur ve hayal ettiği şekilde modülü geliştirmeye başlar. Bu esnada kullanılmayan ya da gereksiz birçok metot ve sınıf oluşur. Bunun sebebi yazılımcının modülün nasıl kullanılacağını bilmemesinde yatmaktadır. En kötü ihtimalle yazılımcı modülün işleyiş tarzını kodladıktan sonra API'si hakkında kafa yormaya başlar. Bu API yırtık yamadan farksızdır ve modülün verimli bir şekilde kullanılmasının önünde engeldir. Bunun bir örneği aşağıda yer alan kodda yer almaktadır.

```
File metaFile = new File("abc");
Storage storage = StorageHelper.createStorage(StorageLoader.load(
    MetaHelper.createMetaFile(metaFile)));
Reader reader = storage.getReader();
Object obj = reader.getObjectById(1);
```

Bu kod parçasına ilk bakıldığında ne yaptığını anlamak zor olmamakla birlikte, kullanılması zor olan bir API'yi ihtiva etmektedir. Bu kod örneğinde Storage isminde bir modül mevcuttur. Kullanıcı bu modülü kullanabilmek için Storage, StorageHelper, StorageLoader, MetaHelper gibi aslında pek varlıklarından bile haberdar olmaması gereken sınıflarla boğuşmaktadır. Bunun sebebi bu modülün bir API'sinin olmaması ve geliştiricisinin geliştirme sürecinde bu modül nasıl en kolay bir şekilde kullanılır sorusuna cevap aramamış ya da bu soruya cevap bulamamış olmasıdır. Yazılımcı API zannettiği birçok sınıfı ortaya atmış, hem modülün basit bir API üzerinden kullanılmasını engellemiş, hem de modül icinde olup bitenleri herkesin göreceği şekilde açığa vurmuştur. Bu tarz bir modül ve API oluşturulması gereksiz bağımlılıkları beraberinde getireceğinden, kodun bakımını uzun vadede çıkmaza sokacaktır. Şöyle bir API işi çok daha kolay yapmaz mıydı?

```
Storage storage = Storage.create("abc");
Object obj = Storage.getObjectById(1);
```

ya da;

```
Object obj = Storage.instance("abc").getObjectById(1);
```

Yukarıda yer alan örneklerde veri tabanından belli bir nesneyi edinmek için önce bir Storage nesnesinin oluşturulması ve akabinde getObjectById() metodun kullanılması yeterli olmaktadır. Modülün API'si sadece bu ya da buna benzer metotlardan oluşmaktadır. Kullanıcının diğer örnekte görüldüğü gibi modül içinde kullanılan StorageLoader ya da MetaHelper gibi sınıfları tanıması ve kullanması şart değildir. Bu modülün kullanımını kolaylaştıran ve API'yi sadeleştiren bir durumdur. Böyle bir API'de kullanıcı sadece Storage sınıfına bağımlı olduğundan, modülün içinde yer alan sınıflar bünyesideki herhangi bir değişiklik kullanıcıyı etkilemeyecektir. Verdiğim ilk örnekte modülün sahip olduğu iç sınıflara bağımlılık doğrudan olduğu için, modülü kullanan kodun kırılganlığı modül üzerinde yapılan her değişiklik ile doğru orantıda artacaktır.

API tasarımı modül için gerekli kodun yazılmasından sonrasına bırakılamayacak kadar ciddi bir konudur. Öyle ise API tasarımını kod yazmadan destekleyecek bir yöntemin kullanılmasında fayda vardır. Bu yöntemin ismi test güdümlü yazılımdır (TDD – Test Driven Development).

Test güdümlü yazılımın en büyük avantajlarından birisi kod yazmadan, geliştirilmek istenen kod biriminin kullanıcı gözünden görülmesini sağlamasıdır, çünkü yazılan tesler oluşturulan kod birimleri için kullanıcı olma niteliğindedir. Yani aslında test sınıfları API'lere yönlendirilmiş kullanıcılardır. Bu sebepten dolayı testler bir kullanıcının ihtiyaçları doğrultusunda kod birimlerini test ederler. Durum böyle olunca test güdümlü yazılım yaparken oluşturlan testlerde akla gelen ilk soru "ihtiyacım nedir ve API bu ihtiyacımı nasıl giderir" şeklinde olmaktadır. API yoksa oluşturulur, varsa doğrudan kullanılır.

Şimdi modül için gerekli kodu yazmadan testler aracılığı ile ihtiyaç duyduğumuz Storage API'sini nasıl oluşturabileceğimizi bir örnek üzerinde inceleyelim. Testler ile birlikte API ve akabinde modül için gerekli kod yavaş yavaş oluşmaya başlayacaktır. İhtiyaç duyduğumuz API'yi aşağıdaki şekilde test ederek başladığımızı farz edelim. Ortada henüz Storage isminde bir

sınıf yok, dolayısıyla bu sınıfın getObjectById() isminde bir metodu da bulunmuyor. Ama biz ihtiyaç duyduğumuz API'yi hayal ederek, yani kullanıcı olarak yola çıkarak böyle bir sınıf ve böyle bir metot olsaydı, istediğimiz veriyi veri tabanından böyle edinebilirdik şeklinde hayal ettik ve bunu da test olarak ifade ettik.

```
@Test
public void
storage_should_deliver_the_value_1(){
    Storage storage = new Storage("abc");
    Object obj = storage.getObjectById(1);
    assertThat(obj.getValue(), equalTo(1));
}
```

Oluşturduğumuz test bizi ister, istemez sade bir API oluşturmaya itmektedir. Bu tarz bir test yazmak bizi kesinlikle önce bir MetaHelper sınıfı, akabinde bir StorageLoader ve daha sonra veri tabanını kullanıma hazırlamak için kullanılan StorageHelper sınıfını oluşturmaya yönlendirmemektedir, çünkü kullanıcı olarak bunlar bizi ilgilendirmeyen, modülün kendi sorumluluğunda olan konulardır. Bizim kullanıcı olarak tek bir beklentimiz vardır, o da API üzerinden belli bir verinin veri tabanından en kolay şekilde nasıl edilenilebileceği konusudur.

Testi oluştururken taktığımız kullanıcı gözlüğü bizi her zaman kullanıcının gereksinimlerini doğrudan tatmin eden bir API'yi oluşturmaya yönlendirmektedir. Kullanıcı gözlüğü modül içinde olup, bitenler ile ilgilenmemektedir, çünkü detayları bilmesi verdiğim ilk örnekteki gibi kafa karıştırıcı tarzda olacaktır. Bu yüzden kullanıcı her zaman en sade API'yi tercih etme iç güdüsüne sahiptir.

Yazmış olduğum testin başarılı olması için Storage isminde bir sınıfın oluşturulması ve bu sınıfın getObjectById() isminde bir metodunun olması gerekmektedir. Bu noktadan itibaren detayları bilen yazılımcı gözlüğünü takarak modülü ve içeriğini düşündügüm şekilde geliştirebilirim. Böyle bir API oluştuktan sonra yazılımcı iç güdüsel olarak Storage modülünün tüm işleyiş tarzını kullanıcıdan saklamaya özen gösterecektir, çünkü kullanıcının getObjectById() metodu haricinde başka bir şeye ihtiyacı olmadığını bilmektedir. Bunu kendisine söyleyen yazdığı birim testidir. Birim testi modülün kullanılma şekillerinden birisini yazılımcıya göstermiştir. Storage modülü için yapılabilecek implementasyonlardan birisi şu şekilde olabilirdi.

```
public class Storage{

private static Storage storage;

private Reader reader;

public static Storage create(String dir){
    if(storage == null){
        storage = new Storage(dir);
    }
    return storage;
}

private Storage(String dir){
    initStorage(dir);
}

private void initStorage(String dir){
    File metaFile = new File(dir);
    reader = StorageHelper.createStorage(StorageLoader.load(
    MetaHelper.createMetaFile(metaFile))).getReader();
}

public Object getObjectById(int i){
    return reader.getObjectById(i);
}

class Reader{
...
}

class StorageHelper{
...
}

class StorageLoader{
...
}

class MetaHelper{
...
}
```

Storage modülünü bir singleton olarak implemente ettim. Görüldüğü gibi Storage.create() statik metodu ile singleton olan bir storage nesnesi edinmek mümkündür. Akabinde bu nesneyi kullanarak getObjectById() metodu aracılığı ile veri tabanından istediğim türde bir veriyi çekebilmekteyim. Bunun yanısıra modül bünyesinde kullanılan tüm sınıfları iç sınıf olarak tanımladım. Bu şekilde dış dünyadan hiç kimse bir Reader ya da StorageHelper nesnesi oluşturamaz ya da kullanamaz. Storage modülü bu şekilde iç dünyasında olup bitenleri tamamen dış dünyadan gizlemekte ve tanımladığı iki metot aracılığı ile kullanımını mümkün kılmaktadır. Bu iki metot Storage modülünün API'sidir. Storage modülünün kara bir kutu gibi işlev görmesi, kullanıcıları etkilemeden modül bünyesinde değişiklik yapılabilmesini mümkün kılmaktadır.

Yapmış olduğumuz API tanımlamasına "API kullanıcı ve sunucu arasında bir kullanım sözleşmesidir" ibaresini ekleyebiliriz. Nitekim bir modül sahip olduğu API'si aracılığıyla dış dünyaya nasıl kullanılabileceğinin mesajını verir. Kullanıcılar istedikleri API metotlarını seçerek modülü kullanmaya başlarlar. Bu kullanıcı ve modül arasında bir bağ oluşturur. Kullanıcı her zaman oluşan bu bağın sağlam olmasını arzular. Değişikliğe uğrayan API kullanıcılarını kırılgan hale getirir. Bunu engellemek için API'nin sıkça değişmemesi şarttır. Durum böyle olunca bir defa kullanıma sunulan bir API'nin değiştirilmesi artık kolay değildir, çünkü sayısı bilinmeyen birçok kullanıcısı vardır. Bunu göz önünde bulundurduğumuzda API tasarımcısı olarak her zaman tutucu (konservatif) bir pozisyonda durarak oluşturduğumuz sınıfları öncelikle iç sınıf ya da package private olarak tanımlamamız gerekir. Sadece bu durumda hemen keşfedilerek kullanılmalarını engelleyebiliriz. Kullanımını engelleyemediğimiz sınıflardan her zaman sorumluyuzdur. Bir API kullanıma açıldığında o API'de ne kadar sınıf ve metot mevcutsa, modülün ömrü boyunca bu sınıflara ve metotlara destek vermek zorunda kalırız, çünkü kullanıcılarını API'yi değiştirerek sinirlendirmemiz gerekir. Bu yüzden API'nin çapı ne kadar küçükse, sorumluluk ve verdiğimiz desteğin oranı o oranda küçük olacaktır.

API'lerin kullanıcılara hitap edecek ve onların gereksinimleri doğrudan ve kolay bir şekilde tatmin edecek şekilde tasarlanmaları şarttır. Windows ortamında C/C++ ile Windows API'sini kullanarak uygulama geliştirenler çok iyi bilirler. Bu API en zor API'lerden bir tanesidir. API sadece birkaç sınıf ya da metot birleşimi bir sey değildir. Aynı zamanda API'nin ihtiva ettiği sınıf ve metot isimlerinin ifade gücünün yüksek olması gerekir. Her sınıf ya da metot için seçilen isim bu API'nin arkasındaki gizli servisin nasıl kullanılabileceğini ifade edebilmelidir. Unutmayalım programlar bilgisayarlar için yazılıyor, lakin kodu okuyan insanlar. Bu yüzden ifade gücü yüksek olan isimlerin seçilmesi API'nin doğru kullanımı kolaylaştıracaktır.

Bol TeletAPI'li günler diliyorum.

Alet İşler, El Övünür

http://www.kurumsaljava.com/2013/02/22/alet-isler-el-ovunur/

Birçok meslekte araç ve gereç sahibi olmadan iş yapmak mümkün değildir. Günlük hayatımızda da birçok araç ve gereci kullanırız. Örneğin bir resmi duvara asabilmek için bir çivi ve bir çekic kullanırız. Çoğu zaman işimizi gördükten sonra başarımız ile övünür, bir sonraki ihtiyacımıza kadar kullandığımız araçları hatırlamayız.

Araçlar zamandan tasarruf etmek ve işimizi kolaylaştırmak için vardır. Bazı araçlar olmasaydı, isteğimiz işi yapmamız bile mümkün olmazdı. Çekiç sahibi olmadan duvara çivi çakmak mümkün değildir. Bunun için bir taş parçası bile kullansak, kullandığımız bu taş parçası bir araç haline dönüşür.

Yazılımcı olarak ta günlük hayatımızda birçok araç kullanırız. Bunların başında kod yazmak için kullandığımız IDE (Integrated Development Environment) araçları gelir. Ben günlük işlerimde genellikle Eclipse IDE'yi tercih ediyorum.

Araçlar işimizi kolaylaştırmak ve zamandan tasarruf etmek için var demiştim. Ne yazık ki bazı yazılımcılar bunun bilincinde değiller. Gerekli araç ve gereçleri kullanmalarına rağmen, zamandan tasarruf ettikleri söylenemez, çünkü kullandıkları araçların hakkını veremiyorlar.

Kod yazmak için kullanılan bir aracın en verimli kullanım şekli, bazı aksiyonlar (örneğin yeni bir sınıf oluşturma, yeni bir metot oluşturma) için kısa yol tuşlarını tanımak ve uygulamaktır. Örneğin bir interface sınıfını implemente etmek istediğimizde, implementasyon sınıfında tek tek metot gövdelerini oluşturmak yerine Eclipse altında STRG+1 tuşlarına baştıktan sonra Add unimplemented methods seçimini yapmak yeterlidir. IDE otomatik olarak interface sınıfı bünyesinde yer alan her metot için implementasyon sınıfında gerekli metot gövdesini oluşturacaktır. Bu işlemi elden yapmak çok büyük bir zaman kaybıdır. Bu aracı kullanıp, hakkını verememektir.

Bir yazılımcının en önemli araçlarından birisi kullandığı IDE olduğuna göre, IDE'nin sundugu imkanlar doğrultusunda yaptığı işlemlerin sürelerini kısaltmanın yollarını aramalıdır. Bunun yolu kısa yol tuşu hakimiyetinden geçmektedir. Bunun yanısıra fare kullanımından sakınmalıdır. Fareye uzanan el bir thread context switch gibidir. İş akışını bir an için bile olsa böler. Programcının en önemli aracı klavyesidir. Klavyeye ek olarak fareyi kullanması, klavyenin hakkını veremediği anlamına gelir. Aynı işi tek bir araç ile yapabilecekken, iki değişik araç kullanarak zaman kaybeder.

Bir başka zaman kaybı türü de yazılımcının on parmak kod yazamamasıdır. Yazılımcının kod kaynağına bakacağı yerde, x tuşu nerede diye klavyeye bakması context switch'dir. Devamlı bu şekilde çalışmak yazılımcıyı hem yorar, hem verimini düşürür hem de zaman kaybetmesine neden olur. Bu sebepten dolayı her yazılımcının on parmak kod yazabilme yetisini geliştirmek için çalışma yapması zaruridir.

Yazılım otomasyon demektir. Otomasyonun olmadığı yerde zaman kaybı var demektir. Ant ya da Maven ile otomatik sürüm oluşturmak yerine, bu işi her defasında elden yapmaya çalışmak

çok büyük zaman kaybıdır. Yazılımcının bu noktada zamanının belli bir kısmını sürüm oluşturma araçlarının kullanımını öğrenmek için harcaması, bu yatırımın sürümlerin daha hızlı ve otomatik oluşturularak amortize olmasını sağlayacaktır.

Kullandığı araçlara hakim olmayan bir ustaya kim usta gözüyle bakar? Elimizin övünmesini istiyorsak, kullandığımız araçlara hakim olduğumuzu gösterelim.

Eklenti (17:51 . 22.02.2013):

Bir yazılımcı için kullandığı en önemli araç bilgisayarıdır. Klasik bir iş bilgisayarları okuma kafası dakikada 5000 ila 7000 arasında dönen sabit disklere sahiptir. Taş devrinden kalma bu teknoloji yüzünden yazılımcılar hergün dakikalar ya da saatlerle ölçülebilen zaman kaybına uğruyorlar. Windows gibi bir işletim sistemi ve yetersiz hazıfa (ram) kapasitesi buna eklendiğinde yazılımcıların mesaileri kabusa dönüşebiliyor. Bunun önüne geçmek için yazılımcını 64 bit genişliğinde, SSD sabit diskine ve en az 8 GB hazıfa alanına sahip bir bilgisayar kullanmalıdır. Bunun altindaki her konfigürasyon işletim sisteminin akışkanlığını bozarak, yazılımcının belli aktiviteler için bekleme süresini artıracaktır ve verimliliğini düşürecektir.

Cahilliğime Verin

http://www.kurumsaljava.com/2013/02/18/cahilligime-verin/

BTSoru.com'da bir soru sorulmuş. Tüm soruları kontrol ederek ihtiyaç durumunda başlığı, içeriği ve etiketleri düzenlemeye çalışıyorum. Bu soruyu incelerken verilen kod örneğini Struts kodu zannettim ve sorunun etiketini Struts olarak değiştirdim.

Kısa bir zaman sonra soruya bir yorum yapan bir arkadaşımızdan bir uyari e-postası aldım. Arkadaş iletisinde etiketin asp.net-mvc olmasının daha anlamlı olacağını bildiriyordu. İlgisi için kendisine teşekkür ettim ve etiketi asp.net olarak değiştirdim. Bilmediğim bir şey vardı: asp.net ve asp.net-mvc iki değişik şeydi.

Kısa bir zaman sonra aynı arkadaşımızdan tekrar bir uyarı iletisi aldım. Çok kibar bir şekilde asp.net ile asp.net-mvc'nin iki ayrı teknoloji olduğunu belirtiyordu. Özür dileyerek sorunun etiketini asp.net-mvc olarak değiştirdim.

Buradan çıkarabileceğimiz bazı dersler var. Bunlar: - Yanlış olanı düzeltmek için çaba sarfetmek her programcının asli görevlerinden birisidir. - Sesini yükseltip, yanlışlara işaret edebilmek cesaret ister. Bunu yapabilen yaptığı işin hakkını veriyordur. - Birilerinin bir şeyler hakkında daha fazla bilgiye sahip olma ihtimali yüksektir. Her şeyi bilirim edasıyla hayatın içinden geçip gitmek abesle iştigaldir. - Daha iyi bilenin yakasına yapışmak ve ondan öğrenmek farzdır. En kolay öğrenme yöntemi budur. - Bu konu hakkında bilgim yok, ama öğrenebilirim demek bir erdemdir. Bilgisi olmadığı halde bilgisi varmış gibi davranmak ahmaklıktır. - Tatlı dil yılanı deliğinden çıkarır. Bahsettiğim arkadaş kendi bilgisel üstünlüğünü ortaya koymak için kühtahça bir ileti de gönderebilirdi. Bunun yerine çok nazik ve kibar bir üslup ile yanlışa işaret etmeyi tercih etti. İki şekilde de kazanan o oldu: hem isteğinin yapılmasını sağladı, hem de efendi bir üsluba sahip oldugunu ispatladı.

Hz. Ali'nin güzel bir sözü var: **Bana bir harf öğretenin kırk yıl kölesi olurum... **

Açık Sözlü Programcı

http://www.kurumsaljava.com/2013/02/17/acik-sozlu-programci/

Programcı takım arkadaşı ile kodu gözden geçirme (code review) seansı yapıyor. Kodun içinde bulunduğu durumdan hoşnut değil, lakin bunu takım arkadaşına söylemiyor. Arkadaşının yanlış anlamasından mı korkuyor?

Tüm yazılımcılar iterasyon sonunda bir araya gelerek geri bakış (retrospective) seansı yapıyorlar. Bazı programcılar boğazlarına kadar dolmuşken, bunun nedenini takım arkadaşlarıyla paylaşmıyorlar. Çok mu sabırlılar?

Günlük stand-up toplantı yapılırken bazı programcılar planın gerisinde kaldıklarını, üzerinde çalıştıkları kullanıcı hikayesinin zamanında tamamlanmasının münkün olmadığını söylemiyorlar. Ekip arkadaşları önünde küçük düşmekten mi korkuyorlar?

Programcı başka bir ekip arkadaşının yazdığı kodu değiştirmeye çekiniyor. Aynı şeyin kendi yazdığı kodun başına gelmesini mi istemiyor?

Verdiğim örneklerin hepsinde bir tikanıklık ve blokaj söz konusu. Bahsettiğim programcılar samimi ve açık sözlü değiller. Bu ne yazık ki çevikliği öldüren, şeker hastalığı gibi çok sinsi bir şey. Çevikliğin temel değer sistemini cesaret, iletişim, geribildirim ve basitlik oluşturuyor. Bunların olmadığı yerde çevik olunması imkansız.

Kimsenin etlisine, sütlüsüne karışmamak çevik süreci sabote etmek gibi bir şeydir. Herkes açık sözlü olma cesaretine sahip olmayabilir. Ama bunun için gerekli ortam oluştuğunda açık sözlü olmamak için de bir sebep yoktur. Bir retrospektive seansında kimseyi suçlamadan ve küçük düşürmeden yanlış giden şeyler üzerinde konuşulabilir. Fikir birliği sağladıktan sonra yolunda gitmeyen şeyleri tekrar rayına sokmak zor değildir. Burada önemli olan samimi olmak ve yanlış giden şeyleri açık sözlülükle dile getirmektir.

Yazılım bir ekip işidir. Bireyin başarısı takımın başarısıyla ya da başarısızlığıyla doğrudan orantılıdır. Tüm takım aynı bot içinde oturduğuna göre, botun ilerleyebilmesi için herkesin aynı şevk ile kürek çekmesi gerekir. "Bot su alıyor, ama bana ne", "bazı arkadaşlar kürek çekmiyor, benim derdim değil", "bazı arkadaşlar kürek çekme tekniğine tam hakim değiller, kim uğraşacak şimdi onlarla" gibi düsüncelere sahip olmak botu hedefine ulaştırmaz.

Samimiyet ve açık sözlülük programcının takım içindeki saygınlığını artırır. Bunun tersi durumunda halka en zayıf yerinden kopacaktır. O en zayıf nokta samimi ve açık sözlü olmayan programcının kendisidir. Bunu herkes bilmesede kendisi çok iyi bilir.

Programcılık Çıtası Yükseliyor

http://www.kurumsaljava.com/2013/01/29/programcilik-citasi-yukseliyor/

Ben Java'nın ilk günlerinden beri bu dili kullanan bir programcıyım. 1998 senesinin bir günü kampüste laflarken bir arkadaşım yeni haberini aldığı Servlet teknolojisinden bahsetmişti. Bugün gibi hatırlıyorum: "vay... demek Java ile appletler harici web programcılığı yapılabiliyor...." demiştim. Java ile geçirdiğim ilk yıllarda hakim olmam gereken konular JDBC, Reflection, RMI ve Servlet gibi teknolojilerle sınırlı idi. On beş sene sonra durum çok farklı! Başlangıçta küçük bir havuzda yüzerken, şimdilerde kocaman bir okyanusun içindeyim ve bu okyanusun ucu, bucağı yok.

Java dili eklenen her yeni teknoloji ile günümüzde bir platform haline geldi. Artık Java dili bu platform üzerinde kullanılan dillerden sadece bir tanesi. JVM üzerinde koşturulabilen Scala, JRuby, Groovy, Clojure gibi birçok dil mevcut.

Bundan beş, altı sene öncesine kadar tipik bir Java programcısı JMS, JTA, EJB, JAXB, JSF, JSP, JAX-WS gibi çeşitli Java çatılarına hakimliği ile övünebilirken, durum bugün çok daha değişik. İsmi geçen çatılar bugün bir Java programcısının repertuarında var olması gereken şeyler. Hepsine hakim olmak bir marifet değil, bir zorunluluk haline geldi, çünkü piyasa bir Java programcısından bunu bekliyor.

Durum sadece bununlu sınırlı değil. Artık JVM üzerinde çalışabilen dört yüze yakın programlama dili mevcut. Bunların başında daha önce ismini verdiğim Scala, Clojure, JRuby, Groovy ve Jython geliyor. Doğal olarak piyasada bu dilleri kullanarak geliştirilen proje sayısı göz ardı edilmeyecek kadar fazla. Twitter, LinkedIn ve FourSquare Scala dili ile geliştiriliyor. Bu durumda klasik bir Java programcısının bu dillere yönelme haricinde fazla bir alternatifi kalmıyor. Bu ekstra öğrenim yükü demek.

Bu durum sadece Java dilini kullanan programcılar için geçerli değil. Net platformunu kullanan programcılar da birçok değişilik ile baş etmek zorundalar. Aynı şey mobil programcılar için de geçerli. Programcılar gidişatı tam anlamıyla kavrayabilmek için ilgi alanlarına giren her şeyi tam teşekküllü takip etmek zorundalar. Bu programcının şimdilerde onlarca blog sitesini takip etmesi, yüzlerce kitap okuması, onlarca yeni programlama dili öğrenmesi anlamına geliyor. Programcının sahip olduğu bilgilerin yarı ömrü aylarla ölçülür oldu.

Bugün profesyonel programcı olarak çalışan insanların çok büyük bir kesimi imperatif dilleri kullanıyor. Bu dillerde eğitim aldılar ve tüm düşünce yapıları bu dillerin sunduğu yapılar üzerine kurulu. Bu programcıların çoğu bugünlerde büyük bir kültür şoku yaşıyorlar. Son günlerde fonksiyonel dillere olan ilgi çok artmış durumda. İmperatif bir dünyadan gelen bir programcının fonksiyonel dillere adapte olması ve anlaması kolay değil. Closure, lambda, higher order functions, functions as first class citizen, recursiv programming, pure functions gibi terimler imperatif programcıların kafasını karıştırmış durumda. Ama piyasa bu konseptleri anlamış programcılar istiyor. Er ya da geç imperatifcilerin fonksiyonel programcılık konseptlerine vakif olmaları gerekiyor. Programcılara yönelik beklentiler yükselmiş durumda.

İs görüşmelerinde de büyük bir değişimin yaşandığı bir gerçek. Eskiden piyasada programcı

bulduklarında öpüp, başına koyan iş verenler, şimdilerde adayları intensif görüşmeler ve testler ardından işe alıyorlar ya da almıyorlar. İş bulmak zorlaştı, çünkü bir programcıda aranan vasıflar değişti. Artık CV'de yer alan kelimelere göre programcı alımları son bulmuş durumda.

İsteyenin parmağını kaldırıp, "Java kelimesinin nasıl yazıldığını biliyorum, ben programcıyım" diyerek iş buldugu devir artık son buldu. Günümüzde programcı olmak ve kendisini programcı olarak yetiştirebilmek çok güç bir hale geldi. Vakıf olunması gereken çok çesitli konu ve konseptler mevcut. Bu yeni süreç artık işine muazzam derecede hakim programcılar doğuracak. Bu işi kıvıramayanlar daha işin başında iken elenip, piyasadan uzak tutulacaklar, çünkü piyasada onların yapabileceği iş olmayacak.

Programcı olma çıtası on sene öncesine göre çok yükseldi ve yükselmeye devam ediyor. Bu güzel bir gelişme. Artık akla, karanın ayrıştığı bir dönemdeyiz. Programcılık hafife alınabilecek ve yarım yamalak bilgi ile yapılabilecek bir iş değil. Bu işi çalışma sahalarındaki her şeyi en ince detayına kadar anlamış, öğrenme kapasiteleri yüksek, değişikliklere adapte olan ve yeniliklerden çekinmeyen programcılar yapmalı. Çıtanın yükselmesi doğal bir seleksiyon yöntemi. Er ya da geç gerçekleşecekti. Bu işe gerçek anlamda gönül verenler ilerleyebilecek, para icin yapanlar ise çok kısa bir zamanda kendilerine baska çalışma alanları aramak zorunda kalacaklar, çünkü bu değişime ayak uydurmaları imkansız.

İyi programcıların ağaçta yetişmediği bir gerçek. Bir programcının iyi bir programcı olabilmesi için çok okuması, öğrenmesi, öğrendiklerini uygulaması ve yenilikleri takip etmesi gerekiyor. Bu vasıflara sahip olan programcıları tanımak zor değil. Sahip olmayanları da tanımak zor değil. Bu işin başında olanlar iş bulamayacak tezini savunmuyorum. Yaşadığımız bu gezegende bugün düne nazaran çok daha fazla programcıya ihtiyaç var. Ama bu ihtiyacın gözü eskisi gibi kör değil. Neyi, hangi tip programcıyı aradığını iyi biliyor ve seçimini ona göre yapıyor. Bu işe gönül verenler, bu işin başlangıcında da olsalar bu ihtiyaca her zaman cevap verebilecek yeteneklere sahip olacaklar. Zaman içinde bilgilerine bilgi, tecrübelerine tecrübe katarak çok daha iyi programcılar olacaklar. Çatının yükselmesi onlar için bir sorun olmayacak, çünkü çatıyı yükselten kendileri olacak. Gelecekte bu tip programcılar ile çalışmak çok ama çok zevkli olacak.

Nasıl Arkadaş Kazanır ve Diğer İnsanları Etkilersiniz

http://www.kurumsaljava.com/2012/12/16/nasil-arkadas-kazanir-ve-diger-insanlari-etkilersiniz/

Dale Carnegie tarafından 1937 yılından yazılan How To Win Friends and Influence People (nasıl arkadaş kazanır ve diğer insanları etkilersiniz) başlıklı kitap her yazılımcının mutlaka okuması gereken ve güncelliğini günümüzde de koruyan çok değerli bir kaynak kitap.

Kitapta günlük hayatta takip edilmesi gereken bazı kurallar yer alıyor. Bunlar:

Kimseyi eleştirmeyin, yargılamayın ve şikayet etmeyin

Bir başkasını eleştirmek faydasızdır, çünkü eleştirilenin kendisini ve haklılığını savunmasını doğurur. Eleştirmek tehlikelidir, çünkü eleştirilenin onurunu incitir, kendisini kötü hissetmesini sağlar. Bir şeyleri eleştirerek, durumun daha iyileşmesini sağlamak mümkün değildir.

Taktiri hak edenlere bunu gösterin

İnsanların en önemli motivasyon kaynaklarından birisi taktir görmek ve kendini önemli hissetmektir. Büyük eserlerin, yapıtların ya da şahsiyetlerin oluşmasında en büyük rolü bu ihtiyaç oynamıştır/oynamaktadır. İnsanlar bu ihtiyaçlarını giderebilenlerin elinde hamur gibidirler. Hak ettiği için övülen ve taktir edilen şahıs bunu hiçbir zaman unutmaz.

Başkalarının içinde istek uyandırın

Bir bireyin istekleri diğer bireyleri ilgilendirmez. Her birey kendi istekleri ekseninde hayatına şekil verir. Bu sebepten dolayı insanları etkilemenin en kolay yolu onların istekleri hakkında konuşmaktır. Her bireysel hareketin temelinde bir istek yatar. İnsanları herekete geçirmek için içlerinde bu isteği uyandırmak yeterlidir.

Başkalarıyla ilgilenin

Baskalarıyla ilgilenenlere insanlar arası ilişkilerde her türlü kapı açıktır. İlgi çekmek için uğraş veren bir bireyin iki senede kazanamadığı arkadaşlıkları, başkalarıyla ilgilenen birisi iki ay içinde elde edebilir. Dost ve arkadaşlığın temelinde bu samimi ilgi yatar.

Gülümseyin

Gülümseme iyi niyetin göstergesidir. Gülümseme onu görenlerin gününü güzelleştirir. Gülümseme "size kanım kaynadı, beni mutlu ediyorsunuz, sizi gördüğüme sevindim" demektir.

Herkes için kendi isminin en hoş ve önemli kelime olduğunu unutmayın

Insanları kazanmanın en kolay yollarından birisi, isimlerini hatırlamak ve onur duymalarını sağlamaktır. İsimler insanları özelleştirir. İsimlerle birlikte gönderilen mesajlar daha başka anlam kazanirlar.

Bunlar gibi birçok tavsiye Dale Carnegie tarafından 1937 yılından yazılan How To Win Friends and Influence People (nasıl arkadaş kazanır ve diğer insanları etkilersiniz) başlıklı kitapta yer almakta. Okunulması tavsiye olunur.

Copy/Paste Programcı

http://www.kurumsaljava.com/2012/11/18/copypaste-programci/

Internetin bu kadar büyümesi ve özellikle Google gibi arama motorlarının günlük iş hayatımızın bir parçası haline gelmesi biz programcılar için ne kadar hayırlı oldu, bilemiyorum. Pek te hayırlı olmadığı kanısındayım. Açıklamaya çalışayım.

Çalıştığım projede bir çalışma arkadaşımın masasında algoritmalarla ilgili kalınca bir kitap gördüm. Havadan, sudan konuşurken kendisi böyle kitapların artık gereksiz olduğunu, çünkü internette istediğin algoritmayı bulup, kullanabildigini söyledi. Doğru, haklıydı.

Bugünlerde Pratik Spring Core 3 isminde yeni kitabım üzerinde çalışmalarımı sürdürüyorum. Spring ile XML bazlı deklaratif transaksiyon yönetimi konfigürasyonu nasıl yapılır diye internette araştırma yaparken, birkaç klasik blog sayfasına rastladım. Bu klasik blog yazılarını tanırsınız. Basit bir örnek üzerinde belli bir konfigürasyonun nasıl yapıldığını gösterirler. Bu örnekleri copy/paste yaparak programcının kendi uygulamasına eklemesi zor değildir. Zamanında az yapmadık. Copy/paste işlemi esnasında insan keşfettiği örneğin altındaki konseptlerin ne olduğunu sorgulamıyor. Programcı için önemli olan bir şeyleri çalışır hale getirebilmek. Ve bunu başardıktan sonra, "ya, bunu çalışır hale getirdim ama, bu işin temelindeki konseptler nelerdir" şeklinde bir sonuca varmıyor ya da varamıyor. İşte internet üzerinden bilgiye bu kadar hızlı, sorgusuz ve sualsiz erişmenin getirdiği hayırsızlıkta bu noktadan itibaren başlıyor. Programcı bir bilgiyi edindi ve kullandı. Ama o bilginin neyi temsil ettiğini, yani temelinde neler olduğunu bilmiyor. Spring ile XML bazlı deklaratif transaksiyon yönetimi konfigürasyonu örneğinde 3 satırlık copy/paste edilen XML konfigürasyonunun altında AOP (Aspect Oriented Programming) ve Proxy tasarım şablonu gibi konseptler var. İki satırlık konfigürasyonu kopyaladım, ama temelinde yatan bu bilgileri kopyalayamadım. Bunlar internette kalmaya devam etti. Benim kopyaladığım cansız, iki satırlık kod parçası.

Kod örneğiyle, kodun temelindeki konseptlerin zahmetsiz bir şekilde beyne nasıl kopyalandığını Matrix filmede gördük. Neo anında Karate-Kid olmuştu. Bizim böyle bir imkanımız yok. Böyle bir şeyin gelecekte mümkün olacagını da düşünmüyorum. Geriye kalan, kaynağını bulup, copy/paste yaptığımız bilgilerin temelinde yatan konseptleri ögrenmek ve uygulamak. O zaman hayırsız olarak tabir ettiğim işlemi hayırlı bir işleme dönüştürebiliriz.

Copy/paste zihinlere o kadar yerleşmiş ki, bilgiye sahip olmanın değil de, bilgiye nasıl ulaşabileceğini bilmenin daha önemli olduğunu savunanlar var. Bu artık copy/paste filozofisinin mükemmelleştirilerek, getirildiği en son noktadır, yani bu akımın en tepesindeki noktadır. Bu filozofiye sahip olanların, iş görüşmelerinde bu tür ifadelerde bulunmamalarını tavsiye ediyorum, çünkü iş verenin en sevmediği ve iş görüşmesini anında sonlandırabileceği durumların başında, iş görüşmesi yaptığı şahsın "ufak bir internet araştırması ile bu sorunu çözebilirim" tarzı söylemlerde bulunmasıdır. Böyle bir şey söyleyen şahıs aslında "ben bu bilgiye sahip değilim, başkaları sahip, ama bir bakayım, bakalım, bir yerlerden bulabilir miyim" demektedir. Böyle bir şey söyleyen bir programcıyı siz olsanız işe alır mıydınız?

İyi bir programcı olabilmek için iyi bir temele sahip olmak gerekir. Bu temel edinilmiş ve zaman

içinde edinilen bilgiden oluşur. Programcı tüm yeteneklerini bu temel üzerine inşa eder. Zaman zaman internetten bir şeyleri copy/paste etmek suç ya da günah değildir. Bu hepimizin yaptığı bir şeydir. Lakin programcı copy/paste yaptıktan sonra, copy/paste yaptığı kod parçasının temelinde yatan bilgiyi sahip olduğu temele eklemekle yükümlüdür. Aksi taktirde bu bakkaldan bir sakız alıp, parasını ödememek gibi bir şey olur. Copy/paste yapılan şeyin bedeli ödenmelidir. Copy/paste edilen kodun üzerinde fiyat etiketi olmadığı için beleş sananlar vardır. Hayır, beleş değildir. Bedeli ödenmeden kullanılan bilgi, temeli sağlam olmayan bir binanın en ufak bir rüzgar esintisinde sallanması misali programcıya zarar verir.

Copy/paste yapalm, ama bedelini ödeyerek...

Test Edebilme Uğruna Her Şey Mübahtır!

http://www.kurumsaljava.com/2012/11/13/test-edebilme-ugruna-her-sey-mubahtir/

Geçenlerde yine tartışması yapılıyor: *private olan metotları nasıl test ederiz?* Benim cevabım: *edemeyiz!* Karşıdan gelen cevap: *dediğim gibi, her şeyi test etmek mümkün değil demek ki!.* Benim cevabım: *her şeyi test etmek mümkün, private'i protected yaparsın, olur, biter.* Karşı tarafın cevabı: *kardeşim ortada OOP diye bir şey var, kafana göre nasıl öyle private bir metodu proteted yaparsın?*. Yaparım canım kardeşim. OOP'yi filan takmam! Protected'de yaparım, public'de. Bir sınıfı test edebilmek için her türlü yöntemi kullanırım, gözünün yaşına bakmam. Bu uğurda her şey mübahtır.

Bu bazıları için çok radikal bir yaklaşım tarzı gibi görünebilir. Ama uygulama çalışırken bir sebepten dolayı patladığı zaman kimse gelip size, bu sınıfı neden OOP'ye uygun tasarlamadınız diye sormaz. Bu sınıf neden patladı, sizin testleriniz yok mu diye sorar! O zaman da "bu metot private idi, o yüzden test yazamadım" mi diyeceksiniz? Pek ikna edici değil, değil mi?

Bir uygulama geliştirken bir kod biriminin test edilebilirliğinin önündeki engel OOP prensipleri olmamalıdır. Eğer öyle ise, OOP prensipleri yanlış uygulanmış demektir. OOP kod geliştirmek için kullanılan bir araçtır. private yerine göre kullanılabilecek bir erişim mekanizmasıdır. Lakin private olan bir metot benim için yoktur, yani test edilmesi mümkün değildir. Benim için test edilemeyen kod birimi olamayacağı için, private ile tanımlanmış metotları yok sayarım. Test edilemeyen bir metodun başına her türlü iş gelebilir. Benim bu tür metotları görünce ilk yaptığım şey, metodu hemen protected yapmak ve o metodu izole bir şekilde test eden bir birim testi yazmak olur. Protected yetmedi ise public yaparım. Nihai amaç metodu bir şekilde test edebilmektir.

Ne test edilir, ne test edilmez, nasıl test edilir tartışmalarının sonu gelmez ne yazık ki. Geçenlerde yine kocaman, ne yediği belli olmayan bir sınıfa yeni bir özellik eklemem istendi. Eğer gerekli kodu o sınıfa direk ekleseydim, yeni eklediğim kod birimini test etmem çok zor olacaktı, bunu biliyordum. Sınıfın tümünü test edecek bir birim testi oluşturmak için vaktim yoktu. Bende bunun yerine yeni bir static iç sınıf oluşturup, kodu bu sınıfın bir metodu olarak geliştirdim ve bir birim testi ile test ettim. Testin başarılı olduğunu gördükten sonra ne yediği belirsiz diye tabir ettiğim sınıfın gerekli metoduna oluşturduğum yeni sınıfın bir nesnesini yerleştirerek, static sınıfın metodunu orada koşturdum. Böylece ne yediği belirsiz sınıf dolaylı bir şekilde yeni oluşturduğum kod birimine sahip oldu. Doğal olarak gelen ilk soru şu oldu: "neden static bir iç sınıf oluşturdun ki? Orada bu işi görecek bir metot vardı, o metoda kendi kodunu ekleyebilirdin". Eklerdim, eklemesine, lakin bu sınıfa eklediğim kodu nasıl test edebilirdim? Edemezdim. Sınıf için daha önce hiç birim testi yazılmamıştı. Sınıfa eklediğim kod birimini test edebilmek için tüm uygulamayı ayağa kaldırıp, benim eklediğim kod birimi çalışıyor mu diye otomatize edilemeyecek türden bir test yapmam gerekirdi. Yok efendim. bu tür yöntemler sınıf enflasyonuna sebep oluyormuş. Olursa, olsun, sorun nedir? Önemli olan benim yeni kod birimini izole bir şekilde test etmemdi.

OOP, tasarım şablonları ya da tasarım prensipleri sadece test edilebilirliğin emrinde çalışabilecek erlerdir, tersi değil! Bu saydığım araç, gereçlerin hepsinin efendisi test edilebilirlik özelliğidir.

Bunların arkasına saklanıp, bu kod birimi test edilemez diyen yazılımcı kendisine ve müşterisine ihanet içinde olur.

Deneme Yanılmanın Bedeli

http://www.kurumsaljava.com/2012/11/09/deneme-yanilmanin-bedeli/

Yazılım yaparken en büyük zaman kaybının nedeni, kodu değiştirip, derleyip, çalışır hale getirdikten sonra değişikliğin sonucunu test etmektir. Kodu değiştir/derle/çalıştır/dene süreci otuz saniyeden beş dakikaya kadar sürebilir. Günde bunu on sefer yaptığınızda bir saatlik bir zamanı boşa harcamış olursunuz. Bu sebepten dolayıdır ki EJB2 ve benzeri teknolojilerin yerine Spring gibi daha hafif yazılım yapmayı sağlayan çatılar (framework) oluşmuştur.

Çoğu programcı yılmadan kodu değiştir/derle/çalıştır/dene döngüsünde programlamaya devam ediyor. Büyük bir inatla, akıl almaz derecede komplike olan algoritmaları deneme, yanılma usulü ile geliştirmeye ya da değiştirmeye çalışanlar var. Neymiş efendim, bir sonraki döngüde çalışacakmış... Murphy'nin kuralları gereği yapılan değişikliğin beklenen neticeyi getirmeme ihtimali çok yüksek. Ve böylece programcı hiç farkına bile varmadan dönme dolap içinde dönüp, duran bir fare misali hayatını devam ettirir....

Aslında böyle durumlarda beyinde tehlike çanları çalmaya başlayıp, programcının içinde bulunduğu kısır döngüyü aşması için gerekli rutinlerin çalışması gerekir. Mesela bu rutinlerden bir tanesi "hemen kodu test eden bir birim testi yaz" olabilir. Bir birim testi bahsi geçen kod birimini izole edilmiş halde çok hızlı bir şekilde test etmek için kullanılabilir. Yapılan değişikliğin neticesini görmek bir saniye bile sürmeyecektir. Gerekli değişiklikler yapıldıktan sonra ayrıca bonus olarak bir birim testine sahip olunur. Daha sonra yapılması gereken değişiklikler birim testi sayesinde çocuk oyuncağı haline gelir. Zaman ayrılıp, mutlaka böyle bir yatırım yapılmalıdır, aksi taktirde deneme, yanılma tarzı programlamanın bedeli çok daha ağır olacaktır. Sözüm tabi verimliliğim düştü derdinde olmayanlara değil.

Hemen test yaz vari bir rutinin devreye girebilmesi için iki şeye ihtiyaç duyulmaktadır: - Böyle bir rutine sahip olmak. - Rutini zamanı gelince otomatik olarak tetiklemek.

Bu rutini çalıştırabilmek için bu rutine sahip olmak gerekiyor. Böyle bir rutine nasıl sahip olunabileceğine Kod Kata ve Pratik Yapmanın Önemi başlıklı yazımda değinmeye çalıştım. Bu işin sırrı devamlı pratik yapmaktan geçiyor. Pratik sahibi olan bir programcının aklına ilk gelen şey **ya ben neden bir birim testi yazmıyorum da, burada taklalar atıyorum** olur. Basit gibi görünen bu beyin performansını sağlayabilmek için beyni bu yönde eğitmek gerekiyor. Bunun yolu da kod katalarından geçiyor. KodKata.com'a bir göz atmanızı tavsiye ederim.

Deneme, yanılma tarzı programlama hem insanı yorar, hem usandırır, hem de bir zaman sonra bıkkınlık verir. Ayrıca bu tarz programlamanın yazılım mühendisliği ile yakından, uzaktan ilgisi yoktur. Mühendis olmak demek yol, yordam bilmek demektir, metotlu, yöntemli çalışmak demektir. Çaylak gibi bilgisayar başına geçip, deneme, yanılma tarzı programlama yapan yazılımcıya gülerler. Profesyonel programcı hemen çekmecesinden ihtiyaç duyduğu aracı, yani birim testi yazma yetisi, çıkarır ve sorunu en kısa sürede daha önce etkinliliği milyarlarca defa kanıtlanmış bir yöntemi kullanarak çözer. Yol, yordam bilmek budur. Profesyonel yazılımcıya yakışan budur.

Veresiye Satan Yazılımcı

http://www.kurumsaljava.com/2012/09/17/veresiye-satan-yazilimci/

Editörü açtınız, public class yazarak yeni bir sınıf oluşturdunuz. Bu sınıfa yeni bir metot eklediniz. Başka bir şey yapan yeni bir metot daha eklediniz. Sınıf yavaş yavaş şismeye başladı. Birkaç refactoring yaptınız. Buradan yeni bir sınıf dünyaya gözlerini açtı. Bir sınıf, bir sınıf daha derken sınıf sayısı onlara ulaştı. Her sınıfın metot küfesi iyice ağırlaşmaya başladı. Göz açıp, kapayana kadar birkaç bin satır kod oluştu. Program istediğiniz şekilde çalışıyor sanırım. Yaptığınız işten memnunsunuz, ama içinizden bir ses bir şeylerin doğru gitmediğini söylüyor. Eksik olan ne? Testler!

Yazılımcı neden test yazmaz? Sıralayalım:

- Uygulamayı test etme konseptlerinden bihaberdir.
- Test etmeyi gerekli ve işinin bir parçası olarak görmez.
- Test etmeyi angarya ya da zaman kaybı olarak görür.
- Geliştirdiği uygulamanın test edilemez olduğunu düşünür.
- Test etmek için zamanı yoktur.
- Daha sonra test yazarım diyerek kendini avutur.
- Testlerin yazılımcı olmayan şahıslar tarafından yazılmasını sağlamak için lobi çalışması yapar.
- Testlerin devamlı bakıma ihtiyacı olduğunu ve bu yüzden verimli olmanın önünde engel olduklarını düşünür.

Bir yazılımcının görevi nedir? Sıralayalım:

- Altına tereddüt etmeden ismini yazıp, imzalayacağı, çalışır, hatasız ve kaliteli bir sürüm ortaya koymak.
- ve daha birçok başka şey.

Yazılımcı test yazmadığı zaman ne olur? Sıralayalım:

- Devamlı oluşan hatalarla boğuşur.
- Kodu yeniden yapılandırma cesaretine hiçbir zaman sahip olamaz.
- Yeni müşteri gereksinimlerini kodlamakta zorlanır, çünkü yeniden yapılandırılamayan kodun değiştirilmesi güçtür.
- Sürüm çıkardıktan sonra birkaç gece uyuyamaz.
- Hataları gidermek için fazla mesai yapmak zorunda kalır.
- Uygulamanın her yeni sürümü ile hata sayısı çoğalır, kod kalitesi düşer.
- Yaptığı işten zevk almamaya başlar.
- İşini tam olarak yapmadığını bildiği için vicdan azabı çeker.

Veresiye satan tüccar ile peşin satan tüccarın hikayesini bilirsiniz. Veresiye satanın sonu malumdur. İşi başından sıkı tutmadığı için batma aşamasına gelir. Test yazmayan programcının da durumu aynıdır. Projenin geleceğini veresiye dağıtır.

Siz veresiye mi satıyorsunuz, peşin mi?

Uzman ve Usta Yazılımcı Arasındaki Fark

http://www.kurumsaljava.com/2012/09/17/uzman-ve-usta-yazilimci-arasindaki-fark/

Fanatik futbol severleri bilirsiniz. Takımları için yapmayacakları yoktur. Bu fanatiklik başka insanlara zarar vermeye kadar varabilir. Yazılımda da durum farklı değildir. Tek fark bu fanatikliğin insana zarar verecek seviyede olmamasıdır.

Çok duymuşsunuzdur Java dili şöyle, Java dili böyle, diğer dillerden üstündür... diye. Neden bazı yazılımcıların böyle fanatizm olarak görülebilecek bir ilgi ve alaka ile bazı bilişim öğelerine bağlandıklarını biliyor musunuz? Bu yazımda bu soruya cevap vermeye çalışacağım.

Bilişim ya da yazılımda fanatik olanlar savundukları konu hakkında uzman ya da uzman olduklarını düşünen şahıslardır. Yıllar süren çalışmalar sonunda örneğin bir programlama diline uzman seviyesinde hakimdirler. Sahip oldukları fanatizm ile savundukları konu hakkında yıllar süren çalışmalar ardından yoğun bilgi, tecrübe ve beceri sahibi olmuşlardır. Kısacası bu konuda uzmanlaşmışlardır. Fanatik bir biçimde kendilerini ifade etmeye çalışmalarının altında gizli olan iki şey vardır: Birincisi sahip oldukları bilgi, tecrübe ve beceriyi ortaya koymak istemeleri, ikincisi farkında olmadan yeniliklere açık olmadıklarını ifade etmeleri.

Bir konuda uzmanlık ne yazık ki çok uzun sürebilecek bir bağımlılığı beraberinde getirebilir. Bu insanda zaman içinde sahiplenme hissi doğurur. Çok emek harcayarak bir yerlere geldiğini düşünen şahıs için savunduğu konu kutsallaşır. Bu durum savunma ve koruma güdülerini tetikler. Buradan da tanıdığımız ve hoşumuza gitmeyen verimsiz tartışmalar doğar. Belli bir programlama dilinin fanatikçe savunulduğu tartışmaların kimseye bir şey katmayacağını çok iyi biliriz. Ama bu fanatizm yine de bir son bulmaz. Devam eder gider, çünkü yazılımcı uzman olmaya devam eder, çünkü usta olmanın ne olduğunu kavrayamamıştır.

Tipik bir uzmanın bir çalışma ömrü boyunca aynı programlama dilini kullandığını görmek mümkündür. Uzmanlık sıfatı ne kadar olumlu görünsede, bahsettigim sebeplerden dolayı sakınılması gereken bir durumdur. Bir uzman yazılımcı at gözlüğü takmışcasına hayatın içinden geçer, gider, kişisel gelişim için gerekli nimetleri edinemeden. Kendisini bir konuya adadığı için, bu konunun modası geçtiğinde ortada kalıverir. Hakim olduğu konu haricinde yeni bir şeyler öğrenme yetilerini geliştirmeği için iş hayatının son bulması teklikesi ile karşı karşıya kalabilir. Buraya kadar yazdıklarımdan bir konuda uzman olmaya sıcak bakmadığımı düşünebilirsiniz. Bir değil, birden fazla konuda uzmanlığı tercih ederim. Bunun yazılımdaki karşılığı uzman değil, usta yazılımcı olmaktır.

Usta bir yazılımcının öz geçmişine göz attığınızda, onlarca programlama dilini kullandığını görürsünüz. Hepsinde uzman olup, olmadığı tartışılır. Lakin bu onun uzmana nazaran değişikliklerle daha iyi yaşayabildiği anlamına gelmektedir. Zaman ve mekan neyi gerektiriyorsa, o konuyu seçerek, konu üzerinde çalşır ve zamanla uzmanlaşır. Geniş bir perspektife sahip olmasının sebebi buradan kaynaklanmaktadır. Çok değişik dil, teknoloji ve platformlarla çalışmak daima onun ufkunu genişletir. Bu onun günlük işlerine olumlu olarak yansır. Bu ona pragmatik olmayı öğretir. Bu ona teknoloji fanatiği olmamayı öğretir. Uzman ve usta yazılımcı arasındaki fark budur!

Göz Boyamaca

http://www.kurumsaljava.com/2012/08/15/goz-boyamaca/

Size herhangi bir şey satmak için kapınıza gelen bir satış temsilcisine ne gözle bakarsınız? Ben bu durumla karşılaşınca, beynimde otomatik olarak şahsı başımdan savma rutinleri devreye giriyor. Birilerinin benim üstüme bu şekilde gelerek bana bir şey satması imkansız. Bu inisiyatifin benden kaynaklanması lazım.

Yazılım sektöründe de durum anlaşılan farklı değil. Birileri biz yazılımcılara devamlı bir takım, ne işe yaradıkları belirsiz sertifikalar satmaya çalışıyor. Tek dertleri para kazanmak. Kullandıkları ve insanları en çok etkileyen argümanları ise yazılım kalitesini artırmak ve eleman yetiştirmek için çaba sarfediyor oluşları. Yesinler! Ben bunu ispatlayan ne bir araştırma ne de bir rapor gördüm.

Bir gün içinde alınan sertifikalar var. Artık para basma makinasının sesi o kadar yükselmiş ki utanıp, sıkılıp, hadi daha fazla dikkat çekmeden para basmaya devam edelim ama sertifikayı vermeden önce çakma bir imtihanla sertifikanın alınmasını zormuş gibi gösterelim diyenler ve bunu uygulayanlar var. Daha önce herkese para karşılığında dağıttıkları sertifikaları şimdilerde dandik bir imtihanla vermeye devam ediyorlar. Ne kadar büyük bir gelişme! Artık yazılım sektörü bu sertifikalarla iş başına gelen yazımcıların meydana getirdikleri yazılım kalitesinde boğulup gidecek. Bu sertifika satıcıları bana cennetten tapu satanları hatırlatıyor. Bir gün içinde sertifika almış ve karşıma geçip bana çevik süreçlerin nasıl işlediğini anlatmaya çalışan birisini ben ciddiye almıyorum.

Peki yazılımcılar neden böyle şeylere aldanıp, zor kazandıkları paralarını pencereden dışarı atıyorlar? Ha ben gidip bir günlük bir eğitim almışım, akabinde bir test ile bana sertifika verilmiş, ha bu adamlar benim kapıma gelmişler ve sertifikayı bana kapıda satmışlar. İkisi de aynı şey değil mi? Neden bunlara aldanıyor insan?

Devir ne yazık ki göz boyama devri. İş görüşmelerinde sertifikasyona gereğinden çok fazla önem veriliyor. İşveren eğer bu programcı A, B, C sertifikalarına sahipse, iyi bir programcı olmalı diye düşünüyor. Ne kadar yanlış! Bu sadece programcının bir yolunu bulup, sertifikayı aldığını gösterir. Ne kadar iyi bir programcı olduğunu göstermez.

Çok gezen mi çok bilir, çok okuyan mı? Çok sertifika sahibi mi çok bilir, çok pratik yapan bir programcı mı?

Bir programcıyı iş başına getirme kriteri sahip olduğu sertifika kolleksiyonun genişliği olmamalı. Bu sertifikalar programcının sahasında ne kadar iyi olduğunun göstergesi olamaz. Bir programcının iyi olup olmadığını anlamak için, onunla bir gün boyunca çalışıp, hangi yeteneklere ve bilgi seviyesine sahip olduğunu anlamak zor değil. Buna alternatif olarak işveren programcının eski çalışma arkadaşlarından referans getirmesi isteyebilir. Eski patronundan referans demedim, eski mesai arkadaşlarından dedim. Programcılar birbirlerini tanır, karşılıklı yetenek, beceri ve bilgi seviyelerini tartabilirler. Hiçbir programcı, iyi bir seviyede olmayan bir programcı için iyi bir programcı diyerek, kendi itibarını zedelemek istemez. Bu yüzden programcının eski mesai arkadaşları her zaman onun hakkında bilgi edinmek için

kullanılabilecek güvenli bir kaynaktır.

Bizim yazılımcı olarak bu sertifika sövüşlemesine gelmeden, kendimizi yazılımcı olarak nasıl geliştirebiliriz sorusuna cevap aramamız gerekir. Eğer programcı mutlaka bir sertifika sahibi olmak istiyorsa, o sertifikayı edinsin. Ama kendisi için yapsın, etrafta A,B,C sertifikalarım var diyerek, böbürlenmek için ya da daha kolay iş bulurum düşüncesiyle değil.

Ambalajın güzelliğine aldanmayalım. Ambalajın içinden ne çıktığı önemli. Sertifikaların çok büyük bir kısmı güzel ambalajdan başka bir şey değil. Bunu Oracle Certified Master, Java EE Enterprise Architect (SCEA 5) sertifikası sahibi olarak söylüyorum.

Detayları Görebilmek Ya Da Görememek

http://www.kurumsaljava.com/2012/08/10/detaylari-gorebilmek-ya-da-gorememek/

Robert Martin Transformation Priority Premise – Transformasyon Öncelik Teorisi başlıklı yazısında yeniden yapılandırmaya (refactoring) karşı yöntem olan kod transformasyonuna değiniyor. Refactoring uygulamanın dışa gösterdiği davranış biçimini değiştirmeden kodun yeniden yapılandırılması anlamına gelirken, kod transformasyonu uygulamanın davranış biçimini yeniden şekillendirmek için kullanılan bir yöntemdir.

Benim kod transformasyonlarını bilinçli bir şekilde algılamam kod kataları yapmaya başladığım zamana denk gelir. Daha önce de test güdümlü yazılım yaparken kod transformasyonlarına şahit olmuştum, lakin kod kataları ve Robert Martin'in yazısı aracılığı ile transformasyon sürecinin nasıl işlediğini daha iyi algılama fırsatı buldum.

Bu konuda lafı fazla uzatmadan kod transformasyonunun ne olduğunu bir örnek üzerinden sizlerle paylaşmak istiyorum. Aşağıda yer alan kod prime factor katasından alıntıdır. Bu kata bünyesinde herhangi bir rakamın asal sayı (prime number) olan faktörleri hesaplanmaktadır. Bir asal sayı sadece bire ve kendisine bölünebilir. 2,3,5,7,11,13 asal sayılardır. Örneğin 10 rakamı 2 ve 5 asal sayı faktörlerinden oluşur. 6 rakamının asal sayı faktörleri 2 ve 3'dür. Kata bünyesinde oluşturulan generate() metoduna kod transformasyonları aracılığı ile herhangi bir rakamı oluşturan asal sayı faktör listesini oluşturabilecek kabiliyet kazandırılır. Kata için oluşturduğum birim testi aşağıda yer almaktadır:

```
package com.kodkata.kata.primefactor.kata;

import static com.kodkata.kata.primefactor.kata.PrimeFactorz.
            generate;
import static java.util.Arrays.asList;
import static org.junit.Assert.assertEquals;

import java.util.Arrays;
import java.util.Collection;
import java.util.List;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;

@RunWith(Parameterized.class)
public class PrimeFactorTest {

    private List<Integer> list;
    private int expected;

    public PrimeFactorTest(List<Integer> list,
                int expected) {
        this.list = list;
        this.expected = expected;
    }

    private static List<Integer> list(
            Integer... integers) {
        return Arrays.asList(integers);
    }

    @Test
    public void generate_as_expected()
                throws Exception {
        assertEquals(list, generate(expected));
    }

    @Parameters
    public static Collection<Object[]> testData()
                throws Exception {

       return asList(new Object[][] { { list(), 1 },
                { list(2), 2 },
                { list(3), 3 }, { list(2, 2), 4 },
                { list(5), 5 },
                { list(2, 3), 6 }, { list(7), 7 },
                { list(2, 2, 2), 8 },
                { list(3, 3), 9 }, { list(2, 5), 10 }, });
    }
}
```

Bu parametrik JUnit test sınıfı testData() bünyesinde yer alan veri listesini kullanarak, generate*as*expected() test metodunu koşturmaktadır.

İlk birim testi { list(), 1 } veri kümesi ile 1 rakamı için yapılmaktadır. 1 rakamının asal sayı faktörü yoktur, yani generate() metodu boş bir listesi geri döndürmelidir. Bu testi oluşturduktan sonra aşağıdaki şekilde generate metodunu oluşturuyorum.

```
public static List<Integer> generate(int i) {
    return null;
}
```

Burada ilk kod transformasyonunu görmek mümkün. Ortada hiçbir kod yokken, bir metot oluşturdum ve metodu null değerini geri döndürecek şekilde yapılandırdım. Robert Martin atılan bu ilk adımı ({}–>nil) şeklinde tanımlıyor.

Testin geçebilmesi için kodu aşağıdaki şekilde dönüştürmem (transform) gerekiyor:

```
public static List<Integer> generate(int i) {
    return Collections.EMPTY_LIST;
}
```

Bu dönüşüm (nil->constant) şeklinde ifade edilebilir. Birinci testin geçebilmesi için yapmamız gereken sadece boş bir listeyi (constant) geri döndürmektir.

İkinci test için { list(2), 2 } veri kümesini kullanıyorum. generate() metodu 2 rakamını parametre olarak aldığında, bünyesinde 2 rakamı olan bir listeyi geri döndürmeli. Bu testin geçebilmesi için kodu aşagıdaki şekilde dönüştürmem gerekiyor.

```
public static List<Integer> generate(int i) {
    List<Integer> primes = new ArrayList<Integer>();
    if (i > 1)
        primes.add(2);
    return primes;
}
```

Bu dönüşüm (unconditional->if) şeklinde ifade edilebilir. Testin geçebilmesi için en basit olan çözümü seçtim. Bu durumda en basit çözüm if kullanarak 2 rakamını listeye eklemek. Tetiklediğim iki kod transformasyonu iki testin geçebilecek seviyeye gelmesini sağladı.

Üçüncü test için { list(3), 3 } veri kümesini kullanıyorum. generate() metodu 3 rakamını parametre olarak aldığında, bünyesinde 3 rakamı olan bir listeyi geri döndürmeli. Bu testin geçebilmesi için kodu aşağıdaki şekilde dönüştürmem gerekiyor.

```
public static List<Integer> generate(int i) {
    List<Integer> primes = new ArrayList<Integer>();
    if (i > 1)
        primes.add(i);
    return primes;
}
```

Bu dönüşüm (constant->scalar) şeklinde ifade edilebilir. Testin geçebilmesi için bir constant olan 2 rakamını i (scalar) parametresi ile değiştirdim.

Dördüncü test için { list(2,2), 4 } veri kümesini kullaniyorum. generate() metodu 4 rakamını parametre olarak aldığında, bünyesinde 2,2 rakamları olan bir listeyi geri döndürmeli. Bu testin geçebilmesi için kodu aşağıdaki şekilde dönüştürmem gerekiyor.

```
public static List<Integer> generate(int i) {
    List<Integer> primes = new ArrayList<Integer>();
    if (i > 1) {
        if (i % 2 == 0) {
            primes.add(2);
            i /= 2;
        }
        if (i > 1)
            primes.add(i);
    }
    return primes;
}
```

Bu dönüşüm (unconditional->if) şeklinde ifade edilebilir. Testin geçebilmesi için yeni bir if bloğu içinde ikinin katlarını bulan modülo işlemini gerçekleştirdim. i/=2 ile i'nin değerini ikiye bölmüş oldum. Bunun yanısıra ikinci bir if bloğu içinde arta kalan değeri birden büyük ise listeye ekledim.

5,6 ve 7 rakamları için generate() metodunda değişiklik yapmaya gerek yok. generate() metodu bu rakamlar için geçerli listeleri geri döndürüyor.

8 rakamının asal sayı faktörlerini bulmak için { list(2,2,2), 8} veri kümesini kullanıyorum. Bu testin geçebilmesi için kodu aşağıdaki şekilde dönüştürmem gerekiyor.

```
public static List<Integer> generate(int i) {
    List<Integer> primes = new ArrayList<Integer>();
    if (i > 1) {
        while (i % 2 == 0) {
            primes.add(2);
            i /= 2;
        }
        if (i > 1)
            primes.add(i);
    }
    return primes;
}
```

Bu dönüşüm (if->while) şeklinde ifade edilebilir. Testin geçebilmesi için ikinci if bloğunu while döngüsüne dönüştürmem yeterli oldu.

9 rakamının asal sayı faktörlerini bulmak için { list(3,3), 9} veri kümesini kullanıyorum. Bu testin geçebilmesi için kodu aşağıdaki şekilde dönüştürmem gerekiyor.

```
public static List<Integer> generate(int i) {
    List<Integer> primes = new ArrayList<Integer>();
    int divisor = 2;
    while (i > 1) {
        while (i % divisor == 0) {
            primes.add(divisor);
            i /= divisor;
        }
        divisor++;
    }
    return primes;
}
```

Bu dönüşüm (if->while) ve (constant->scalar) şeklinde ifade edilebilir. Divisör (bölen) olarak kullandığım 2 rakamını bir değişkene (constant->scalar), ilk if bloğunu bir while döngüsüne dönüştürdüm.

generate() metodu bu son transformasyon ile herhangi bir rakamın asal sayı faktörlerini hesaplayacak duruma gelmiş oldu. Yapabileceğim iki transformasyon kaldı. Bu transformasyonlar ile satır sayısını azaltabilirim. Bu iki tranformasyonu (while->for) olarak tanımlayabiliriz.

```
public static List<Integer> generate(int i) {
    List<Integer> primes = new ArrayList<Integer>();
    for (int divisor = 2; i > 1; divisor++) {
        for (; i % divisor == 0; i /= divisor)
            primes.add(divisor);
    }
    return primes;
}
```

Görüldüğü gibi testler geçtikten sonra metodun davranış biçimini kod transformasyonları uygulayarak yeniden yapılandırmamız mümkün. Uyguladığımız her transformasyon ile oluşturduğumuz çözüm spesifik olmaktan çıkıp umumi olmaya doğru gidiyor.

Test güdümlü çalışmayan bir programcı doğrudan generate() metodunu yukarda gördüğünüz şekilde kodlayacak ve mevcut kod transformasyonlarından bihaber olacaktır. Bu transformasyonları görmek bana ne fayda sağlar diye bir soru aklınıza gelebilir. Bilmiyorum! Belki bir faydası yok. Ama ben artık işin daha derinlerinde başka şeylerin varlığını hissetmeye başladım. Bu bana programcılıkta başka bir boyuta geçmek ya da olup bitenleri daha iyi algılamak için kullanabileceğim bir kapı olabilir gibi geliyor. Bazı şeyler detaylarda gizli. Onları görebilmek lazım.

Yazılımda Çeviklik İflas mı Etti?

http://www.kurumsaljava.com/2012/08/09/yazilimda-ceviklik-iflas-mi-etti/

Çevikliğin Böylesi başlıklı yazımı okudunuz mu? O yazımda çevik kelimesinin yerli yersiz her şey için kullanıldığını ve bu yüzden anlam erozyonuna ugradığından bahsetmiştim. Bu yazımda çevikliğin tanımını yapmaya çalışacağım.

Martin Fowler bu link üzerinden ulaşabileceğiniz yazısında şöyle diyor:

... lack of rigorousness is part of the defining nature of agile methods, part of its core philosophy. ... Any attempt to define a rigorous process that can be tested for conformance runs contrary to [the agile] philosophy.

Martin Fowler yazılımda çevikliğin belli bir çerceveye sokulmuş bir süreç (process) olarak tanımlanmasının mümkün olmadığını, bunun çeviklik filozofi ve düşüncesine ters düştüğünü söylüyor.

Bazı kesimler Martin Fowler'in sözlerini çevikliğin iflası olarak yorumlayabilirler. Onların mutlaka katıksız bir tanımlamaya ihtiyacı var, çünkü ipe sapa gelmeyen bir şeyin pazarlamasını yapmak mümkün değil. Bu yüzden çeviklik filozofisinin sadece bir kısmını ihtiva eden yarım yamalak çevik süreçler tanımlamaya ve satmaya çalışıyorlar. Onlar için önemli olan elle tutulur, gözle görülür ve ölçülebilir bir süreç tanımlaması yapmak. Eğer A, B, C adımlarını uygularsan, X, Y neticesini elde edersin diyebilmek istiyorlar. Keşke müşteri gereksinimleri odaklı yazılım bu kadar basite indirgenebilseydi!

Bu sebeptendir ki birçok sözde çevik süreç ile istenilen neticeler elde edilemiyor, çünkü sürecin ihtiva ettiği adımları birebir uygulamak her projede mümkün olmuyor. Doğal olarak bu durumda sürecin yeniden tanımlanıp, ihtiyaçlar doğrultusunda yeniden yapılandırılması gerekiyor. Sözde çevik süreçler bu değişime izin vermedikleri için yazılımda çeviklik yarı yolda kalıyor.

Benim Çeviklik Tanımım

Ben yazılımda çevikliği bir soğanın kabukları gibi iç içe geçmiş, değişik katmanlardan oluşmuş bir yapı olarak görüyorum. En üst katmanda değişikliklere karşı koymadan, onlarla beraber yaşayabilmek için gerekli olan cesaret yer alıyor. Yazılımda çeviklik cesaret ile başlar ve en içteki katmana kadar varlığını sürdürür. Değişiklikle yaşamak ve yeni yollar, yöntemler keşfetmek için birey ve ekip bazında casaret gereklidir. Bunun olmadığı yerde sadece sözde çeviklik olabilir.

Sözde çevik süreçlerde cesaretin yerini sabit süreç tanımlamarı alır. Bunlar tek tek ve sırayla atılması gereken adımlardır, bir nevi yemek tarifidir. Değişikliğe bu şekilde bilinçaltı karşı konulmaya ve her şey daha kontrol edilir hale getirilmeye çalışılır. Yazılım projelerinde değişmeyen tek parametrenin değişikliğin kendisi olduğunu düşünecek olursak, sabit süreç tanımlamaları ile buna karşı koymanın imkansız olduğu ortadadır. Sabitlik isminden de anlaşıldığı gibi dinamizmin karşıtıdır ve ani değişikliklere cevap verebilecek nitelikte ya da

değişikliklere adapte olacak yapıda değildir. Değişikliğe karşı koymanın tek yolu, ona karşı koymamak ve onunla beraber yaşamayı ögrenmekten, bunun yolu da cesaret ve cesaretli olmaktan geçer.

Casaret aynı zamanda müşteriye devamlı ne istediğini sorabilmektir. Bu amaçla yazılımda çeviklik çok kısa döngülerin doğurduğu geribildirimlerden beslenir. Bu döngülerin başında proje gidişatını yönlendirmek için kullanılan iterasyon bazlı planlama yer alır. Değişikliğin her an damdan düşer gibi projeyi alt üst edebilecegini varsaydığımızda, buna karşı koymanın en kolay yolu, projeyi kısa ve tanımlanmış zaman dilimlerinde müşterinin gereksinimlerini tatmin edecek seviyeye getirmektir. Bu örneğin bir ya da iki haftalık bir zaman dilimini ihtiva eder. Bu zaman diliminde müşteri kendisi için önemli olduğunu düşündüğü gereksinimlerini tespit edip, gerçekleştirilmek üzere yazılım ekibiyle paylaşır. Yazılım ekibi müşterinin seçmiş olduğu gerekli gereksinimleri hayata geçirir. Ekip seçilen gereksinimlerin öngörülen zaman diliminde tamamlanır tarzda olmasına dikkat eder. Öngörülen zaman dilimi tamamlandığında müşteriye çalışır bir prototip sunulur ve kendisinden geribildirim sağlanır. Bu noktatan itibaren müşteri gidişati kontrol edebilir bir araca sahiptir: çalışan bir prototip. Çoğu zaman müşteri bu prototipe bakarak sahip olduğu gereksinimleri doğru olarak tanımlamadığının farkına varır. Bu kendisi ve yazılım ekibi için bir geribildirimdir ve değişikliğin habercisidir. Bir sonraki çalışma safhasında (iterasyon) ekip bu geribildirimden beslenerek, müşterinin gereksinimlerini tam anlamıyla tatmin edecek değişikliklere gider.

Geribildirim alma çevikliğin ana temellerinden birisini teşkil etmektedir. Müşteriye ne istediğini sorarak, iterasyon bazlı çalışıp prototipler oluşturarak, sürekli entegre edip, entegrasyon seviyesinin ne durumda olduğunu sorgulayarak, birim testleri yazıp, uygulamanın test edilbilirliğini ölçerek birçok katmanda geribildirim ediniyoruz. Bu bizim yazılımcılar olarak müşteri gereksinimlerinin ifade edilmesi ile yazılımcılar tarafından tatmin edilmesi arasında nerede durduğumuzu anlayayıp, tartabilmemiz için çok önemli bir veri olma özelliği taşımaktadır.

Teknik olarak yazılımda çevikliği incelediğimizde çevik olmanın çekirdeğinde çok önemli bir metodun yer aldığını görmekteyiz: uygulamayı yeniden yapılandırma. İngilizce refactoring olarak isimlendirilen bu yöntem, biz yazılımcılara yeni müşteri gereksinimleri ile uygulamayı hamur gibi yogurma yetisi kazandırmaktadır.

Birçok sözde çevik süreci ya da çevik süreçlerin uygulandığı projeleri yakından incelediğimizde, başarısızlıklarının sebebinin uygulamayı hamur gibi yoğurma kabiliyetlerinin olmadığından kaynaklandığını söylemek mümkündür. Bazı sözde süreçler uygulamanın yeniden yapılandırılabilme özelliğine oluşturdukları gidişat planında yer bile vermemektedirler. Bir uygulamanın müşteri gereksinimlerine cevap verebilmesi için yeniden yapılandırılabilmesi gerekmektedir. Bunun ne olduğunu bile bilmeyen ya da tanımlayamayan bir çevik süreç nasıl çeviklik ibaresi olma savına sahip olabilir? Bunu anlamak güç!

Belki çevikliği tanımlamak kolay değil. Belki bu sebepten dolayı herkes her şeye çevik ismini takabiliyor. Ama bildiğim bir şey var, o da birim testi yazmadan ve sürekli refactoring yapmadan çevik olunamayacağıdır. Bu ikisinin nasıl yapılması gerektiğine dair birçok kaynak bulmak mümkündür. Martin Fowler'in Refactoring isimli kitabı yazılımcılara bu konuda ilham kaynağı olacaktır. Refactoring'i mümkün kılmak için birim testleri yazılması gerekmektedir. Bunu isteyen test güdümlü (Test Driven Development), isteyen klasik tarzda uygulamayı kodladıktan sonra yapabilir. Test güdümlü yazılımı şiddetle tavsiye etmekle beraber, klasik birim test yazılımını da hor görmüyorum. Uygulamayı hamur gibi yoğurabilmek için otomatik çalışan her türlü birim testi mübahtır. Bu birim testleri yeniden yapılandırma sürecinde oluşan hataları yazılımcıya gösterirler. Bunun yanısıra yazılımcının özgüvenini artırarak, daha cesaretli bir şekilde yeniden yapılandırma sürecine dahil olmasını sağlarlar. Birim testleri olmadan mantıklı çercevede uygulamanın yeniden yapılandırılması mümkün değildir.

Müşteri gereksinimlerini tatmin etmek için oluşturulan bir uygulama, üç katlı bir apartman gibi sabit bir yapı değildir. Uygulama devamlı değişiklige maruz kalır, çünkü müşteri kendi çalışma sahasında değişikliğe maruz kalmaktadır. Bunun doğal olarak uygulamaya yansıması gerekmektedir. Aksi taktirde uygulama müşterinin güncel gereksinimlerini tatmin edemez. Müşterinin gereksinimlerini tatmin edemeyen bir uygulama, işe yaramayan bir uygulamadır.

Demek oluyor ki "Birim testi yoksa refactoring yok. Refactoring yoksa çeviklik yok. Çeviklik yoksa, müşteri yok (olur)". Benim çeviklik konusundaki formülüm bu.

Kişisel Gelişim

http://www.kurumsaljava.com/2012/07/26/kisisel-gelisim/

Bir programcının tipik bir haftasının kırk saati patronu için çalışmakla geçer. Daha önceki bir yazımda bir programcının mesai saatlerinde kendisini geliştirmesinin mümkün olmadığını, ama mesai haricinde pratik yaparak programcılık yeteneğini geliştirmesi gerektiğinden bahsetmiştim. Gel gelelim günde sekiz saat çalıştıktan sonra eve gelip, pratik yapmak ya da kitap okumak kolay bir şey olmayabilir. Yemek yedikten ve biraz televizyon seyrettikten sonra kişisel gelişim için gerekli motivasyon tabana vurabilir. Bu çok doğal bir şey. Bu sebepten dolayı kimse kendisine kızmamalıdır. Daha ziyade insan kendisine bu kısır döngüyü nasıl kırabilirim sorusunu sormalıdır.

Kişisel gelişim için hiç zaman yokmuş gibi görünsede, bu doğru değildir. Mesai saatlerini çıkardığımızda geriye 128 saat kalmaktadır, yani tam tamına 5.3 gün. Bu zaman dilimine kocaman bir dünyayı sığdirabiliriz, yeterki zaman yönetimini iyi yapabilelim.

Öncelikle kişisel anlamda bize bir fayda sağlamadan geçip giden zaman dilimlerine bir göz atalım. Bu zaman dilimlerini kişisel gelişim için nasıl kullanabiliriz, onu da akabinde inceleyelim.

Eğer programcı binek otosuyla ile işe gidiyorsa, her gün ortamala 1-3 saatini anlamsız bir şekilde israf etmektedir. Programcı eğer mümkünse toplu taşıma araçlarını kullanarak işe gidip, gelmelidir. Bu şekilde yolculuk esnasında bir kitap okuyabilir ya da akıllı telefonu ya da tableti ile ilgilendiği konuda bir sunum seyredebilir. Bu şekilde her hafta bir kitabı bitirmek mümkündür.

Gelelim öğle aralarına. Kimin akıllı bir telefonu yok, el kaldırsın! Evet, herkesin varmış. O zaman akıllı telefonumuzu kişisel gelişim aracına çevirelim. Öğle aralarında bir programcı akıllı telefonu üzerinden ilginç bulup, takip ettiği blogları okuyabilir ya da sektörün önderliğini yapan kişilerin sunumlarını seyredebilir. Mesai arkadaşları ile lak lak yapmak yerine, her gün bir saatlik bir zaman dilimi kişisel gelişim için kullanılabilir. Yemek yerken bir blog okumak ya da sunum seyretmek düşünüldüğünden daha kolaydır.

Gelelim işin zor kısmına. Mesaiden eve döndükten sonra, motivasyon sorunumu nasıl çözerim? Yapılacak işi gözde büyütmeyerek ve küçük parçalara bölerek. Eğer yeni bir kitabı okumaya başladıysam, büyük bir ihtimalle motivasyonum ilk yirmi sayfa için yeter. Kitabın geri kalan beş yüz sayfasını okuyabilmek için motivasyonumu geri getirici bir taktik geliştirmem gerekir. Bunu sağlamak için Promodoro tekniğinden (http://www.pomodorotechnique.com/) faydalanilabilir. Ne yazik ki insanlar en fazla yarım saat, bilemediniz bir saat tam konsantre çalışabilir. Bu zaman dilimi dolduktan sonra beyin ara vermek için bahaneler uydurmaya başlar. Bu bize motivasyon eksikliği gibi yansır, ama sorunu ufak bir ara vererek çözebiliriz. Promodoro tekniğinde planlanan aktiviteler yirmi beş ya da otuz dakikalık birimlere bölünür. Örneğin iki saat ara vermeden kitap okumak yerine, okuma süreci otuz dakika uzunluğunda dört seansa bölünür. Her seans başında bir çalar saat otuz dakika sonra çalacak şekilde ayarlanır ve yapılmak istenen aktiviteye konsantre olunur. Seans süresince sadece seçilen aktiviteye

konsantre olunması gereklidir. Eğer konsantreyi bozan yan etkiler tespit edilirse, bunların seans esnasında bir kenara not edilerek, daha sonra analiz edilmelerinde fayda vardır. Örneğin Promodoro seansında telefon çalıyorsa, bir sonraki seansta telefon kapatılmalıdır.

Her Promodoro seansından sonra beş dakikalık ara verilerek, beyin dinlendirilmelidir. Bu tarz aktivite yapılandırması kişisel gelişim için atılması gereken adımların daha kolay hazmedilmesini sağlar. Kişi her Promodoro seansından sonra verimli bir şeyler yaptığı hissine sahip olarak kişisel gelişimine katkıda bulunacaktır.

Kişisel gelişimi daha verimli ve istikrarlı bir hale getirmek için kisisel gelişim projeleri hazırlanabilir ve uygulanabilir. Bunun Promodoro ve Scrum kombine edilerek nasıl yapılabileceğini değinmek istiyorum.

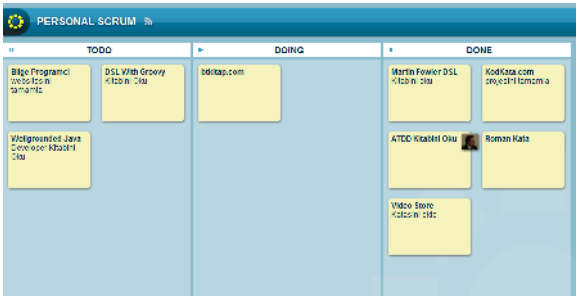
Scrum çevik projeleri yönetmek için kullanılan bir metotdur. Proje bünyesindeki aktiviteler bir ya da iki haftalık Sprint ismi verilen zaman birimlerine bölünür. Sprint öncesi, o Sprint bünyesinde yapılmak istenen aktiviteler tespit edilir. Seçilen aktivitelerin Sprint bünyesinde tamamlanacak nitelikte olması gereklidir. Programcılar tahminlerde bulunarak, Sprint bünyesinde yer alacak aktivitelerin seçilmesine katkıda bulunurlar. Programcılar ayrıca seçilen aktivitelerin Sprint bünyesinde tamamlanmalarından sorumludurlar.

Kişisel gelişimimizi de bir proje olarak el alırsak, bu projeyi organize etmek için kişisel Scrum'dan (personal Scrum) faydalanabiliriz. Scrum bünyesinde gerçekleştirmek istediğimiz aktiviteler, kişisel gelişim aktivitelerimiz olacaktır. Bu aktiviteleri not ederek, bir aktivite listesi oluşturmakta fayda vardır. Bu aktivite listesinde örneğin okunacak kitaplar ve bloglar, yapılacak kod kataları, izlenecek eğitim videoları ve diğer kişisel gelişim için gerekli olduğunu düşündüğümüz aktiviteler yer alabilir.

Aktivite listesini oluşturduktan sonra Sprint planlamasına geçilebilir. Her Sprint belirli bir zaman dilimini kapsar. Bu zaman diliminde üzerinde çalışılacak aktiviteler seçilir ve Sprint başlatılır. Örneğin bir Sprint süresini bir hafta olarak seçti isem, o zaman bu Sprint bünyesinde bu bir haftalık zaman diliminde üstesinden gelebileceğim kadar aktivite tanımlaması ve seçimi yapmam gerekir. Bir haftalık Sprint süresi benim bu bir hafta içinde kişisel gelişimim için ayırmak istediğim zaman dilimini kapsayacaktır. Örneğin her gün iki saat kişisel gelişimim için ayırmayı düşünüyorsam, bir haftalık Sprint toplamda 10 saatlik çalışma planlamasını ihtiva eder (on saat, çünkü beş iş günü).

Seçilen aktivitelerin yarım saat ile en fazla üç saat içinde tamamlanabilecek türde olmasına dikkat edilmelidir. Örneğin üç saat süren bir aktiviteyi bir çırpıda tamamlamak mümkün olamayacağı için, bu tür aktiviteleri yarım saatlik Promodoro seanslarına bölmekte fayda vardır. Söz konusu bir kitabı okunamak olduğunda aktivite planlaması nasıl yapılmalıdır? Bir aktivitenin en fazla üç saat içinde tamamlanabilir yapıda olması gerektiğini yazmıştım. Bir kitap büyük bir ihtimalle üç saat içinde okunamayacağı için, kitabı okuma işleminin alt aktivitelere bölünmesinde fayda vardır. Örneğin kitabın her bir bölümü için ayrı bir aktivite tanımlanabilir. Böylece en fazla aktivite uzunluğu olan üç saat aşılmamış olur.

Sprint ve aktivite yönetimi için bir Kanban panosu kullanılabilir. Benim kullandığın Kanban panosu aşağıdaki şekildedir:



(Bakınız: leankitkanban.com)

Kişisel gelişim için gerekli gördüğümüz aktivitelerin gözümüzün önünde olması ve kaydettiğimiz gelişimi ölçebilir durumda olmamız büyük önem taşımaktadır. Ben örneğin bir Kanban panosu kullanmadan önce kişisel gelişimim için gerekli aktiviteleri aklımda tutmaya ve uygulamaya çalışıyordum. Kısa bir zaman sonra kafamdaki bu listenin aslında bir LIFO (Last In First Out – Son giren ilk çıkar) yapısı olduğunu gördüm. Karşıma devamlı ilgimi çeken yeni kitaplar ve bloglar, uygulamak istediğim yeni teknolojiler ve ilgimi çeken diğer konular çıkıyor. Durum böyle olunca karşılaştığım her şey tepeden aklımda yer alan LIFO listesine yerleşiyor ve listede yer alan diğer konuları aşağıya doğru itiyor. Zaman içinde listenin tepesinde olan aktivitelerin aşağıya doğru kaybolup gittiğini fark ettim. Bu şekilde kişisel gelişimi organize etmek verimli değil. Bunun için bir Kanban panosu kullanmaya başladım. Bu pano bana neyi ne zaman yapmam gerektiğini hatırlatıyor. Kafamdaki LIFO'yu silip, attım.

Kafamdaki LIFO'yu silip atmanın bana sağladığı bir avantajı daha olduğunu gördüm. Kafamdaki aktivite LIFO'su ile yaşadığım sürece devamlı tedirgin olduğumun farkına vardım. Her aktiviteyi akılda tutmak, ne zaman neyi yapmam gerektiğini organize etmeye çalışmak beyin için çok yorucu bir işlem. Beynim devamlı böyle yarım kalmış aktivitelerle haşır neşir ve onların nasıl üstesinden geleceğinin planını yapıyor. Bu yükten kurtulmak için Kanban panosu gibi bir yapıdan istifade edilebilir. Bu web tabanlı bir Kanban panosu olmak zorunda değil. Kağıt üzerinde buna benzer bir planlama yapılabilir. Önemli olan kafadaki LIFO'nun boşaltılması ve aktivitelerin Kanban panosuna eklenmesi. Kafadaki LIFO silindikten sonra beyin kişisel gelişim için daha zinde olacak ve daha verimli çalışacaktır. Bunu hayatın diğer alanlarına da yaymak mümkün. Aklımızda ne kadar LIFO varsa, tüm içeriklerini kağıda ya da Kanban panosuna dökün ve LIFO'lari silin. Ne kadar rahat ettiğinizi göreceksiniz. Ben yorulacağıma kağıt yorulsun :)

Buraya kadar kişisel gelişimimizi nasıl organize edebileceğimizden bahsettim. Şimdi birazda kişisel gelişim için gerekli pratiğin nasıl yapılabileceğine değinmek istiyorum.

Daha önce Kod Kata ve Pratik Yapmanın Önemi isimli bir blog yazdım. Biz programcılar ne yazik ki pratik nedir tam olarak bilmiyoruz. Sabah ofise geliyoruz ve direk kod yazmaya başlıyoruz. Aynı şey müzisyenler ya da sporcular için söylenemez. Bu iki meslek gurubu pratik ve performansı birbirinden ayırt ediyorlar. Sahnede olmak onlar için performans. Sahne aktiviteleri haricinde devamlı pratik yaparak, performansları için hazırlık yapıyorlar. Biz

programcılar sadece performansı tanıyoruz ve pratik yapma yetimiz gelişmiyor.

Bu durumu aşmak için kod kataları yapmamız gerekiyor. Bu amaçla KodKata.com isminde bir web sayfası hazırladım. Kod kataları on beş ila bir saat arası süren kod yazma pratikleridir. Kod kataları hakkında detaylı bilgi için yukarda linkini verdiğim yazımı okumanızı ve KodKata.com'a bakmanızı tavsiye ederim.

Kişisel gelişim için kod katası yapmak çok önemli. Bu sebepten dolayı her sabah işe başlamadan önce bir kata yapmayı alışkanlık haline getirilmesini tavsiye ediyorum. Bu amaçla KodKata.com'dan bir kata seçin ve yapın. Yaptığınız kataları ve adetlerini bir yere not edin. Zaman içinde sağladığınız gelişmeler gözle görülür hale gelecektir. Bir programcının ustalaşmasının sırrı kod katalarında gizlidir.

Çok yazdım, kafanızı şişirdim, kusuruma bakmayın. Kişisel gelişim benim için çok önemli bir konu. Burada yer alan öneriler kendi kişisel gelişimimde kullandığım ve faydasını gördüğüm yöntemleri ihtiva etmektedir, laf olsun diye işkembeden salladığım şeyler değildir. Verimli olduklarını düşünmesem sizinle paylaşmazdım. Umarım faydasını görürsünüz. Kullandığınız kişisel gelişim metotlarını benimle paylaşırsanız sevinirim.

Ultra Lüks

http://www.kurumsaljava.com/2012/06/19/ultra-luks/

Biliyorsunuz yazılim sektörü sadece biz yazılımcılardan oluşmuyor. Bu sektörün proje yöneticisi, satış elemanı, testcisi gibi emekçileri de var. Bu meslek gurupları içinde teknik anlamda yazılımın nasıl yapılması gerektiğini bilen ya da bilmesi gereken biz yazılımcılarız. Bu sektördeki diğer meslek guruplarından bu konuda bir ümit beklemeyin.

Durum böyle olmasına rağmen, bir yazılım projesinin nasıl yürütülmesi gerektiği konusunda en az söz sahibi olan biz programcılarız. Bize gelene kadar birçok insan köklü bilgi sahibi olmadan yazılım projelerine yön veriyor, vermeye çalışıyor. Biz programcılar için yazılım kalitesi ön planda iken, ön planda olmalıyken, bu kişiler çok daha değişik parametrelerin gravitasyonuna kapılıp, açıkçası işi berbat ediyorlar. Bu şahısları motive eden en önemli faktör: "ürünü bir an önce satmak".

Bir ürünü çabucak satmanın önündeki en büyük engel nedir? Cevap: zaman kaybı olarak algılanan aktiviteler. Hangi aktiviteler örneğin? Cevap: birim testleri yazmak, birlikte kod inceleme seansları yapmak (peer code review), sürekli entegrasyon vs. İnanmıyorum dercesine kafanızı sağa, sola salladığınızı görür gibiyim. Benim gibi bunların yazılım kalitesi açısında önemli aktiviteler olduğunu düşünüyorsunuz değil mi? Ama programcılar haricindeki diğer yazılımla ilgili meslek guruplarında böyle bir algılama ne yazık ki mevcut değil. Onlar için bu saydıklarım lüks, biz programcılar ise büyümeyi beceremediş ve istekleri bitmeyen şımarık çocuklarız.

Ellerinden gelse bizi hemen kapıya koyacaklar ama göbekten bize bağlı olduklarını bildiklerinden sesleri çıkmıyor. Bizim de sesimiz çıkmıyor. Arkadaş, yukarda saydığım aktivitelerin gerekliliğine inanmış, ama uygulanmadıkları yerde masaya yumruğunu vurup, "savulun ulan, biz bu aktiviteleri yapacağız, işinize gelirse" diyen cesur bir programcı görmedim. Ağlamayan bebeğe kim emzik verir ki? Bize bu şekilde davranmaları ve anormal şartlarda çalıştırmaları normal!

Yukarda saymış olduğum aktiviteler modern yazılım pratiklerinden bazılarıdır. Nasıl bir duvar ustasının spatula, fırça gibi alet, edevatı olmadan çalışması zorsa, biz programcıların da bu pratikleri uygulamadan anlamlı sonuç üretmesi zordur. Ha, bu söylediklerim zaten günü kurtarmaya çalışanlara değil! Ben yazılım yapmayı bir bilim ve mühendislik dalı olarak görenler adına konuşuyorum.

Saydığım aktivitelerin lüks olarak görülmesini engelleyememişken, test güdümlü yazılım gibi aktiviteler nasıl algılanıyor sizce? Cevap: **ultra lüks!** Adam kafadan "iki programcı neden aynı işi yapsın ki, boşuna kaynak harcamam" diyor. Bak! Bak! Arkadaşım sana iki çift sözüm var: Birincisi biz kaynak değiliz, insan evladıyız, ikincisi şimdi sana test güdümlü yazılımın faydalarını anlatırdım, ama anlama kapasiten var mı bilmiyorum.

Zor! Gerçekten bu şartlarda, bu köhnemiş kafalarla beraber çalışmak çok zor. Bir dünya hayal ediyorum, içinde profesyonel programcıların mutlu, mesut oldukları, test güdümlü yazılım gibi pratiklerin ultra lüks değil de, gerekli oldukları düşünülen, proje gidişatını yetkin yazılımcıların

tayin ettiği bir dünya. Ne yazik ki böyle bir dünya yok. Elimizdeki ile yetinmek zorundayız. Ama bu ara sıra masaya yumruğumuzla vuramayız anlamına gelmiyor.

Benlik Güden Programcılar

http://www.kurumsaljava.com/2012/06/04/benlik-guden-programcilar/

Geçenlerde bir okuyucumdan bir e-posta aldım. İletisinin bir bölümünde şöyle yazmış:

Mağrur programcı ustalar sizin gibi mi? Burnundan kıl aldırmaz. Onlara da şöyle diyorum: Mağrur olma padişahım senden büyük Google var. Bunu okuyunca aklıma bu yazıyı yazmak geldi.

Ninem bize ufakken "benlik gütmeyin yavrularım, güzel bir şey değil" derdi. Yıllar sonra ne demek istediğini anladım. Allah rahmet eğlesin, çok dayağını yedim ufakken :) Bir keresinde... neyse bunları başka bir yazımda anlatırım.

Benlik gütmek demek, her şeyi ben merkezli görmek demektir. Benlik güden bir şahıs yakaladığı her fırsatta "ben yaptım, ben başardım, en iyisini ben bilirim, bu konuda benden daha iyisi olamaz" şeklinde kendini ifade eder ya da düşünür. Kendisini mükemmel ve kusursuz görerek benlik güder ve kibirlenir.

Programcılık mesleği de ne yazık ki benlik gütmek için ideal bir ortam sunar. Programcılığın temelinde bilgi sahibi olmak yatar. Bu bilgiye sahip olan haklı bir gururla işini yapmaya çalışır. Bazı şahıslarda ne yazık ki bu gurur kibire dönüşür. **Bilgi bu tür şahısları ne oldum delisi yapar.** Bilgiye sıkı sıkı sarılırlar. Kimseyle paylaşmazlar. Gizli gizli yeni bilgi edinip, bilgilerine bilgi, kibirlerine kibir katalar.

Bu tip programcılarla mutlaka siz de karşılaşmışsınızdır. Her projede bir örneğini bulmak mümkündür. Ben bu tür programcılara yıldız programcı (star developer) diyorum. Yıldız programcının hemen göze batan birkaç özelliği vardır. Toplantılarda en sesli onlar konuşurlar. Söyleyecek mutlaka bir şeyleri vardır. Genelde ortaya atılan görüşe ters pozisyon alarak kendilerini dolu dolu ifade etme fırsatı yaratırlar. Siz bir şey söylemek istediğinizde "evet, ama..." şeklinde bölüp, kendi düşüncelerinin ne kadar daha doğru olduğunun altını çizerler. Karmaşık çözümleri yeğlerler. Bu onlara iki türlü avantaj sağlar: kimsenin kodu anlamaması yıldız programcının uzun vadede çalışma yerini korumasını sağlar (job protection); kodu anlamayan diğer programcılar devamlı yıldıza danışmak zorunda kalırlar. Bu da ona vazgeçilmez olduğu hissini verir. Yıldızlar blog ya da kitap yazmazlar. Kitap yazdılarsa bile referans olsun, namım yürüsün düşüncesiyle yazmışlardır. Bilgiyi güç olarak görürler ve bilgi paylaşmayı güç paylaşma olarak algılarlar. Böyle bir tip ile çocukluğuna geri dönüp, nelerin yanlış gittiğini gerçekten görmek isterdim. Mutlaka bir yerlerde bir şey olmuş olmalı. Ya annesinden yeterinde ilgi görmedi ya da okulda çok dayak yiyip, ezik büyüdü. Bu çünkü normal bir davranış biçimi değil!

Sektörümüzde benlik güden, kendini beğenmiş, sözde programcılara ihtiyacımız yok. İhtiyacımız olan, bilgisini karşılık beklemeden paylaşan, her konuda mütevazi, gençlerimize örnek olabilen programcılardır. Alın size birkaç örnek: Cem Ikta, Murat Yener, Gökalp Kuşcu, Mehmet Çeliksoy ve daha niceleri. Hepsinin blog sayfası var, devamlı bilgiyi paylaşma derdindeler. Neden? Bilginin paylaşarak çoğaldığının bilincindeler. Yeni yetişen gençlere yardım etmeyi seviyorlar. Kendilerinin de bir zamanlar o konumda olduğunu unutmamışlar. Saygı ve hürmeti hak ediyorlar.

Okuyucumun dediği gibi *Mağrur olma padişahım senden büyük Google var.*

Melek Programcılar ve Şeytan Yatırımcılar

http://www.kurumsaljava.com/2012/05/31/melek-programcilar-ve-seytan-yatirimcilar/

Melek yatırımcıları (business angels) bilirsiniz; hani şu yeni kurulmuş firmalara (startup) yatırım yapan şahıslar. **Melek programcıları bilir misiniz?** Bunlara gelmeden önce şeytan yatırımcıları açıklamam gerekiyor. Melek yatırımcıların yanında bir de şeytan yatırımcılar (business devils) vardır. Bunlar da melek yatırımcılar gibi yatırım yaparlar, ama niyetleri iyi değildir. Maksatları bellidir; kısa zamanda firmayı satıp (exit), para yapmak. Bu uğurda yapmayacakları yoktur. Genelde dünyadan ve ticaretten haberi olmayan, belki üniversiteyi yeni bitirmiş, İnternet üzerinden uygulanabilecek iyi bir ticari fikre sahip bir veya birden fazla programcının kurduğu firmalardır kendilerine seçtikleri kurbanlar. Ben bu programcılara **melek programcılar** diyorum.

Geçenlerde Türkiye'de Formspring.com vari bir İnternet platformu kurmuş ve ufak miktarda yatırım almış iki gencimizin haberini okudum. İlk bakışta güzel bir gelişme olarak görünüyor; sevindirici. Ama yakından incelediğimizde durum gençler açısından içler acısı. Benim dikkatimi çeken iki nokta oldu: gençler sadece programcı; iş modelleri hakkında bir fikre sahip değiller, yani nasıl para kazanacaklarını bilmiyorlar. Şeytan yatırımcılar için bu arkadaşlar çantada keklik.

Bu girişimci arkadaşlara soruyorlar iş modeliniz nedir diye. Arkadaşlar platformu kurmuşlar, yatırımı da almışlar ama verdikleri cevap çok enteresan: „Şimdilik bu konuda bir çalışmamız yok; önce platformu büyütüp, gelişmelere göre çalışmalarımızı yönlendireceğiz". **What?** Bu arkadaşlara bir çift çözüm var. Böyle bir saçmalık nerede görülmüş. Girişimci olmak için kolları sıvıyorsun, ama nasıl para kazanacağın hakkında fikrin yok. Böyle bir şey olabilir mi? İnanılacak gibi değil! Oluyormuş anlaşılan. Türkiye'deki genç girişimcilerin çoğu aynı durumda. Nasıl para kazanacaklarını bilmeden piyasaya atlayıp, şeytan yatırımcıların kurbanı oluyorlar.

Gelelim şeytan yatırımcılara. Böyle bir firmaya yatırım yapsa yapsa bir şeytan yatırımcı yapar. Neden? Öncelikle iki çaylağın ufak miktarda yatırım yaparak gözünü boyamak kolaydır. Mümkünse, yani bizim çaylakların çaylaklık oranı yüksekse yaptıkları yatırım ile firmanın yüzde altmışını ya da daha fazlasını ele geçirirler. Bunun karşılığında da elli, altmış bin Lira yatırım yaparlar. Ne oldu? Öncelikle firma elden gitti. Kurucu programcılara ne oldu? Bu arkadaşlar da kendi firmalarının çalışanı haline geldi. Şeytan yatırımcı bununla yetinir mi zannediyorsunuz. İki genci beleşe çalıştıracağı başka projeleri vardır mutlaka çekmecede. Arkadaşlar gıkını çıkaramadan başlarlar şeytan için çalışmaya. Şeytan için her şey mükemmeldir. Beleşe firma sahibi ve maaş bile vermeden çalıştırdığı programcıları vardır artık.

Bizim çaylakların sonu malum.

Girişimcilik kötü bir şey değil, girişimci olmayın demiyorum. Ama hangi şartlarda ve hangi bedeli ödeyerek girişimci olunmalı, buna dikkat çekmek istiyorum. Programcılar ne yazık ki çok çabuk gaza gelip, şen şakrak bir arkadaş ortamında ortaya atılmış saçma sapan bir fikri İnternette gerçekleştirmek için kolları sıvıyabiliyorlar. Sonuçta bu işin maliyeti ne ki, iki satır kod ve kiralık bir sunucu. Ben bu işi beceririm diyor programcı kendi kendine. Ama çalışır bir platformu meydana getirmek kıranın sadece yarısı, bunun farkında değil. Bunun geliştirilmesi, pazarlanması, aylık sabit giderleri gibi birçok derdi var. Programcının bu sorunların altından tek başına kalkması mümkün değil. Çoğu programcı pazarlama işinden anlamadığı için böyle projelerin çoğu yarı yolda kalıyor. Verilen emeğe yazık! Bu arada birde paçayı bir şeytan yatırımcıya kaptırdıysalar işler da kötü demektir. Hulasai kelam bu işler kolay değildir. İki satır kod yazmayla girişimcilik olmaz.

Bana soracak olursanız içinde sadece programcıların olduğu bir girişim muvaffak olamaz. Mutlaka değişik kulvarlarda koşabilecek şahısların bir araya gelmesi gerekir. Örneğin bir programcı yazılım sistemini oluştururken, işletmeci diğer bir şahsın platformu pazarlama planı yapması lazımdır. Bunun yanında finansman konusunda üçüncü bir şahsın gider ve gelirleri dengede tutup, melek yatırımcılar ile bağlantı kurması ve yatırım oluşturmaya çalışması gerekir. Bir programcı tek başına bu işlerin üstesinden gelemez. İşin kötüsü yazılım sevdasından dolayı düşündüğü en iyi yazılım sistemini oluşturmaya çalışması ve zaman kaybetmesidir. Çoğu zaman teknik detaylarda kaybolup, hitap etmek istediği kitlenin gereksinimlerine cevap verecek bir çözüm oluşturamaz. Bir şeyleri becermek, çalışır hale getirmek ona çok daha fazla haz verir. Müşteri gereksinimleri arka plana düşer. Nasıl para kazanırım sorusuna "en kötü ihtimalle reklamdan para kazanırım" gibi absürt bir cevap verir. Reklam pastasının, pastayı yemeye çalışanlarla doğru orantıda büyümediğini iş işten geçtikten sonra anlar. Günaydın!

Şeytan yatırımcılara dikkat edin diyorum sadece; onlar insanın ruhunu satın almaya çalışırlar!

Mantığın Köleleri

http://www.kurumsaljava.com/2012/05/14/mantigin-koleleri-2/

Programcılık Sanat mı, Zanaat mı? başlıklı yazıma gelen yorumlar, programcılığın sanat olduğu yönünde. Bunun aksini düşünenler de var. Programcılığın bir sanat olarak algılanması subjektif ve bir yanılgıdır. Programcılığın bir sanat olmadığının altını bu yazımla tekrar çizmek istedim.

Sanat ve sanat eserleri görecelidir. İnsanlik var olduğundan beri sanat vardır, ama hala bu konudaki tartışmalar son bulmamıştır. „Renkler ve zenkler tartışılmaz“ sözünü sıkça işitmişizdir. Bu her bireyin sanat algılayış tarzının değişikliğine işaret etmektedir. Benim beğendiğim bir resmi, başka bir insan beğenmeyebilir. Bu yüzden sanatı tanımlamak ve bu işin içinden çıkmak çok karmaşık bir şeydir. Programcılığın da bir sanat olarak algılanıp, savunulması bu açıdan baktığımızda anlaşılabilir bir durumdur.

Bir sanat ya da sanat eseri ruha, göze ve gönüle hitap eder. Sanatın doğasında soyutluk vardır ve insanın kendisine özgün yorumuna ihtiyaç duyar. Bu yüzden herkesin sanatı algılayış tarzı değişiktir. Herkes sanatı kendi tarzında yorumlar. Eğer yazılan bir program parçası bir sanat eseri olmuş olsaydı, o zaman bu programa bakan herkes başka bir şey algılayacaktı ya da ondan kendi yorumladığı bir davranış biçimi bekleyecekti. Böyle bir şeyin yazılımda ne anlama geldiğini düşünebiliyor musunuz? Programcılık deterministik sonuçlar üretmek, mantık çerçevesinde kalıp, çözüm sunmak zorundadır. Ruha, gönüle değil, mantığa hitap eder. Sadece bu özelliğinden dolayı bile bir sanat olma şansı yoktur. Kim bir programa bakarsa baksın, aynı davranış biçimini görür, görmek zorundadır. Bir program her zaman aynı sonucu üretmek zorundadır, kod nasıl yazılmış olursa olsun. İnsanların yorumuna ihtiyacı yoktur, çünkü herkes aynı şeyi görür, daha dogrusu görmek zorundadır.

Programlar ihtiyaçtan doğarlar. Aynı şeyi sanat eserleri için söylemek mümkün değildir. Sanatkar bir şeylerden esinlenerek sanatını icra eder, zorda kalmadan, birilerine sanatını nasıl icra etmesi gerektiğini danışmadan. Programlar müşterileri tarafından kendi gereksinimlerini tatmin etmek için sipariş edilir. Müşterinin gereksinimleri programı yoğururken ön plandadır. Programı oluşturulurken müşteri ile iletişim esastır. Onun istediği olur.

Bunu yanısıra programcının kod yazarken soyutluğu ifade etmek için çok fazla seçenegi yoktur. Java dilinde interface ya da abstract sınıfları kullanarak soyut bir şeyler ifade edebilebilir. Bu ressamın paletinde sadece bir, iki renk olduğu anlamına gelir. Sadece iki rengi kullanarak sanat eseri oluşturmak mümkün müdür? Belki! Bir ressamın paletinde onlarca renk vardır, bunları karıştırarak sonsuz sayıda yeni renk elde edebilir. Bu renklerin hepsini kullanma potansiyeline sahiptir ve bu sanatına yansır. Günümüzün programlama dillerinde böyle bir zenginlik söz konusu değildir. Kaldı ki en tabana indiğimizde bir programcı sıfır ve birlerden oluşan verilerden başka bir şeyle ugraşmaz. Sadece iki renkli olan bir dünyada yaşar ve bir takım verileri A'dan B'ye taşıyacak programlar yazar. İşin hepi, topu budur.

Günümüzde nesneye yönetlik programlama paradigması popülerdir. Ne olduğuna baktığımızda programcı olarak işimizi yapabilmemiz için çok kısıtlı sayıda araç ihtiva ettiğini görmekteyiz. Gerçek dünyayı modellemek için sınıfları, bu modellere hareket kabiliyeti verebilmek icin

metotları kullanırız. Koca dünyayı modellemek için sadece bu iki aracı kullanabiliyoruz. Kod yazarken yapmak istediklerimizi ifade etmek için if ya da while gibi basit komutları kullanıyoruz. Kullanabileceğimiz başka ne var ki? Bu araçlardan faydalanıp nasıl bir sanat eseri ortaya koyabiliriz? Bu bir sanat eseri olsa bile fonksiyonel işlevi haricinde nesi kimi ilgilendirir?

Görüldügü gibi programcı sahip olduğu araçlar itibari ile çok kısıtlı bir dünyada yaşamaktadır. Oysaki bir sanatçının dünyası rengarenk ve pırıl pırıldır. Sayısız derecede kombinasyon imkanları vardır. Bunlardan faydalanarak sanatını icra eder ve sanat eserleri oluşturur. Biz programcılar sıkışıp kaldığımız mantıksal dünyada bu araç gerece ve lükse sahip degiliz.

Kanımca programcılıkta kreatif olma, sanat icrası ile karıştırılmaktadır. Programcılar kreatiftir. Devamlı karşılaştıkları sorunları aşmak için çözümler üretirler. Bunun için bir zanaatkar usta gibi bir takım alet, edevata ihtiyaç duyarlar. Usta programcılar örneğin tüm tasarım prensiplerine hakımdirler. Nerede hangi tasarım şablonunu kullanmaları gerektiğini bilirler. Az ve öz kod yazmayı yeğlerler. Yazdıkları kod roman gibi okunur, kod okundukça bir hikaye gibi kendisini anlatır. Burada bir tutam estetiklik yok değildir. Ama yinede bu aktivitelerin hiçbirisi bir sanat eseri ortaya koymaz.

Programcılık mantık işidir. **Biz programcılar mantığın kölesiyiz.** Programcılıkta bütün gidişati mantık yönetir. Mantığın bittiği yerde programcılık biter. Bir sanatçı hiçbir şeyin kölesi değildir. Aradaki fark budur. Mantıkla sanatı kavramaya çalışmak beyhude bir iştir.

Meğer Neymişiz!

http://www.kurumsaljava.com/2012/05/10/meger-neymisiz/

Bu sabah yanımda kocaman, Coca Cola pet şişeleri taşıyan TIR vari bir araç durup, yol soruyor. Sürücünün neden navigasyon aleti yok diye düşünürken, bu yazıyı yazma fikri oluşuyor kafamda.

Şimdi birkaç saniyeliğine bir dünya hayal edelim; İçinde IPhone ve Android akıllı telefonların olduğu, ama app denen birşeyin bilinmediği, bilgisayarların olduğu, ama işletim sistemi diye birşeyin icat edilmediği, internetin olduğu, ama iletişim kurmak için e-posta ve web sunucularının olmadığı bir dünya... Bu dünyada eksik olan nedir? Evet, yazılım yapılmıyor bu dünyada. Yazılım sistemlerini kim yapıyor? Evet, biz programcılar. Yani? Bu dünyayı daha yaşanır hale getiren, oluşturdukları yazılım sistemleri ile devrim yaratan, insanlığa yön veren, insanlık tarihinde dünyanın ilk kez kocaman bir köy olmasını sağlayan ve insanlığa hizmet eden biz programcılarız. Ameliyat eden robotlardan tutun, uzaya gönderilen uydulardan, havada uçan uçakların kontrol sistemlerine hadar her yerde, içinde program çalışan birşeyler görmek mümkündür. Yazılım sistemleri modern insanın sürdürdüğü yaşam tarzının vazgeçilmez bir parçası haline gelmiştir.

Hacker'ları bunun dışında tutmak şartıyla oluşturduğumuz birçok yazılım sistemi doğrudan insanlığa hizmet etmektedir. Bunun en güzel örneğini açık kaynaklı yazılım sistemleri oluşturmaktadır. Onların başında da Linux işletim sistemi gelmektedir. Toplum yazılım sistemlerinin insanlık için önemini anlamış olmalı ki, yazılımcılara nişanlar verilmektedir. Bunun bir örneğini geçtigimiz ay Linux işletim sisteminin mucidi Linus Torwald'e verilen **Millennium Technology Prize 2012** nişanı teskil etmektedir.

Ödülü veren akademi tarafından yapılan açıklamada Linus'un açık kaynaklı bir yazılım sistemi geliştirmesinin dağıtık yazılım sistemleri oluşturmaya ve internetin özgür olmasına katkıda bulunduğu belirtilmektedir. Linux işletim sistemi milyonlarca, belki de milyarlarca insan tarafından kullanılma potansiyeline sahiptir. Linus yaptığı açıklamada, yazılım sistemlerinin modern dünyada çok önemli olduklarını ve insanlığa hizmet edebillmeleri için mutlaka açık kaynaklı olmaları gerektiğinin altını çizmiştir.

Durum böyle iken biz programcıların üstlendiği toplumsal sorumluluk artmaktadır. Hatasız yazılım sistemleri oluşturmamız gerekmektedir. Birçok insan günlük hayatında kullandıkları yazılım sistemlerinin hatasız çalışmasına güvenmek zorundadır. Yazılım hatalarının meydana gelmesi onların hata yapmasına sebep verebilmektedir. Baştan savma yazılım sistemleri oluşturarak onların güvenini yitirmeyelim.

Bu arada TIR sürücüsüne *navigasyon aygıtınız yok mu* sorusunu sormadan edemedim. *Var, ama çalışmıyor* dedi. Bir yazılım hatasıymış sanırım ;-) Yazılımda daha iyi olmamız lazım. İnsanlar bize güveniyor.

Programcıyım, Yönetici Değil!

http://www.kurumsaljava.com/2012/05/08/programciyim-yonetici-degil/

Türkiye'de genç programcı adaylarının kariyer planlaması şu şekilde:

Üniversite »» Birkaç yıl programcılık »» Proje yöneticiliği »» Bölüm yöneticiliği »» Bölüm müdürlüğü

Benim kariyer planlamam şöyle:

Üniversite »» Ömür boyu programcılık

Ben programcıyım, yönetici değilim! Üniversitede eğitimini almadığım birşey nasıl olabilirim? Neden programcıların çoğu bir yerlere yönetici ya da müdür olmak istiyor? Bunun açıklaması kolay: daha fazla maaş alacaklarını düşündükleri ya da pozisyon itibariyle daha fazla dikkate alınmak istedikleri için.

Herşeyin statü sembolleri ile ölçüldüğü bir toplumda programcıların daha fazla para kazanmak için yönetici olma sevdalarını ayıplamıyorum. Bu sadece programcılar için geçerli bir durum değil. Benim ayıpladığım programcının bu uğurda ruhunu satmasıdır. Eğer bir insan programcı oldu ise, bu işe gönül verdiği için olmuştur. Mecburiyetten olan programcı zaten programcı sayılmaz. Eğer yaptığı işi sevmiyorsa, bırakıp başka bir iş yapması mübahtır. Aldığı para kendisini tatmin etmiyorsa, benim kişisel olarak yapabileceğim birşey yok (ama daha fazla nasıl kazanabileceğine daha sonra değineceğim). Ama bu işin birde manevi tarafı yok mu? Haz alarak yaptığı bir iş insanı mutlu etmez mi? Neden daha fazla maaş, insanı mutlu eden bir işe tercih edilir ki? Bunu anlamam çok zor.

Yöneticilik ya da müdürlük öyle her babayiğidin harcı değildir. Yönetici olmak isteyen programcılar bunu bilmiyor galiba. Bu işin bir defa eğitimini almak gerekir. Yönetim kademesindeki insanların liderlik, çalışanlara örnek olma, onları motive etme, güçlü iletişim gibi vasıflara ihtiyacı vardır. Çok ağır şartlarda çalışıp, neticesi geniş kapsamlı kararlar vermek zorunda kalırlar. Üstlerinde büyük sorumluluk taşırlar. Daha fazla maaşı hak ederler, ama bedelini de öderler.

Saydığım bu sebeplerden dolayı yöneticiliğe talip değilim. Ben bu işten anlamam. Ben bildiğim işi yapmayı yeğlerim. Programcıyken, yöneticiliğe soyunanlara da kazaları mübarek olsun derim.

Programcılar Yönetici Olmaya Zorlanıyor

Ömür boyu programcı olarak çalışmak isteyenler de yok değil. Nedendir bilmiyorum ama Türkiye'de „insan kırk yaşından sonra programcı olarak çalışmaz" kanısı oluşmuş. Yaşlı programcılara kötü gözle bakıldığı bir toplum içinde yaşıyoruz. Hala bir baltaya sap olamamış ve programcılık yapıyor denilebiliyor. Aman allahım, inanılacak gibi değil!

Bu şartlar altında programcıların üzerinde yönetici olma doğrultusunda baskı oluşmuyor değil. Lakin bu baskıya karşı koyup, programcının programcı olarak yoluna devam etmesi şarttır.

Programcının gönlünde yatan aslan bellidir. Programcılar kendi kariyer planlamalarını kendileri yapmalı, başkalarının buna müdahele etmesine karşı koymalıdırlar. Müdahele olduğunda buna baş kaldırıp, dik durmalı ve statükolarını korumalıdırlar. Yoksa sonuç bellidir: yaptığı işten memnun olmayan bir sürü eski programcı yönetici.

Bunun yanısıra birde firmalardaki kariyer yapma imkanlarına göz attığımızda, çoğu programcıya yönetici olmaktan başka seçenek kalmadığını görmekteyiz. Avrupa ve Amerika'da durum çok farklıdır. Büyük yazılım firmalarına baktığımızda, ömür boyu mühendis kalınabildiği, bu konuda uzmanlaşma imkanı veren kariyer patikalarının olduğunu görmekteyiz. Bunun bir örneğini IBM'in çalışanlarına sunduğu Distinguished Engineer olma imkanı teşkil etmektedir.

Programcılar Mutlu Değil

Programcılar ne yazık ki yaptıkları işten memnun ve mutlu değiller. Devamlı günü kurtarmaya çalışmak, verilen emeğe ilgi ve saygının olmaması, fazla mesai yapmaya zorlamalar programcının içindeki şevki kırabilir. Ama bir programcı yöneticiliğe upgrade yaptığında durum farklı mı olacaktır? Pek zannetmiyorum. Öyleyse gönül verdiğimiz işi yapacak şekilde ortam hazırlamaya bakalım.

Programcılar İşsiz Kalmaz

Programcılar şarap gibidir. Kendilerini geliştirmeleri, ustalaşmaları şartıyla yaşlandıkça kıymetlenirler. Yetkin bir programcının piyasada işsiz kalması imkansızdır. Herhangi bir işi yapan bir yöneticinin işsiz kalma ya da işten çıkarılma rizikosu yetkin bir programcıdan çok daha yüksektir. Yetkin bir programcıyı kolay kolay kimse işten çıkartamaz. Geliştirdiği uygulama hakkında o kadar çok bilgiye sahiptir ki yeri kolay kolay doldurulamaz. Yetkin programcılar piyasada freelancer olarak çalışıp, maaşlarının birkaç katı fazlasını kazanabilirler.

Bu kadar uzman progragramcının arandığı bir piyasada kim ne yapsın yöneticiliği. Programcılık güzel ve geleceği olan bir meslektir. Kıymetini bilelim.

Programcılık Sanat mı, Zanaat mı?

http://www.kurumsaljava.com/2012/05/08/programcilik-sanat-mi-zanaat-mi/

Çoğu zaman aklımda olan bir soru var: „Programcılık bir sanat mıdır yoksa bir zanaat mıdır? Biz programcılar sanatkar mıyız yoksa zanaatkar mıyız?

Vikipedi'ye baktığımızda sanat ve zanaat için aşağıdaki tanımlamayı yapmakta:

Sanat en genel anlamıyla, yaratıcılığın ve hayalgücünün ifadesi olarak anlaşılır. Tarih boyunca neyin sanat olarak adlandırılacağına dair fikirler sürekli değişmiş, bu geniş anlama zaman içinde değişik kısıtlamalar getirilip yeni tanımlar yaratılmıştır. Bugün sanat terimi birçok kişi tarafından çok basit ve net gözüken bir kavram gibi kullanılabildiği gibi akademik çevrelerde sanatın ne şekilde tanımlanabileceği, hatta tanımlanabilir olup olmadığı bile hararetli bir tartışma konusudur. Açık olan nokta ise sanatın insanlığın evrensel bir değeri olduğu, kısıtlı veya değişik şekillerde bile olsa her kültürde görüldüğüdür. Sanat sözcüğü genelde görsel sanatlar anlamında kullanılır.

Zanaat , sermayeden çok nitelikli emeğe dayalı; öğrenimin yanısıra el becerisi ve ustalık gerektiren meslek. Zanaatkâr, zanaatle uğraşan kişi anlamına gelir. Marangozluk, ayakkabıcılık, kuyumculuk (takı üreten), kumaş boyama, çömlekçilik, berberlik, bakırcılık gibi mesleklerin hepsi birer zanaattir. Bir kimsenin zanaatkâr olması için el becerisi gerektiren bir malı veya hizmeti sadece satması değil, bilfiil üretmesi gerekir.

Yaptığımız işte he ikisinden de birşeyler var değil mi? Okunması kolay estetik kod yazarak bir sanatkar, müşteri gereksinimlerini tatmin eden uygulamalar geliştirerek bir zanaatkar olabiliyoruz. Ama gerçek hayatta durum ne yazık ki çok farklı. Sanatkar değiliz, çünkü müşterimiz güzel kod yazıyoruz diye bize iş vermiyor. Zanaatkar değiliz, çünkü yazılımcı olarak usta-çırak geleneğinden yetişerek işimizin başına gelmiyoruz. Neden böyle düşündüğümü bu yazımda sizinle paylaşmaya çalışacağım.

Birgün yolunuz Gaziantep'e düşerse Bakırcılar Çarşısı'na mutlaka uğrayın. Orada çalışan bakır ustaları kelimenin tam anlamıyla hem sanatkardırlar hem de zanaatkar. Sanatkardırlar, çünkü estetik değerler oluştururlar. Zanaatkardırlar, çünkü usta-çırak kültürüyle yetişip, el emekleri ile bu değerleri oluştururlar.

Bir bakır ustasının elinde vücut bulan ve yirmi ile yüz Türk Lirası arasında bir fiyata sahip bir bakır kupanın sadece su içmek için edinildiğini görmedim. Ufak bir sanat eseri olan bu bakır kupayı evimizdeki vitrinleri süslemek için ediniriz. Bizim için önemli olan kupanın fonksiyonel işlevi (kupadan su içmek) değildir. Daha ziyade kupanın estetik yönü ilgimizi çeker. Bu yüzden bakır kupa bir sanat eseridir, bakır ustası da bir sanatkar.

Peki camdan üretilen su bardakları sanat eserleri midir? Bu su bardaklarını üretenler sanatkar mıdır? Şüphesiz tanesi yüzlerce Türk Lirası'nı bulan su bardakları bulmak zor değildir. Ama pahalı olmaları sanat eserleri oldukları anlamına gelmez. Camdan yapılmış bir su bardağını su içmek için alırız. Bizim için su bardağının tatmin etmesi gereken fonksiyonel bir işlevi vardır. Su bardağının bu fonksiyonel işlevi yerine getirirken bir sanat eseri olup, olmadığı bizi ilgilendirmez. Herhangi bir marketten tanesi bir Türk Lirası olan onlarca su bardağı alıp, mutfaktaki dolaba koyarız. Su içmek istediğimizde bir bardak alıp, kullanırız. Bizim için camdan üretilmiş bir su bardağının fonksiyonel işlevi ön planda oldugu için, bu su bardağı bir sanat eseri değildir. Aynı şekilde bu bardakları üretenler de bizim gözümüde sanatkar değildirler.

Buradan çıkardığım sonuç şudur: Sanat eserlerinde fonksiyonel işlev önemli değildir. Dikkat bulan, icra edilen sanattır. İcra edilen bu sanat, sanat eserleri meydana getirir. Fonksiyonel işlevin önemli olduğu yerlerde ise sanat eserleri tercih edilmez. Sanat eseri ve sanatkar olmayı belirleyen faktör budur.

Programcılık sanat, Programcı da sanatkar değildir

Bazı programcılar program yazma aktivitesini sanat olarak, bu işi yaptıklarından dolayı kendilerini de sanatkar olarak görmektedirler. Bu ne yazık ki doğru değildir. Program yazmak fonksiyonel bir işlevi meydana getirmek için yapılan bir aktivitedir. Camdan su bardağı örneğinde olduğu gibi, müşterimiz bizim yazdığımız programın estetik kısmıyla ilgilenmez. Onun için oluşturulan uygulamanın tatmin etmesi gereken sadece fonksiyonel bir işlevi vardır. Uygulamanın ne kadar güzel kodlandığı ile ilgilenmez. Hiç, bir müşterinin kodu açıp, okumaya çalışıp, ne kadar güzel yazılmış, ya da yazılmamış diye yorum yaptığını görmedim. Durum böyle iken, müşteriye sattığımız uygulama bir sanat eseri değildir. Bir sanat eseri oluşturup, satmadığımıza göre, programcı olarak sanatkar olmamızda mümkün değildir.

Bir programcı olarak müşterinin gereksinimlerini tatmin eden, sadece fonksiyonel işlevi olan ürünler ortaya koyarız. Müşteri oluşturduğumuz uygulamaya baktığında bir sanat eseri, bize baktığında bir sanatkar görmez. Ama aynı şahıs bir bakır ustasına baktığında bir sanat eseri ve sanatkar görür. Demek oluyor ki sanat eserine ve sanatkara bu sıfatı değer görenler başkalarıdır. Kendi kendimize sanat eseri ortaya koyduğumuzu ve sanatkar olduğumuzu iddia etmemiz birşeyi değiştirmez. Bu yüzden programcının kendisini sanatkar olarak görmesi doğru değildir, çünkü müşterisinin gözünde sanatkar değildir.

Programcılar zanaatkar değildir

Zanaat demek ustalık demektir; gelenek demektir; ustadan öğrenmek demektir. Siz hiç bir berber çırağının ilk işe başladığı gün saç kestiğini gördünüz mü? Çırak önce ustasına hizmet edip, ondan birşeyler öğrenmeye ve yavaş yavaş olgunlaşmaya başlayacaktır. Usta ise çırağını kendi tecrübeleri doğrultusunda ve kendi ustasından öğrendigi şekilde yönlendirecektir. Belli bir zaman diliminden sonra çırak yavaş yavaş bir zanaatkara dönüşecektir. Bunun gerçekleşmesi için ustasının kendisine gösterdigi yolu takip etmesi gerekmektedir. Bu yolu gitmek yıllar sürebilir. Ama bu yolun sonunda çırak için kendi ustası gibi bir usta, bir zanaatkar olma ödülü vardır. Ustalaşmak ve bir zanaatkar olmak hiç te kolay olmayan bir süreçtir. Ustası olmayan bir çırağın, bu yolu tek başına giderek ustalaşması çok zor olacaktır ya da mümkün olmayacaktır.

Yukarda yer alan tanımlamaya gore programcılar zanaatkar değildir, çünkü programcılar usta-çırak geleneginden gelerek yetişmezler. Programcılar için işe başlama bariyeri yok denecek kadar azdır. Bir bilgisayar ya da başka bir mühendislik ögrencisi diplomasını eline aldığı andan itibaren programcı olarak çalışmaya başlayabilir. Hiç kimse senin ustan kim ya da kaç sene çırak programcı olarak çalıştın diye soru sormaz. Programcının daha önce hangi projelerde çalıştığı sorulur. Buradaki amaç daha çok programcının ne kadar tecrübeli olduğunu anlamaya çalışmaktır. Ama yıl bazında edinilen tecrübe, şahsın ustalık oranını göstermez. On sene tecrübesi olduğunu söyleyen bir programcı, son dokuz senesini, birinci sene öğrendiği şeyleri tekrarlamakla geçirdiyse, son dokuz sene tecrübe topladığı söylenemez.

Tecrübesiz ve bir usta tarafından yetiştirilmemiş bir programacının hemen program yazmaya başlamasını, diplomasını almış bir doktorun hemen kalp ameliyatı yapmaya başlamasıyla kıyasladığımızda, durumun programcılar açısından aslında ne kadar absürd olduğunu görebiliriz. Kalp ameliyatı yapmaya aday bir doktor yıllarını usta doktorlara asistanlık yapmakla geçirir. Bu ona kalp ameliyatlarının nasıl yapıldığını ustalarından öğrenme fırsatı verir. Asistanlık dönemi bittikten sonra, asistan doktor seçtiği dalda uzmanlaşma dönemine girer. Hem asistanlık hem de uzmanlaşma döneminde doktor adayı ustalarından gerekli herşeyi öğrenerek, onlarca seneyi bulacak bir zaman diliminden sonra kalp ameliyetlarını yapabilecek kıvama gelir. Yukarda da belirttiğim gibi programcılar için bunların hiçbirisi geçerli değildir. Programcı diplomasını alır, masasının başına oturur ve hemen müşteri için program yazmaya başlar. Neden birçok projenin yapılan hatalı kalp ameliyatlarından dolayı öldüğü ortadadır, çünkü ameliyatı yapan programcılar yetkin değildir.

Usta-çırak geleneğinin olmadığı yerde bir zanaatin icrası ya da yapılan işin zanaat olarak tanımlanması çok zordur. Programcılar ustasız yetiştikleri için, zamanlarının büyük bir kısmını deneme yanılmayla boşuna sarf ederler. Yıllarca aynı şeyleri yapıp, yerlerinde saysalar bile, bunu tecrübe yapmış olarak algılarlar. Kendilerine yön gösteren bir ustaları olmadığı için kendilerini geliştirmekte ve ustalaşmakta çok zorlanırlar. Bu sebepten dolayı birçok programcı ustalık mertebesine erişemez. Oysaki bir ustanın denetiminde olsalar ustalaşmaları çok daha kolay olacaktır.

Programcılık zanaattir

Bir önceki bölümün başlığına baktığınızda bu bölümün başlığı tezat gibi algılanabilir. Bir önceki bölümde neden zanaatkar olamadığımızı anlatmaya çalıştım. Bu bölümde ise aslında neden

zanaatkar olduğumuzu anlatmak istiyorum.

Zanaati, sermayeden çok nitelikli emeğe dayalı; öğrenimin yanısıra el becerisi ve ustalık gerektiren meslek şeklinde tanımlamıştık. Bunun yanısıra bir zanaat usta-çırak geleneğini kapsamaktadır. Bu tanımlama bir programcıya tamamen uygun yapıdadır. Eksik olan programcıların ustasız yetişmeleridir. Ama bu onların zanaatkar olmadıkları anlamına gelmez. Bu eksiği kapatmak için ustalığa giden yolun tarifini yapmamız gerekmektedir. Bilge Programcı programı (http://www.bilgeprogramci.com – şu an üzerinde çalıştığım, yapım aşamasında olan projemdir) böyle bir yolun tarifesini ihtiva eder. Daha once bu yoldan geçmiş, şimdilerde usta programcılardan kendimize örnek alarak, bizde yaptığımız işte ustalaşabiliriz. Ustalaştıkca bir zanaatkar olmaya başlarız ve yaptığımız iş bir zanaat haline dönüşür. Usta programcılar zanaatkardırlar. Genç programcılara da onlara çırak olmak düşer. Eti senin, kemiği benim misali :)

Not: Bu yazının devamı olarak Mantığın Köleleri başlıklı biz yazı daha oluşturdum. Bilginize sunarım.

Ekibin Dikkati ve Saygısı Nasıl Kazanılır?

http://www.kurumsaljava.com/2012/05/05/ekibin-dikkati-ve-saygisi-nasil-kazanilir/

Her programcı yeni bir ekibe dahil olduğunda kendisini ispatlamak zorundadır. Bu gerçekleşene kadar ekip içindeki diğer programcılar yeni programcıyı dikkate almazlar. Birilerinin ekibe dahil olduğunu beyinlerinin en arkasında çalışan bir thread ile belki farkederler, ama bunu dikkate almadan günlük işlerine devam ederler. Yeni programcıya kendisini ispatlayana kadar pek ekibin bir parçası olarak bakmazlar. O yokmuş gibi davranabilirler, çünkü kendi aralarında sosyal bir ağ kurmuşlardır ve tüm iletişim trafiği mevcut bu ağlar üzerinden gerçekleşir. Programcı bu sosyal ağa henüz dahil olmadığı için oyun dışı kalabilir. Ekibe ve ekibin sosyal ağına dahil olmak ve ciddiye alınmak için programcının derhal kendisini ispatlaması gerekmektedir.

Yeni bir ekibe katılan bir progrmacının kendini ispatlama stratejisi nasıl olmalıdır? Eğer programcının ünü yeni ekibe kendisinden önce ulaşmadı ise, o zaman programcının işi hiç te kolay değildir. Eğer ünü yoksa, durum daha da vahimdir. Kendisini o projede ispatlayana kadar canı çıkabilir, belki de bunu hiç başaramaz ve bu onu mutsuz olmaya itebilir.

Bu stratejinin nasıl olmaması gerektiği ile başlayalım isterseniz. Eğer programcı bir kenara çekilip, "bana verilen işleri yapmaya çalışayım" derse, o zaman baştan oyunu kaybetti demektir. Farklılık yaratmak ve profil oluşturmak istiyorsa, bir görevin (task) kendisine verilmesini beklemeyip, kendisi görev seçmelidir. Genelde kimsenin cesaret edip, üstlenemediği görevler ortada kalır. Bu tip görevler, diğer ekip arkadaşları arasından sıyrılıp, kendisini kısa zamanda ispatlamak için biçilmiş kaftan gibidir. Doğal olarak zor görevler yetkinlik gerektirir. Ama programcıyı pişiren bu tür görevlerdir. Örneğin her projede mutlaka ve mutlaka performans problemleri yaşanır. Uygulama sunucusu göçer, JVM takılır kalır, threadler bloke olur vs. Yaşanan bu tür problemler her programcının ilgi alanına girmediği için bu tür işlere bulaşmadan kendi işlerine devam ederler. Yeni programcı bunu fırsat bilip balıklama bu tür görevlerin üstüne atlamalıdır. Herkesin başını ağrıtan bu tür problemlerden birisini çözdüğü andan itibaren kendisini ispatlamış sayılır ve yeni ekibi kendisini sevgi, hürmet ve selamla kuçaklar. **Buradan da anlaşıldığı gibi en büyük erdemlerden birisi cesarettir; cesaret edip görev üstlenmektir.**

Hadi diyelim yeni programcı üstlenebileceği enteresan bir görev bulamadı ya da kendisine bir görev tayin edildi. Bu durumda işini en iyi şekilde tamamlayıp, elde ettiği neticeleri tüm ekibe duyurmanın yollarını bulmalıdır. Her ekibin kullandığı e-posta listeleri vardır. Bir e-posta iletisi ile tüm ekibe, hatta proje ve takım yöneticilerine erişmek mümkündür. Yeni programcı kod yazarken nasıl farklılık yaratarak, takım arkadaşlarının dikkatini çekebileceği hakkında düşünmelidir. Örneğin kendisine verilen görevi test güdümlü kodlayıp, elde ettiği yüksek seviyedeki testlerin kod kapsama alanı metriğini e-posta listesi üzerinden duyurabilir. Otomatik çalışan unit testleri, %90 üzerindeki code coverage (kod kapsama alanı), SOLID uyumlu ve kolay okunan sınıflar ekip arkadaşlarını etkileyecektir. Birçok projede programcılar unit testleri yazmadığı için, yeni programcının oluşturduğu ve otomatik çalışan testler dikkat çekmeyi kolaylaştıracaktır. Yerlerde sürünen kod kapsama alanı, yeni programcının oluşturduğu unit testleri ile birden bire tavana fırladığında, bakın bakalım kim anında yıldız programcı oluveriyor. **Yeni programcının yazılımın nasıl olması gerektiği yönündeki dolaylı boy**

göstermeleri, onun kısa sürede ekibin sosyal ağına dahil olmasını kolaylaştıracaktır.

Eger proje bünyesinde Jenkins, Sonar gibi araçlar kullanılmıyorsa, yeni programcının söz alıp, bu araçların kullanımının proje için ne kadar faydalı olduğunu dile getirmesinde fayda vardır. Örneğin yeni programcı Sonar kurulduktan sonra tüm kodun statik analizini yapıp, sonuçları daha önce bahsettiğim e-posta listesi üzerinden tüm ekip çalışanlarına duyurabilir. **Kendisi bu gibi araçları kurmaya ve işletmeye talip olmalıdır**. Bu onu bir vuruşta belli bir alanda söz sahibi yapar. Yeni araçları kullanmak isteyecek olan ekip çalışanları yeni programcıya danışmak isteyecekler ve bu şekilde yeni programcının ekibe entegre edilme sürecini kolaylaştıracaklardır. **Yeni programcının, yazılım sürecini olumlu etkileyecek yeni fikirlerle gelmesi, kendisini ispatlama sürecini kısaltacaktır.**

Yeni programcı ekip toplantılarında mutlaka kendi fikrini beyan etmelidir. Yanlışta olsa bir fikir beyan etmek, hiç fikir beyan etmemekten daha iyidir (tamamen saçmalamamak şartıyla). Eğer susup, konuşulanları dinlerse, ekip arkadaşları onu dikkate almamayı sürdüreceklerdir. Bir fikir sahibi olmayan ya da bunu beyan etmeyi bilmeyen bir insanın toplum içinde yer bulması zordur. Ekip içindeki programcılar karşılaştıkları problemleri e-posta aracılığı ile çalışma arkadaşlarına duyururlar. Bu gibi e-postalara herkesin okuyacağı şekilde cevap vermek, çözüm önerilerinde bulunmak, zaman içinde yeni programcının profilini biler.

Kendini ispatlamanın bir diğer yolu takım arkadaşları ile kod inceleme (code review) seansları düzenlemektir. Çoğu ekip bu tür bir alışkanlığa sahip değildir. Beraber kod inceleme seansları karşı tarafı eleştirme olarak algılanır ve bu yüzden bu aktiviteden kaçınılır. Yeni programcı bunu fırsat bilip, kendi kodunun ekip içinden başka bir programcı tarafından incelenmesini sağlamalıdır. Daha önce de bahsettiğim gibi kodun özenli yazılmış, her sınıfın sadece bir sorumluluğa sahip (SRP prensibi), diğer SOLID prensipleri ile uyumlu, otomatik çalışan unit testlerine sahip, değişken ve metot isimlerinin kodun ifade gücünü artıracak şekilde seçilmiş olması, kodu inceleyen programcıda hayranlık uyandıracaktır. Bunu kendisi için tutmaz ve mutlaka diğer ekip arkadaşları ile paylaşır. Bundan emin olabilirsiniz. Yeni programcı bu şekilde kendisi için bir taban oluşturduktan sonra, diğer programcılar tarafından kod inceleme seanslarına davet edilip, fikri alınan bir programcı olacaktır.

Her ekip içinde mutlaka en az bir tane, rahatsızlık veren, sivri, kendisinin yıldız programcı (star developer) olduğunu düşünen bir programcı vardır. Bu tip programcılarla çalışmak çok zordur. Yeni programcı da mutlaka bu yıldız programcının hışmına uğrayacaktır. Bu gibi durumlarda yeni programcı alttan almaya çalışmalı, ekip içinde soğuk bir atmosferin oluşmasını sağlayacak karşı karşıya gelmenin önüne geçmelidir. Eğer yeni programcı yıldız programcı ile karşı karşıya gelirse, diğer ekip çalışanları yıldızın yanında olmayı tercih edeceklerdir. Yeni programcı ne kadar haklı olursa olsun, ekip içinde bir nevi sürü içgüdüsü olduğundan, yıldız programcının tarafında olacaklardir. Bu gibi durumlarda yıldız programcı ile aynı görev üzerinde beraber çalışma fırsatı kollanmalıdır. Yıldız ile iletişim içinde olmak onu yumuşatacaktır. Yeni programcı aynı zamanda yetkinliğini yıldıza gösterme fırsatı bulursa, aradaki buzlar eriyip, diyalog normalleşecektir. Yeni programcı aynı zamanda fırsatını yakaladığında yıldıza haddini bildirme cesareti göstermelidir. Örneğin sorduğu bir soruya alaylı bir cevap alıyorsa, takım liderinin de bulunduğu bir ortamda yıldıza neden böyle bir cevap verdiği sorulabilir. Yıldız kem, küm edip, afallayacaktır. Bu şekilde yeni programcı yıldızı hafiften ısırabilir ve yıldıza fazla

üzerime gelme sinyali verebilir. **Yeni programcı başını dik tutma cesareti gösterdiği andan itibaren, yıldızın davranış biçimi değişecektir.** Kimse karşısında dik duramayan insana saygı göstermez. Yıldıza karşı dik durma yeni programcıya diger ekip arkadaşlarının saygısını getirir.

Ekibin dikkatini ve saygısını kazanmanın diğer bir yolu ekibe bilgi transferi (know-how tranfer) yapmaktır. Örneğin yeni programcı ekipden diğer bir programcıya "istersen üstlendiğin yeni görevi test güdümlü (Test Driven) ve eşli (pair programming) kodlayalım, bunun nasıl yapıldığını sana gösterebilirim" teklifini götürebilir. Böyle bir teklife, konuyla ilgili bilgisi olmayan programcı hayır diyemez. Teklifi alan programcı yeni birşeyler öğrenme hevesi ile yeni programcıya kendini ispatlama firsatını tanıyacaktır. Yeni edindiği bilgiler programcıda hayranlık uyandırır ve bunu mutlaka diğer ekip arkadaşları ile paylaşır.

Her programcının Xing ya da LinkedIn platformlarında mutlaka bir profili olmalıdır. Programcı bu profillerinde hakim olduğu konuları, daha önce çalıştığı projeleri, sahip olduğu ünvan ve sertifikaları ve kendi projelerini (örneğin açık kaynaklı bir yazılım) sergilemelidir. Bu tür profiller artık programcılar için kartvizit ya da vitrin görevini üstlenmektedir. Mülakata giden her programcı hakkında mutlaka internette araştırma yapılır. Bu tür profillerin keşfedilmesi, mülakatın gidişatını olumlu etkileyebilir. Bu tür profillerin önemli başka bir fonksiyonu daha mevcuttur. Yeni programcı ekip içinde yazılmamış, boş bir defter sayfasıdır. Ekip üyelerinin yeni programcı hakkında geniş çaplı fikirleri yoktur. **Bunu değiştirmenin en kolay yolu, ekip üyelerinin internet profillerini araştırmak ve onlara bahsettiğim platformlar üzerinden arkadaşlık teklifi göndermektir.** Arkadaşlık teklifi alan diğer programcılar, yeni programcının profilini inceleme ve onun hakkında fikir edinme fırsatı bulacaklardır. Hele hele yeni programcı ender sertifikalardan birine sahipse, bu yemek aralarındaki sohbetlere malzeme olacak ve bu şekilde yeni programcının ekibe entegrasyonunu kolaylaşacaktır.

Ekip elemanları ile smalltalk, onlara çalışma esnasında tatlı birşeyler ikram etmek ve ekip liderine düzenli olarak rapor vermek sayabileceğim diğer yöntemler arasındadır. Olumlu dikkat çekmek her zaman önemlidir. Yetkin ve tecrübeli bir programcı farkında olmadan yukarda saydığım konularda çalışma yapacaktır. Verimli olmanın en önemli şartı, en kısa sürede ekibe entegre olmaktır. Bunu başarmak programcının elindedir.

Eşli Programlama, Code Review, Code Retreat ve Adada Yaşayan Programcılar

http://www.kurumsaljava.com/2012/04/15/esli-programlama-code-review-code-retreat-ve-adada-yasayan-programcilar/

Bir programcıyı alıp bir adaya koysanız, diğer insan ve progcılarla bağlantı kurmasını engelleseniz, kendi başına çalışmasını ve kendisini geliştirmesini isteseniz, bu programcı programcılık konusunda nasıl bir gelişme sağlardı?

Eğer programcıya o imkanı sağladıysanız ve kitap okuma alışkanlığı varsa bol bol kitap okuyacaktır. Kitap okumaktan canı sıkıldığında kod yazacaktır. Kod yazmaktan canı sıkıldığında tekrar kitap okuyacaktır. Belki de hiçbir şey yapmayacaktır. Adada olduğunu hatırlayıp yüzmeye gidecektir. Gününü gün etmeye çalışacaktır. Bu durum programcıdan programcıya değişecektir. Bu şartlar altında programcının kendisini geliştirebileceğini pek düşünmüyorum. Gelişim için insanlar arası interaksiyon ve iletişim gereklidir. Çoğu zaman başka insanların davranış biçimlerini kopyalayarak gelişim sağlamak bile mümkündür. Sıfır-altı yaş gurubu çocuklarda bunu görmek mümkün; zamanlarının büyük bir kısmını ebeveynlerinin davranışlarını incelemek ve kopyalamakla geçer. Kendi kızımdan bir örnek vereyim. Şimdi iki yaşında. Evin içinde saklanbaç oynuyoruz. O saklandıktan sonra "baba ben oturma odasında, perdenin arasındayım" gibi anonslar yapıyor. Gidip, elimle koymuş gibi buluyorum. Kısa bir zaman önce oyunu tersine çevirip, ben saklanmaya başladım ve kızımın beni bulmasını istedim. Ben doğal olarak saklandığım yeri anons etmiyorum. Bu şartlarda kızımın evin içinde beni bulması zorlaşıyor. Ufaklık olayı çakmış, şimdilerde saklandığında "kızım nerdesin" diye sormama rağmen, gıkı çıkmıyor. Benden öğrendi ve olayı çaktı. Şimdi bu yazıyı yazarken mouse-pad'imi kaçırdı. Bir saniye gidip, alıp, geleyim…. {5 dakika sonra} -Mouse-pad kayıp, bulamıyorum. Nerede kızım diye soruyorum; oturma odasındaki masada, git ara diyor…

Gerçek hayatta da bazı programcılar bir adada yaşıyor gibi kod yazarlar. Diğer programcılarla bilgi alışverişinde bulunmadan ve kodu paylaşmak zorunda kalmadan işlerini görürler. Projeyi zamanında yetiştirme telaşı bu tip programcıların işine gelir. Hatta kodu sahiplenirler. Kimse ne yazdığımı anlamasın diye komplike kod yazan programcılar gördüm. Bunun maksadı Job Protection yapmaktır, yani çalıştığı iş yerini korumak ve başkalarına kaptırmamaktır. Ama bu yazımın konusu bu değil. Bu yazımda programcının kendisini geliştirmesi için neden diğer programcılarla interaksiyona girmesi gerekği konusunu incelemek istiyorum. Adada yaşayan programcılar için söylüyorum: kendi kendilerine kendilerini geliştirmeleri biraz zor gibi görünüyor.

Gelişim için insanlar arası interaksiyonun gerekliliğinden bahsettim. Bu özellikle programcılar için geçerli olan bir durumdur. İnsanlar adeta bir araya gelip beraberce çözüm üretmek için yaratılmıştır. Programcılar da kendi aralarındaki iletişimi ve beraber çalışmayı güçlendirerek, birbirlerinden çok şeyler ögrenebilirler. Eğer konumuz programcıların gelişimi ise o zaman bunun sağlandığı en önemli yerlerden birisi programcılar arası interaksiyondur. Bunun örneklerini eşli programlama (pair programming), beraber kod inceleme (code review) ve beraber pratik yapma (code retreat) aktiviteleri teşkil eder. Bu aktiviteleri ve programcıya kattığı

artı değeri yakından inceleyelim.

Eşli Programlama (Pair Programming)

Eşli programlama Extreme Programming (XP) çevik süreci ile popüer olmuş bir programlama aktivitesidir. İki programcı bir araya gelerek, bir müşteri gereksinimini test güdümlü kodlarlar. Eşli programlama oturumunda programcılar sıkça klayveyi kendi aralarında değişirler. Bu hem fikir alışverişini, hem programcıların birbirlerinden öğrenmelerini, hem de ekipte her programcının proje hakkında aynı bilgiye sahip olmasını teşvik eder. Eşli programlama oturumlarda benim en büyük kazancım, partnerimden onun kullandığı yazılım tekniklerini öğrenerek kendi günlük işlerimde adapte etmem olmuştur. Bu şekilde çok şey öğrendigimi itiraf etmem lazım. Bu yüzden eşli programlama çok severek dahil olduğum bir aktivitedir. Eşli programlama yapan programcılar birbirlerinden farkında olmadan çok şey öğrenmektedirler. Bu programcının şimdiye kadar tanımadığı bir kısa yol tuşundan, değişik tasarım prenbiplerinin nasıl uygulandığına kadar geniş bir yelpazeyi kapsamaktadır.

Eşli programlama pratiği ekibe yeni katılmış ya da tecrübesiz programcıların hızlıca bilgilendirilip, takıma entegre edilme süreçlerini kısaltacaktır. Bu amaca doğru yola çıkılmak istendiğinde tecrübeli bir programcı ile tecrübesiz bir programcı bir araya getirilip, eşli programlama oturumunda bilginin serbestçe akışı sağlanabilir. Eşli programlama sadece bu durumda kullanılır diye bir kural yoktur. Mümkün mertebe her fırsatta uygulanması gereken bir aktivitedir. Eşli programlama metodu, iki programcının aynı işi yaptığı düşünüldüğü için zaman kaybı olarak algılanabilir. İlk etapta bu böyle olsa bile, bu yönde yapılmış olan yatırım projenin başarısına katkıda bulunacaktır. Eşli programlama oturumlarında iki çift göz kodu geliştirdiği için hata oranı daha düşük olacaktır. Bilgi ekip içinde eşit olarak dağıldığından, bir programcının projeden ayrılması proje gidişatını olumsuz yönde etkilemeyecektir.

Kod İnceleme (Code Review)

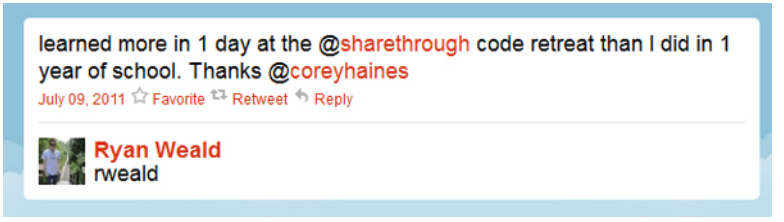
Eşli programlama oturumunda olduğu gibi birden fazla programcı bir araya gelerek bir kod parçasını beraberce incelerler (code review). Genelde bir müşteri gereksinimini kodlamış olan bir programcı çalışmasını tamamladığında, diğer bir çalışma arkadaşını beraberce kod incelemek üzere kendi çalışma masasına davet eder. Dört göz iki gözden daha fazla görür prensibinden yola çıkarak iki programcı kod parçasını beraberce incelerler.

Geçenlerde yaptığımız bir toplantıda beraber kod inceleme konusunda konuşurken, bir çalışma arkadaşım bu tür aktivitelerden hoşlanmadığını, çünkü parmağıyla bir kod satırını gösterip, burası olmamış demenin, kodu yazan tarafından yanlış algılanabileceğini söyledi. Doğru, bu çogu zaman olan birşey, ama birbirimizden birşeyler ögrenmek istiyorsak, o zaman egomuzu artık bir kenara koymamız gerekiyor. Herşeyi bilmemiz, doğru çalışan kod yazmamız her zaman mümkün değil. Bunu böyle kabul etmemiz gerekiyor. Bir çalışma arkadaşımızın kod inceleme oturumunda yanlışlarımızı yüzümüze vurmasını hakaret olarak değil, bir şans olarak algılamalıyız. Bu gibi durumlarda kendisine hakaret edildiğini düşünen programcı adada yaşayan bir programcıdır. Kendimizi geliştirmek istiyorsak kritiğe açık olmamız gerekiyor.

Beraber Pratik Yapma (Code Retreat)

Code retreat Corey Haines tarafından başlatılmış, bir tam gün boyunca 15-25 arası

programcının bir araya gelerek, beraber kod yazdıkları bir aktivitedir. Böyle bir çalışmaya katılmış olan Ryan Weald bir günde ögrendiklerinin okulda bir sene öğrendiklerinden daha fazla olduğunu söylemekte. Bu tür aktivitelerin programcılar için ne kadar önemli olduğu ortadadır.



Diğer programcılarla interaksiyona girmeyen bir programcı, kendisini daha hızlı geliştirmek için sağlanan avantajlardan mahrum kalacaktır. Buradan çıkardığımız sonuç şudur: Job protection yapanlar aslında bindikleri dalı keseler. Oysaki bilgi paylaşımına açık olan programcılar diğer programcılarla olan ilişkilerinde devamlı yeni birşeyler öğrenerek, kendi piyasa değerlerine değer katarlar.

Adada yaşayan programcılara allah kolaylik versin. İşleri kolay değil.

İnşaat ve Yazılım Mühendisleri Arasındaki Fark

http://www.kurumsaljava.com/2012/04/15/insaat-ve-yazilim-muhendisleri-arasindaki-fark/

İnşaat mühendisleri:

- Sıkıysa çok dikkatli çalışmasınlar. En ufak bir yanlış statik hesap yorumlaması bile binanın yıkılmasına sebep olabilir.
- Binayı sadece bir kez inşa edebilirler. Netice hoşlarına gitmeyince, boz yeniden yap yapamazlar, yani tek bir şansları var.
- İnşaatın maliyeti çok yüksektir. Herşey buna odaklı önceden planlanır ve uygulanır. İnşaat mühendisi ne yapacağını önceden en ince detayına kadar bilmek zorundadır. Bunu yazılımda şelale (waterfall) yöntemi ile kıyaslamak mümkündür. Aslına bakacak olursak binalar sadece şelale yönetime göre inşa edilebilir.
- İnşaat sona erdikten sonra müşteri beğenmedi diye örneğin temel sökülüp, yeniden yapılamaz. İnşaatı bitmiş bir bina artık hemen hemen statik ve değiştirilmesi mümkün olmayan ya da çok zor olan bir cisimdir.
- Mimar tarafından bina mimarisi inşaat başlamadan önce yapılır. Mimari yapı belli olmadan inşaat mühendisi binayı inşa etmeye başlayamaz, çünkü ne yapacagını bilemez. Mimar yoksa inşaat mühendisi de yoktur. Gecekondu yapıyorsanız durum farklı tabi.
- İnşaat mühendisi kendisine verilen statik hesaplamaları ve mimariyi harfiyen uygular. İnisiyatif kullanıp, mimariyi ve yapıyı yeniden şekillendiremez.
- Bir binanın yeniden yapılandırılması çok maliyetli bir iştir ve çogu zaman imkansızdır.

Yazılım mühendisleri:

- Yazılım sisteminden bir yapı (build) oluşturmak masrafsız bir iştir. Yazılım mühendisi hergün istediği kadar yapı oluşturabilir.
- Yazılım mühendisi, yazılım sistemini oluştururken inisiyatif gösterip doğru bulduğu tasarım şablonlarını ve tasarım prensiplerini uygulayabilir.
- Yazılım mühendisi, yazılım sisteminin mimarisi belli olmadan kod yazmaya başlayabilir. Nitekim çevik süreçlerde basit ve pragmatik bir mimari ile başlanması önerilmektedir. Zaman içinde müşteri istekleri doğrultusunda mimari oluşur. Değişikliği mümkün kılan da zaten budur.
- Yazılım mühendisi yeni müşteri gereksinimleri ile yazılım sisteminin mimarisini revide edip, değistirebilir. Yazılım mühendisleri ile inşaat mühendisleri arasındaki en bilirgin farklılık budur. Müşteri gelip yazılım mühendisine her zaman yeni isteklerini ve değişiklik taleplerini iletebilir. Yazılım mühendisi bu isteklere her zaman cevap verebilir. Çevik süreç kullanmıyorsa, biraz zor tabi. İnşaat mühendisinin yeni müşteri isteklerine cevap vermesi hemen hemen imkansızdır, çünkü bu tür değişiklikler çok masraflıdır.
- Yazılım mühendisi oluşturduğu testleri kullanarak yazılım sistemini her zaman yeniden yapılandırabilir (refactoring). Yeniden yapılandırma, müşteri gereksinimlerine cevap verebilmek için kullandığı en güçlü silahıdır.
- Yazılım mühendisi Extreme Programming gibi bir çevik süreç kullanarak önünü her zaman açık tutabilir.

Yazılım mühendisleri her zaman yoğrulabilir bir yazılım sistemi ortaya koyma fırsatına sahipken, inşaat mühendislerinin çalışmaları katılaşmış ve değiştirilmesi imkansız neticeler ortaya koyar. Aslında sonuncusu yazılım sektörü için de geçerli bir durumdur. Birçok yazılım mühendisi inşaat mühendisi gibi çalışıyor olmasa da, onlar gibi katılaşmış ve yoğrulması imkansız neticeler (yazılım sistemleri) ortaya koymaktadırlar. Mental çalışma modelimiz inşaat mühendislerinin mental çalışma modellerinden çok farklı olmasına rağmen, neden onlardan farksız sonuçlara imza atamıyoruz? Yoksa bizde mi inşaat mühendisiyiz? **Anneeee....**

Bu arada tüm inşaat mühendisi arkadaşları buradan saygıyla selamlıyorum. Bu yazımda amacım kesinlikle onların çalışma tarzlarını tenkit etmek değildi. Onların çalışma metotları böyle. Başka türlü çalışmaları zaten bizim açımızdan sağlıklı olmazdı :)

Bir yazılım mühendisi olduğum için çok mutluyum. **Güç bende**. İnşaat mühendisleri de mutlu mu acaba? Bir inşaat mühendisi arkadaş bu yazıyı okur ve yorum yaparsa sevinirim.

Kim Daha İyi Programcı?

http://www.kurumsaljava.com/2010/10/23/kim-daha-iyi-programci/

Çoğu zaman programcı adaylarının piyasada en çok talep gören programramlama dilini seçip, bu dili öğrendikleri malum. Bu doğal bir seçim; talep olan yerde arzın bedeli olur. Bu bedel programcının iyi bir maaş ile hayatını sürdürmesi anlamına gelir.

Bilindiği üzere son zamanların en popüler ve talep edilen dili Java. İnternetteki birçok istatiktik Java'nın bir numara olduğunu tastikliyor. Programcı adaylarının da Java'yı seçmeleri doğal.

Java'yi bilenler ve kullananlar iyi ve akıllı programcılardır degil mi? Peki Python ya da piyasası pek fazla olmayan bir başka dili bilen bir programcının sadece Java'ya hakim bir programcıdan daha iyi ve akıllı bir programcı olduğunu söylesem nasıl tepki verirdiniz?

İyi programcı, program yazmayı seven programcıdır. İyi programcı yenilikleri deneyip, ufkunu genişleten programcıdır. İyi programcı programcılık dünyasının sadece Java'dan oluşmadığını bilendir. Peki iyi programcıyı nasıl anlarsınız? Bir sonraki iş görüşmesinde işe alınacak programcı adayına Java haricinde hangi dil ya da dilleri bildiğini sorun. Alacağınız cevap programcının ne kadar iyi olduğunu gösteren indikatördür. Eğer aday sadece Java'yı bildiğini söylerse iyi bir programcı olma ihtimali doğru olabilir. Bunun ispatı edindiği tecrübeler ve bilgi birikimidir. Eğer aday Java'nın yanında Python ya da Groovy dillerine de hakim olduğunu söylerse, bilin ki karşınızda akıllı (smart) bir aday duruyor. Neden? Bunun nedeni çok basit. Piyasası olmayan bir programlama dilini ögrenmiş bir programcı, program yazmayı gerçekten seviyor olmalı ki talebin dışında kalan bir programlama dilini zaman ayırarak öğrenmiş. Bu onun yenilikçi, ögrenmeyi ve program yazmayı seven birisi oldugunu gösterir. Karşınızdaki bu kişi büyük bir ihtimalle sadece Java bilen bir adaydan çok daha akıllı ve programcılık konusunda ileri seviyede.

Yanlış anlaşılmasın; Java'ya hakim olanlarda mutlaka iyi programcılardır. İyileri de var, iyi olmayanları da. Ama piyasının ihtiyacı olmayan bir dili öğrenen bir programcı çok daha fazla potansiyele sahip. Bu bir gercek! Ben böyle bir programcıyı tercih ederdim.

Sadece bir programlama dilinde (bu genelde piyasası olan bir dil olacaktır) takılıp kalmış olanların verdiği cevap hep aynı olacaktır: "Çalıştığım ortamlarda yeni bir programlama dili öğrenme fırsatı bulamadım." Gerçekten programcılığa gönül vermiş birisi o fırsatı hemen yaratır!

Bu konuda Paul Graham'ın yazısını okumanızı tavsiye ederim.

Neden Mikrodevre.com?

http://www.mikrodevre.com/2013/10/03/neden-mikrodevre-com/

Merhaba.

Yeni bir blogun doğuşuna şahit oluyorsunuz. İsmim Özcan Acar. Burada elektronik, mikro denetleyiciler ve mikro işlemciler hakkında öğrendiklerimi sizlerle paylaşmak istiyorum.

Asıl mesleğim programcılık. Son on beş yıldır serbest (freelance) Java programcısı olarak çalışıyorum. Bu konudaki tecrübelerimi KurumsalJava.com'da paylaşmaya çalışıyorum.

Elektroniğe olan ilgim 13-14 yaşlarında başladı. Lakin keşfedilmediği ve teşvik edilmediği için başlamasıyla son buldu. Aradan yirmi beş sene geçtikten sonra tesadüf eseri akıllı ev konusuyla ilgilenirken mikrodenetleyicilerle karşılaştım. Bir hazine bulmuş gibi oldum. Bir programcı olarak donanıma olan ilgim geniş. Yazdığım programlar 64 bit işlemcilerde birden fazla çekirdek üzerinde paralel koşuyor. Yinede elektronik kökenli olmadığım için donanım benim için soyut bir kavram. Nasıl çalıştığını biliyorum, lakin o kadar transistör, kondensatör, diyot ve direncin bir araya gelerek nasıl böyle bir kompleks yapıyı oluşturduğunu son detayına kadar anlamak benim için münkün değil. Bu sebepten dolayı bir programcı olarak donanımı anlamadaki faaliyetlerim mikroişlemcilerin nasıl çalıştığını anlamak ile sınırlı kaldı.

Bir programcı ya da konuyla hobi olarak ilgilenen birisi için bir mikroişlemci ile yapılabilecek fazla bir şey yok. Bir mikroişlemci tek başına bir işe yaramıyor, çünkü anlamlı bir şey yapabilmesi için etrafında erişebileceği hafıza ve disk gibi donanım elemanlarının olması lazım. Bir ana donanım kartı milyonlarca küçük parçacıktan oluşuyor. Elimi lehim çubuğunu alıp, ana donanım kartı üzerinde lehimleme işlemleri yapmak kartın bozulmasından başka bir netice vermez. Mikroişlemciler ve ana donanım kartları yüksek derece entegre sistemler olduklarından elektronik seviyede onlarla anlamlı bir şeyler yapmak çok zor, an azından benim için. Benim perspektivimden bakıldığında durum daha tekeri icat etmemişken bir uzay aracını tamir etmek gibi bir şey. Ne dediğimi anlıyorsunuz sanırım.

Durum mikrodenetleyicilerde çok farklı. Bu ufacık chipler günlük hayatımızın her yerinde. Çamaşır, kahve makinasından tutun, kullandığımız her türlü elektronik cihaz içinde mutlaka bir mikrodenetleyi var. Ufacık bir chip üzerinde hafıza, işlemci, içeriğini koruyan bellek gibi birçok şey var ve bu chipleri bir mikroişlemci gibi programlamak mümkün. Kilobyte ile ölçülen hafıza alanlarına sahip mikrodenetleyiciler ile hobi olarak proje geliştirmek mümkün. Akıllı ev konusundan bahsetmiştim. Evdeki bilimum teknik altyapıyı otomatize etmek için mikrodenetleyiciler biçilmiş kaftan. Birkaç transistor, direnç, LED ve diyot ile mikrodenetleyiciyi bir breadboard üzerinde bir araya getirip, bir kaç satır kod yazarak hemen gözle görülür bir netice elde etmek mümkün. Mikrodenetleyiciler işletim sistemi olmadan çalıştıkları için C/Basic/Assembler dilinde yazılan programların yapıları da çok basit. ATMega8 gibi bir mikrodenetleyiciyi birkaç Lira'ya satın almak mümkün. Ufak bir mikrodenetleyici devresi için yüzlerde Lira harcamak gerekmiyor.

Lakin mikrodenetleyicilerin kara bir tarafı da var. Temel elektronik bilginiz yoksa, mikrodenetleyiciler ile birkaç LED lamlayı kontrol etmekten öteye gidecek projeler geliştirmeniz

zor. Geliştirseniz bile temel elektronik bilgisi olmadan neyin nasıl çalıştığını tam olarak anlayamassınız.

Bunu anladığım ve mikrodenetleyicilere daha hakim olabilmek için mikrodenetleyicileri bir kenara bırakarak temel elektronik öğrenmeye karar verdim. Mikrodevre.com'da bu konuda edindiğim bilgileri sizlerle paylaşmak istiyorum. Bu benim için de öğrendiklerimi pekiştirmek için iyi bir fırsat olacak.

Elektronik kökenli değilim. Bilgisayar mühendisliği okudum. Yazacaklarım tamamen deneme, yanılma yoluyla edindiğim tecrübeleri yansıtacak. Çeşitli kaynaklardan faydalanarak bu konudaki bilgi seveyimi yükseltmeye çalışıyorum. Yazdıklarım hatalı ya da eksik olabilir. Sizden gelecek yorumlar içeriğin daha iyi anlaşılmasına ışık tutacaktır. Bu şekilde hep birlikte bilgimizi pekiştirebiliriz.

Çıkış noktam akıllı ev projem çerçevesinde evdeki teknik altyapıyı otomatize etmekti. Tesadüfen açtığım bu kapının arkasından mikrodenetleyiciler çıktı. O da bana elektroniğin kapılarını açtı. Duyduğum haz ve heyecanı kelimelerle ifade etmem çok güç. Bunun sebebi doğal olarak bu konulara olan ilgim. Eğer temel bir ilgiye sahip olmasaydım, bu konuyla bu kadar geniş çapta ilgilenmek için hazırlık yapmaz ve bu satırları yazıyor olmazdım. Yıllardır yazılıma olan ilgim bana bir şeyler öğretti ve bir noktaya kadar getirdi. Bundan sonra yazılımda daha iyi olabilmek için sadece yazılım konuları ile ilgilenmenin beni ileri götüreceğini zannetmiyorum. Yazılıma başka bir perspektiften bakmak gerekiyormuş, bunun farkına mikrodenetleyiciler ve elektronik ile tanıştığımda vardım. Bir konuda daha iyi olabilmek için, o konuyu destekleyici konularlar da ilgilenmek gerekir. İyi bir piyanist her gün piyanosunun başına oturup, pratik yapar. Bunun yanısıra müzik tarihi ile ilgili kitaplar da okur. Onun daha iyi bir piyanist olmasını sağlayan genel resmi görebilecek kabiliyete erişmesinde yatmaktadır. Sadece piyano çalarak iyi bir piyanist olunmaz. Aynı şekilde sadece program yazarak ya da programcılık dünyasında kalarak iyi bir programcı olunması mümkün değildir. Daha fazla yol katedebilmek için bu dünyanın dışına çıkmak gerekir. Ben bu dünyayı elektronik ve mikrodenetleyicilerde buldum. Başka birisi için bu dünya müzik, güzel sanatlar ya da spor olabilir. Bu yeni dünyanın kapılarını açacak olan temel ilgilerdir. Onlara danışmakta fayda var. Onlar size yolun nereye gittini göstecektir.

Açık Kaynağa Destek

http://www.kurumsaljava.com/2013/05/26/acik-kaynaga-destek/

Şüphesiz açık kaynak (open source) filozofisi biz programcıların hayatını derinden etkiledi. Açık kaynağın bize sağladığı bariz iki avantaj var. Bunlar:

- Başkalarının, bu başkaları çoğu zaman usta olarak tabir edebileceğimiz yetenekli programcılar, yazdığı kodlara bakarak kendimizi programcı olarak geliştirebiliriz.
- Açık kaynaklı programları lisans bedeli ödemeden kullanabiliriz.

İkincisi çok bariz olarak karşılaştığımız bir durum. Hibernate, Spring, Eclipse.. bu meşhur açık kaynaklı programların kullanılmadığı proje sayısı çok az ya da yok gibi.

Birileri, bir yerlerde oturmuşlar, bizim hayatımızı kolaylaştırmak için gece, gündüz demeden açık kaynaklı programlar oluşturmak için çalışıyorlar. Neden bunu yapıyorlar acaba?

Bunun çeşitli sebepleri olabilir. Örneğin:

- İdealistler ve dünyayı iyileştirmek istiyorlar.
- Bu işten para kazanmak istiyorlar. Açık kaynaklı ürünleri satmak imkansız, ama destek sağlayarak para kazanmak mümkün.
- Bir açık kaynaklı ürün oluşturarak piyasada tanınmak ve ün salmak isteyenler olabilir.
- IBM gibi firmalar açık kaynaklı ürünlerin oluşturulmasına destek vererek Microsoft gibi firmaların oluşturdukları monopolleri kırmak isteyebilirler. Nitekim Eclipse IBM tarafından geliştirilmiş ve daha sonra açık kaynak olarak bizlerin kullanımına sunulmuştur. IBM'in amacı Java ekosistemini güçlendirmek ve Java'nın .NET platformuna alternatif oluşturmasını sağlamaktır.
- Belli bir teknolojinin yayılmasını sağlamak için açık kaynaklı programlar oluşturulabilir. Örneğin Java'nın daha yaygın hale gelmesini sağlamak için sahip olduğu ekosistemi oluşturan hemen, hemen her parça açık kaynaklıdır.
- Bir ürünün yayılmasını sağlamak için ürün açık kaynaklı yapılabilir. Örneğin JetBrains firmasının IntelliJ IDEA isminde açık kaynaklı kod geliştirme ortamı bulunuyor. Community Edition ismini taşıyan bu sürüm programcıların temel ihtiyaçlarını karşılarken, JetBrains bu ürünün profesyonel sürümlerini oluşturup, satarak para kazanıyor.

Bu listeye eklenti yapmak zor değil. Benim ilk düşündüğümde aklıma gelen noktalar bunlar. Peki bize bu kadar muazzam getirisi olan bir akım için biz programcılar neler yapıyoruz ya da yapabiliriz? Sayalım:

- Açık kaynaklı ürünleri kullanarak belli teknolojilerin öne çıkmasına destek verebiliriz. Linux örneğinde olduğu gibi bu monopollerin kırılmasını ya da dengede tutulmasını sağlayabilir.
- Açık kaynaklı ürünlerin geliştirilmesine katkıda bulunabiliriz. Açık kaynaklı projelerin çoğu gönüllü olan programcılar sayesinde geliştirilebiliyor.
- Açık kaynaklı ürünlerin kullanımında keşfettiğimiz hataları gerekli yerlere bildirerek, bu hataların giderilmesine katkıda bulunabiliriz. En ideal olanı bu hataları gidererek, projeye doğrudan destek saglamaktır.

- Açık kaynaklı projelerin sürdürülebilmesi için parasal destekte bulunabiliriz, yani bağış yapabiliriz.

Eclipse benim yıllardan beri kullandığım açık kaynaklı bir araç. Bugün Eclipse Foundation için belli bir miktarda bağış yaptım. Ne kadar bağışladığım önemli değil. Önemli olan bağışın kendisi. Yıllardan beri freelancer olarak çalıştığım birçok projede Eclipse'i kullandım. Eclipse olmasaydı başka bir ürünü kullanırdım. Ama Eclipse vardı ve onu kullandım. Bunun karşılığında hiçbir bedel ödemedim. Eclipse bana programcı olarak çok şeyler kattı. Para kazanıp, hayatımı sürdürmeme destek oldu. Yaptığım bağış okyanusta bir damla bile olsa, damlaya, damlaya göl olur sözünü unutmamak lazım. Bağışı yaptıktan sonra Friend Of Eclipse ünvanina sahip oldum :) Ne güzel….

Çoğu programcı kullandıkları araçların açık kaynaklı olduklarına pek kafa yormadan hergün bu araçları kullanarak, işlerini yapıyorlar. Aynı şekilde çoğu programcının kullandıkları açık kaynaklı programların kodlarını inceleyip, daha iyi bir programcı olmak için çaba sarfettiklerini düşünmüyorum. Çoğu programcı bu ürünleri kullanırken keşfettikleri hataları bir yerlere bildirmiyorlar, çünkü birileri bunu yapar diye düşünüyorlar. Çoğu programcı commiter olarak açık kaynaklı projelere destek vermiyor, çünkü bunun için zamanları yok ya da zamanlarının olmadığını düşünüyorlar. Çoğu programcı lisans bedeli ödememek için bahsettiğim açık kaynaklı programları kullanıyorlar. Hepsini topladığımızda açık kaynaklı akımı desteklemeyen, egoistçe kullanan ve hiçbir şey geriye vermeyen bir programcı kitlesi oluşuyor. Ama üzülmesinler, yapabilecekleri bir şey var. Hiçbir şey yapamıyorlarsa, o zaman açık kaynaklı projelere bağış yapabilirler. Bu şekilde fayda sağladıkları projelerin devam etmelerini sağlayabilirler. Kullanıkları açık kaynaklı ürünler olmasaydı hayatlarının programcı olarak ne kadar daha zor olacağını düşünmeleri yeterli.

Her programcıyı kullandığı açık kaynaklı araçları desteklemek için bağış yapmaya davet ediyorum. Bağış bir Lira'da olsa damlaya damlaya göl olur.

Not: Bağış yapanlar bu yazıya yorum bırakarak, hangi açık kaynaklı projeyi desteklediklerini bildirebilirler.

Neden Her Programcının Bir Blog Sayfası Olmalı?

http://www.kurumsaljava.com/2012/06/06/neden-her-programcinin-bir-blog-sayfasi-olmali/

Blog kelimesi Web Log kelimelerinin kısaltılmış halidir. Bir blogu dijital ortamda tutulan bir günlük olarak düşünebiliriz. Aşağıda sıraladığım sebeplerden dolayı her programcının bir blog sayfası olmalı:

- Blog bir programcının özgürce kendini ifade etmesini sağlar. Söyleyecek bir şeyi olan bir programcı blogu aracılığı ile okurları ile buluşur. Programcı okurlarından geribildirim niteliğinde yorumlar alır. Bu kendi düşüncelerini yapıcı yönde sorgulamasını sağlar.
- Bir blog tutmanın en olumlu yanı bilgi paylaşımını teşvik etmesidir. Blog programcıyı daha paylaşımcı olmaya iter, yıldız programcı olma tehlikesine önler.
- Sahip olduğu bilgi ve tecrübeyi blogu aracılığı ile paylaşan programcı diğer programcılara bilgi eksilerini gidermelerinde yardımcı olur. Ne kadar çok programcı bilgi paylaşımında bulunursa, o kadar çok okur bundan faydalanır.
- Verimli olmanın önündeki en büyük engellerden birisi dahil olmak zorunda olunan e-posta trafiğidir. Birçok kişiye aynı konularda tekrar tekrar aynı cevaplar verilir. Sayfalar dolusu e-posta iletisi yazmak yerine, programcı bunu bir defaya mahsus kendi blog sayfasında ya da çalıştığı firmanın wiki sayfasında yaparak, ondan bilgi isteyenlere bu blog ya da wiki sayfasının linkini gönderebilir. Bu şekilde hem aynı cevapları yazarak zaman kaybetmez, hem de sahip olduğu bilgilerin merkezi bir yerde toplanmasını sağlar.
- Blog yazılar zaman içinde programcı için kaynak referans haline gelir ve kendi çözümlerini tekrar gözden geçirme fırsatı verir.
- Blog yazmak günlük tutmak gibidir. Zamanla blog kişisel gelişimin aynası haline gelir.
- İşverenler mülakat öncesi programcının internetteki izlerini araştırırlar. Güncel içeriğe sahip bir blog sahibi bir programcı mülakata 1-0 önde girer. Bloğu aracılığı ile bir programcı piyasa değerini artırmış olur.
- Blog yazan programcının ana dilini kullanma kabiliyeti genişler. Bu günlük işlerine olumlu olarak yansır. Blog yazan programcı daha iyi kod ve doküman yazar ve ana dilini daha iyi kullanır.
- Blog yazıları ile belli bir kesime hitap eden programcı bir zaman sonra güvenilir ve sektörde tanınan bir referans haline gelir.
- Blog yazan programcı kendisi ile aynı fikirdeki diğer programcılarla tanışma ve bir araya gelme fırsatı yakalar. Böylece programcının çevresi genişler, sosyal aktiviteleri artar, kendisi ile aynı fikirdeki diğer programcılarla fikir alışverisinde bulunur.

Bir blog sayfanız yoksa, hemen bir blog sayfası oluşturun. Blog yazı yazmayı zaman kaybı olarak görmeyin ya da yazdıklarımı kim okur demeyin. Bunu ilk etapta kendi kişisel gelişiminiz için yapıyorsunuz. Eğer kaliteli içerik oluşturabilirseniz ki zaman içinde bu şüphesiz gerçekleşecektir, o zaman okuyucu kitlenizin çok hızlı büyüdüğüne şahit olacaksınız.

Blog yazmanın size kattığı artı değeri çok kısa bir zamanda hissetmeye başlayacaksınız. Blog yazmadan bulu bilemessiniz :)

Organizasyonel Değişim

http://www.kurumsaljava.com/2013/01/04/organizasyonel-degisim/

Yazılım camiasında son zamanlarda dikkat çeken bir değişim furyası var. Cevabı aranan soru şu: Yazılım ekibi nasıl daha verimli hale getirilebilir? Bu aslında organizasyonel bir değişimin gerekli olduğu bilincinin oluştuğu anlamına geliyor. Yöneticiler ekiplerini daha çevik hale getirmek için çeşitli yöntemlere başvuruyorlar. Bunların başında örneğin ekibin topluca eğitilmesi geliyor.

Her eğitim şüphesiz ekibe ve bireylerine bir şeyler katıyor. Ama bunu ölçmek çok zor. Bunun yanı sıra cevaplanması gereken bir soru daha var: Eğitim ile istenilen değişiklik sağlanabilir mi?

Alınan eğitimler değişimin sadece başlangıcı niteliğinde olabilir. Bunu yeni bir dil öğrenme süreci ile kıyaslayabiliriz. Öğrenilen her yeni kelime, kelime hazinesini genişletir. Pratik yapılarak kullanılmayan bir kelime hazinesi hiçbir şey ifade etmez ve değeri yoktur. Yazılım ekiplerinin tükettikleri eğitimler genelde sahip oldukları metot hazinelerini genişletmekle birlikte, bu metot hazinesinin günlük işlerde pratiğe dökülmemesi istenilen değişimi sağlayamamaktadır. Değişime gitme çabalarının son bulduğu nokta burasıdır.

Değişim yönetilmesi gereken bir süreçtir. Tesadüflere ve ekibin inisiyatifine bırakılabilecek bir şey değildir. Bir yazılım ekibi rasyonel düşünebilen, soyutlayabilen, problem çözme yetisi gelişmiş bireylerden oluşabilir. Ama unutmamak gerekir ki ekibi oluşturan bireylerin kendilerine has alışkanlıkları vardır. Değişim için gerekli baskı azaldığında ya da ortadan kalktığında bireyler bu alışkanlıklarına geri dönerler. "Ekip belli bir eğitimi aldıktan sonra artık çalışmalarını bu yönde adapte edecektir, dışarıdan müdahale etmeye gerek yok" şeklinde düşünmek bir yanılgıdır. Bu şekilde ekibin inisiyatifine bırakılan değişim süreci başladığı gibi son bulur.

Bunun örneklerini danışmanlık hizmeti verdiğim şirketlerde gördüm. Örneğin bir defasında yazılım sürecini daha çevik hale getirmek için ekibin test güdümlü yazılım eğitimi alması tavsiyesinde bulundum. Ekip gerekli eğitimi aldı. Ekip içinde test yazma duyarlılığı arttı, lakin ekibin hiç bir bireyi gerçek anlamda test güdümlü yazılım yapmadı. Kısa bir zaman sonra yazılımcılar eski davranış biçimlerine geri döndüler. Şimdi yine eskiden olduğu gibi kod yazıyorlar. İstenilen değişimin gerçekleşebilmesi için şu adımların atılması gerekir(di):

- Değişim bir süreç olarak algılanmalı ve buna göre de yönetilmelidir. Bu birilerinin süreç gidişatından sorumlu olması anlamına gelmektedir. Değişikliğin meydana gelmesinden sorumlu olan bu şahıs ekipten gerekli geri bildirimi alarak, değişikliğin hangi safhada olduğunu takip etmek zorundadır. Bu geri bildirim alınması gereken önlemlerin tespiti ve planlanması açısından önem taşımaktadır.
- Değişimin gerçekleşmesi için ekip içinde değişimin yaşanması gerekir. Ekip liderinin alınan eğitim sonrasında kazanılan yeni yetilerin uygulanmasını teşvik ve kontrol etmesi gerekir. Kendisi takım arkadaşlarına iyi bir örnek olarak, değişimi yaşadığı sinyalini vermelidir. Körle oturan, şaşı kalkar misali, ekip bireyleri liderlerini takip edip, değişimin vücut bulmasını sağlayacaklardır.

- Ekip içinde bilgi ve becerilerin hızlı ve eşit bir şekilde dağılmasının en kolay şekli programcıları eşli programlama yapmasıdır. Eşli programlama programcıların kendilerini ve karşılıklı birbirlerini kontrol etmelerini mümkün kılar.
- Değişimi getirecek sürecin doğru işleyip, işlemediğini kontrol etmek için ekip dışından destek alınmalıdır. Belli aralıklarla uzman bir koç gidişatın doğru olup, olmadığını kontrol edip, gerekli rota düzenlemesini yapabilir. Süreç tıkandığında da koç oluşan düğümleri çözecek ve değişime giden sürecin önünü açacaktır.
- Değişim sürecinin tamamen durduğu anlardan birisi programcıların kriz anında eski alışkanlıklarına geri dönmeleridir. Bu ne yazık ki önlenmesi çok zor bir durumdur. Daha önceki bir yazımda bahsetmiştim. Kriz anlarında otomatik olarak devreye giren davranış biçimlerinin önüne geçmenin tek yolu pratik yapmaktan geçmektedir. Programcı kod kata ismi verilen kod pratiklerini devamlı yaparak, krizle ortaya çıkan verimsiz davranış biçimlerini zamanla beyninde silebilir. Kod kataları değişim sürecinin bir parçası haline getirilmelidir.

Yazılımcılar yetenekli insanlar. İstenilen bir şeyleri kavramaları uzun zaman almaz, lakin uygulamada zorluk çekebilirler. Yönetimin yazılım ekibine gerekli desteği sağlaması ve bu sorumluluğu onlarla paylaşması, değişimin daha kolay gerçekleşmesini kolaylaştıracaktır.

Böyle Girişimcilik Olmaz!

http://www.kurumsaljava.com/2012/08/17/boyle-girisimcilik-olmaz/

Bu yazıyı okuyunca daha önce yazdığım Melek Programcılar ve Şeytan Yatırımcılar başlıklı yazımı anımsadım. Orada genç girişimcilerin nasıl şeytan yatırımcılar tarafından üç kağıda getirilip, sömürüldüklerinden bahsetmiştim. Durumun başka bir boyutunu da paylaştığım yazı gösteriyor.

Eğer devamlı hesabını yapacak isem, param yetiyor mu, ya da beş sene sonra hala yeter mi diye, o zaman kendime özel sağlık sigortası yaptırmam doğru değil. Maddi açıdan bu hesabı devamlı yapmak zorunda olmayacak güçte isem, özel sağlık sigortası yaptırabilirim. Aynı şekilde sermayem yetmiyorsa, girişimci olmam da bir hayal. Kendilerine melek yatırımcı diyerek, oraya, buraya üç beş bin TL yatırım yapıp, bunu yatırım zanneden birilerinden medet ummak çok daha büyük bir hayal.

Amerika ya da Avrupada'ki başarılı bir internet girişimini birebir kopyalarak Türk internet pazarında başarılı olacağını düşünen herkes girişimci trenine binmeye çalışıyor. Çoğu yeterli sermaye, pazarlama ve işçi yönetimi bilgisi sahibi değil. Sermayeleri yetmediği için çok kısa bir zamanda işi büyütüp, onların tabiri ile exit yapmak, yani kurdukları işi çok paralara satma hayaliyle yaşıyorlar. Nihayi amaçları exit, başka hiçbir şey değil!

Sanki Türkiye'de programcı eleman eksikliği varmış gibi, birde eleman bulmakta sıkıntı çektiklerinden yakınıyorlar. Hak ettiği maaşı verdikten sonra Türkiye'de programcı bulmak hiç zor değil. Ama bu bahsettiğim internet girişimcilerinin o kadar az sermayaleri var ki, ne yiyip, içtiklerine bile dikkat etmeleri gerekiyor. Kalkıp ayda 8-10 bin TL maaş bir kalifiye programcıya nasıl versinler. Bu yeni nesil girişimcilerin programcılardan beklentileri çok yüksek, ama vermek istedikleri karşılık çok düşük. Birkaç bin TL'ye çalışacak programcı bulamamaları çok normal. Neden şikayet ediyorlar?

Hadi diyelim istedikleri şekilde bir programcı buldular. Bu programcı arkadaşın allah yardımcısı olsun. Ne fazla mesai yapmadığı kalır, ne cumartesi günleri çalışmadığı. Bir programcı günde sekiz saat ve haftada beş gün çalışır, NOKTA! Programcılık kolay bir iş değil. Beynin regenere olabilmesi için haftanın iki günü dinlenmesi gerekir. Bunun yanısıra programcının kendisini geliştirebildiği zaman dilimlerine ihtiyacı vardır. Bir programcıyı işe alıp, günde 9-10 saat çalıştırıp, cumartesi günü de işe gelmeye zorladığınız zaman bilin ki, yazılım ürününde meydana gelen hatalar çalışma süresinin uzunluğu ile doğru orantılı büyüyecektir. İnsanları bu şekilde sömüremessiniz.

Az sermayeli internet girişimcilerinin yegane dertlerinin exit olduğunun bir ibaresi de bu yazı. Adamlar Türkiye'de istedikleri çapta vurgun yapamayacaklarını anladıkları için, apar topar Türk piyasasından çekilme kararı almışlar. Bu adamlar bir iki senedir Türk piyasasındalar. Dört yüzden fazla çalışanı olan bir firma, neden apar topar tüm aktivitelerini durdursun? Çünkü sermayeleri yetmediği için. İşi hemen büyütelim, Türkiye'den ya da Avrupa'dan bir firma bizi satın alır, exit yaparız hayali ile yaşadıkları için. Bu dört yüz tane çalışanın canlarını yakmaya ne hakları var? Ama merak etmeyin. Bu tipler aynı numarayı dünyanın değişik bölgelerinde de

yapıyorlar. Aynı kişiler Almanya'da da dokuz yüz kişiyi anında kapıya koydular. Böyle girişimcilik olmaz. Olsa bile ben bunların söylediklerini ciddiye almam, çünkü yaptıkları işin özünde insana hizmet ve saygı yok. Sadece kendi ceplerini düşünüyorlar, kefenin cebi varmış gibi!

Bahsettiğim ilk yazıda programcı eleman sıkıntısını gidermek için devletten ve üniversitelerden yardım isteniyor. Devlet bu konuda ne yapabilir? Üniversitelerini kurmuş, iyi ya da kötü insan yetiştirmeye çalışıyor. Üniversiteler bu konuda ne yapabilir? Hiçbir şey. Üniversite programcı yetiştirmez, yetiştiremez. Programcı anca bir usta programcının yanında yetişebilir. Ondan doğru ve yanlışları görerek iyi bir programcı haline gelebilir.

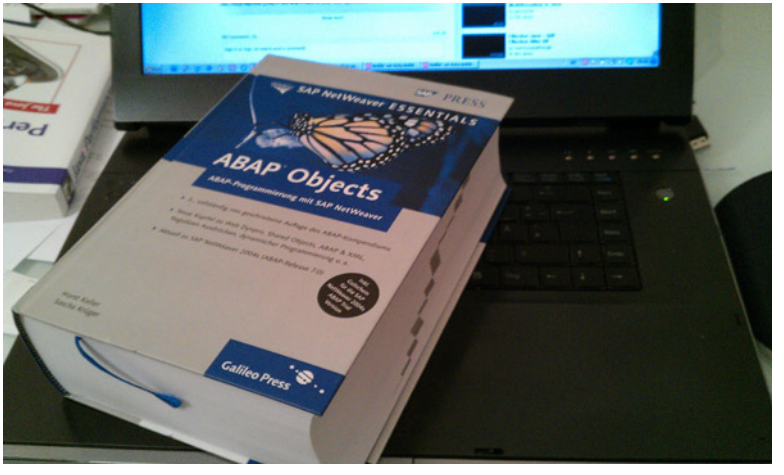
Bu yeni nesil girişimciler fazla emek sarfetmeden insan kaynaklarının kaymağını yemek istiyorlar. Yok böyle bir şey! Devlete ya da üniversitelere programcı yetiştirin diye laf söylemek yerine, yaz okulları açıp, usta programcıların üniversite ögrencilerine ya da yeni mezunlara ders vermesini sağlasınlar. Ama bunu yapamazlar, çünkü sermaye örtüleri çok ince. O zaman ne kadar para, o kadar köfte, sorry! Ama size hiç köfte yok, şekil A'da görüldüğü gibi.

Yeni Bir Programlama Dilini Öğrenmenin En Kolay Yolu

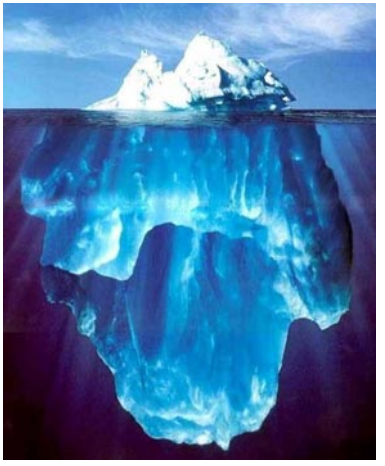
http://www.kurumsaljava.com/2012/05/25/yeni-bir-programlama-dilini-ogrenmenin-en-kolay-yolu/

Geçenlerde bir Abap kitabı aldım. Güya Abap ögreneceğim! Abap SAP tarafından geliştirilmiş bir programlama dili ve SAP'nın ERP sistemlerini programlamak için kullanılıyor.

İlk yüz sayfayı hızlıca geçtim. İlk "Merhaba Dünya" programımı yazdım. Diğer dillerden fazla bir farkı yok. İşte bildiğin bir dil. Geriye 1150 sayfa kaldı. Gel de oku bakalım. Movitasyonum dibe vurmuş durumda. Masamda kitap bana bakıyor, ben kitaba bakıyorum. "Sana o kadar para verdim, en azından masamda durma şerefine nail oluyorsun" diyorum kitaba ara sıra. Okunacak diğer kitaplar yanında Abap kitabı zaten çok soluk kalıyor. İlgimi çeken o kadar başka çok sey var ki! Uğraşacak mutlaka başka bir şeyler buluyorum. Kitabın masamın üzerinde bir yerlerde duruyor olması bana yetiyor. *Bir gün mutlaka geriye kalan 1150 sayfayı okuyacağım ve bu işi çözeceğim" diyorum kendi kendime. Kitap da bana "sen kendini kandırmaya devam et bakalım, beni kesinlikle bir sefer daha eline alıp, Abap'a bir şans daha vermeyeceksin" diyor. Kitap rafında o kitap bir süs olarak kalacak. Bir şarkı sözünde denildiği gibi "sevemedim karagözlüm seni doyunca.*



Yeni bir dil ögrenmek istiyorsanız kitapları ya da kursları unutun! Yeni bir dil ya da teknoloji büyük bir buzdağı gibidir. Okuduğunuz kitaplar ya da katıldığınız kurslar buzdağının su üstünde kalan kısmını anlatırlar. Çoğu bunu bile beceremez. Çoğu programcının yeni bir dile giriş mecarası, buzdağının su üstünde kalan kısmı seviyesinde kalır. İşin kötü tarafı, su üstünde kalan kısma hakım olmanın programcıda "sen bu işi çözdün, artık gurusun" hissini uyandırmasıdır. Bu dilin müstakbel programcısı ne kadar büyük bir yanılğı içindedir.



1999 senesinde BizimAlem.com'u kafamda tasarlamaya başladığımda dünyadan haberi olmayan, Java dünyasından habersiz çaylak bir programcıydım. Bu durum çok kısa bir zamanda değişti, çünkü BizimAlem'in gelişebilmesi için değişmek zorundaydı.

BizimAlem.com yayın hayatına başladıktan kısa bir zaman sonra popüler oldu. Tabi o zamanlar feysbuk, meysbuk yok. Herkes balıklama BizimAlem'e atladı (Facebook çıktıktan sonra da balıklama oraya atladılar; Eye- ya da Hairbook çıktığında da oraya atlayacaklar; yani sazan gibi Facebook'a yatırım yapanlar bu yatırımın üstüne bir bardak soğuk limonata içsinler). Kullanıcı sayısı birden tavana vurdu. Binlerce insan eşli zamanda online olmaya başladı. Bununla birlikte benim baş ve karın ağrılarım da başladı. Uykusuz sunucuların başında çok geceler geçirdim. Ne yapsam yetmez oldu. Program hataları ile boğuşmaktan, uygulama sunucularının bir müddet sonra takılıp işlemez hale gelmesinden, kullanıcılardan gelen şikayetlerden bıktım usandım.

Bu olup bitenler bana bambaşka bir dünyanın kapılarını açtı. Yaşadıklarımı BizimAlem.com – Bir Sistemin Tasarlanış Hikayesi başlıklı yazımda okuyucularımla paylaşmaya çalıştım. Bir programcı olarak dünyayı başka türlü algılamaya başladım. Anladım ki programcı olma yeteneği sorun çözmekle pekişmekteymiş. Bir konuda uzmanlaşmak istiyorsanız başınıza BizimAlem.com gibi bir bela almanız gerekiyor. Hayal bile edemeyeceğiniz işlerle ugraşmak zorunda kaldığınızda bakın bakalım buzdağının dibine doğru nasıl yolculuğa çıkıyorsunuz ve kullandığınız programlama dilini yiyip, yutuyorsunuz.

Kısa bir zaman önce BTSoru.com isimli projem hizmet vermeye başladı. BTSoru.com hazır ve açık kaynaklı bir yazılım kullanıyor. Python dili ve Django çatısı kullanılarak oluşturulmuş. İkisinden de haberim yok. Ne şimdiye kadar Python kodu yazdım, ne de Django ile bir web projesi geliştirdim. "Neden anlamadığınız bir teknoloji yığınını kullanıyorsunuz" sorusunu sorduğunuzu duyar gibiyim. Cevabı çok basit: Python ve Django'u öğrenmek için.

Bir dili ya da teknoloji yığınını öğrenmenin en kolay yolu, bu dil ya da teknoloji yığını ile herkesin kullanabileceği bir proje geliştirmekten geçer. Kimsenin kullanmadığı bir program parçası suskundur. Programcıya geribildirim sağlamaz. Daha da kötüsü sorun yaratmadığı için

programcının başını ağrıtmaz. Başı agrımayan programcı buzdağının derinliklerine doğru yolculuğa çıkmaz. Bu yüzden mutlaka kullanıcılardan geribildirim almak gerekir. Onlar çok kısa bir zamanda gidişatın yönünü belirlerler. İstekleri ve şikayetleri bitmediği için programcı devamlı sorun çözmek zorunda kalır. Bu şekilde kullandığı programlama dilinin ya da teknoloji yığınının derinliklerine dalmak zorunda kalır. Gerçek anlamda öğrenim ve hakimiyet bu noktadan itibaren başlar.

Şimdiden BTSoru.com kullanıcılarından bir sürü istek gelmeye başladı. Yani beni Python dilini öğrenmeye zorluyorlar. Platformu geliştirmek için Python'u ve Django çatısını öğrenmem, anlamam ve kullanabilmem gerekiyor. Ben ne yaptım? İlk önce Python kodunu yazabilecegim bir araç (DIE – Integrated Dev. Environment) aradım. Bir Eclipse kullanıcısı olduğum için, öncelikle Eclipse için bir Python plugini var mı, onu araştırdım. PyDev isminde bir Eclipse plugini buldum. Daha sonra internette Django örnek uygulamalarını araştırdım. Django ile ilk uygulamanı geliştir pratiğini yaptım. Şimdilerde BTSoru.com için ufak defek değişiklikleri yapabiliyorum. Acemiyim, ama bu durum yakında değişesek. Sayfaların yüklenme hızını artırmak için caching mekanizmaları kullanmam gerekiyor. BizimAlem'de Memcached kullanmıştım. Aynı şekilde gerekli değişiklikleri Django içinde de yapmam gerekiyor. Django'nun içinde şimdiden gezinmeye başladım bile. Bunları bir kitapta okumuş olsaydım, gerçek hayatta pratiğini yapmadan "caching faydalı bir şeymiş" deyip geçiştirirdim büyük bir ihtimalle. Ama şimdi bir ihtiyaç haline geldiği için bu konuya yoğunlaşmam gerekiyor.

Bu şekilde bir dili öğrenmek, kitap okuyarak öğrenmekten çok daha zevkli ve verimli. Kitaplar genelde "merhaba dünya" örneğinden çok ileri gidemedikleri için benim ilgimi çekmiyorlar. Kitaplarda yer alan örneklerin çoğu gerçek hayatla ilişkili değil. Örnek olmaktan ileri gidemiyorlar. Bu örnekleri kullanarak gerçek problemleri çözmek mümkün değil. Ayrıca kısa bir zaman sonra bu tür kitapları bir kenara koyup, daha enteresan bulduğum şeylere yoğunlaşıyorum. Bu şekilde bir dili verimli öğrenmem çok zorlaşıyor. Oysaki BizimAlem.com ve BTSoru.com örneklerinde olduğu gibi bir şeyleri ögrenmeye zorlandığımda durum değişiyor.

Benim için BizimAlem.com halen büyük bir laboratuvar. Öğrenmek istediğim herhangi bir teknoloji yığınını BizimAlem'e entegre ediyorum ve kenara çekilip seyretmeye başlıyorum. Oluşan hataları gidererek ve kullanıcılardan gelen geribildirimi değerlendirerek gerçek şartlarda çalışan bir uygulama hakkında öğrenmem gereken herşeyi çok kısa bir zamanda öğreniyorum. Bu bana kullandığım teknoloji yığınını ya da programlama dilini çok değişik açılardan inceleme fırsatı veriyor. Hangi iş için neyi kullanmam gerektigini daha iyi anlayabiliyorum. Bu tür bilgileri bir kitaptan okuyarak ögrenmek imkansız. Öğrenmek için gerçek şartlarda pratik yapmak gerekiyor. Bu konudaki verimliliğinizi artırmak için birilerinin sizi zorlamasını sağlayın.

Bir sonraki web projem ya Lisp ya da Clojure tabanlı olacak. Bu dilleri ögrenmem lazım ;-)

Sevgiyle Kalın…

Matrix'de Yaşayan Programcılar

http://www.kurumsaljava.com/2012/05/12/matrixde-yasayan-programcilar/

Hemen hemen her programcının Matrix filmini seyrettiğini düşünüyorum. Star Wars gibi Matrix filmi de biz yazılımcılar için bir kült. Biraz abartı da olsa fikir olarak çok enteresan, en azından bir yazılımcı için. Matrix'de kullanılan yazılım sistemi dikkat çekiyor. En çok ilgimi çeken dejavü olarak isimlendirilen yazılım hataları (bug) ve Neo'nun bir tren istasyonunda hapis kalması ve trene binmesine rağmen tekrar tekrar aynı istasyona geri dönmesi, yani bir nevi for döngüsü olmuştur. Bir for döngüsünün bu kadar güzel görselleştirilmesi beni çok etkilemişti. Böyle bir sistemin entegrasyon testleri nasıl yapılıyor acaba?

Gelelim gerçek hayattaki Matrix'e. Java ile program yazan programcılar da yazdıkları programlar gibi bir Matrix içinde yaşarlar. Bu dünyanın ismi JVM – Java Virtual Machine yani Java Sanal İşlemcisi'dir. JVM C++ dilinde yazılmış bir programdır. Bir Java programı javac.exe ile derlendikten sonra byte code ismi verilen bir ara sürüm oluşur. Byte code, ana işlem biriminin (CPU – Cental Processing Unit) anlayacağı cinsten komutlar ihtiva etmez, yani klasik Assembler değildir. Java byte code sadece JVM bünyesinde çalışır. JVM, derlenen Java programı için ana işlemci birimi olma görevini üstlenir. Bu özelliginden dolayı Java programlarını değişik platformlar üzerinde çalıştırmak mümkündür. Her platform için bir JVM sürümü Java programlarını koşturmak için yeterli olacaktır. Bu sebepten dolayı Java "write once, run anywhere – bir kere yaz, her yerde koştur" ünvanına sahiptir.

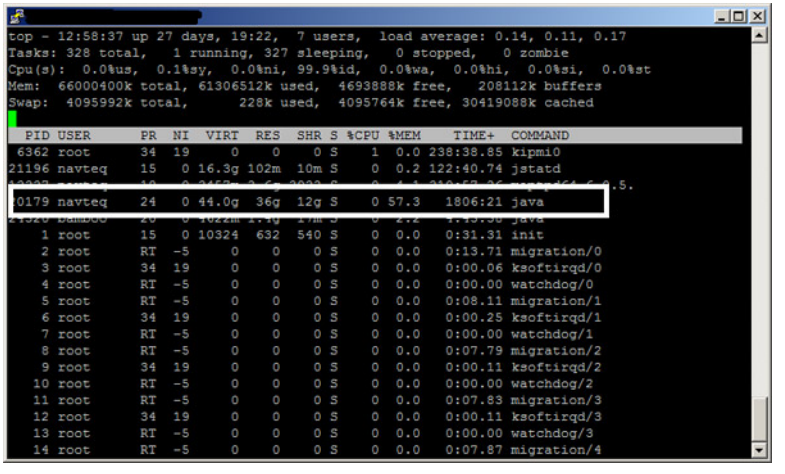
Java programları java.exe komutu kullanılarak koşturulur. java.exe işletim sistemi bünyesinde bir JVM meydana getirir yani Matrix'i oluşturur. Bu Matrix Java programının yaşaması için gerekli ortamı ihtiva eder. Sıra dışı olmayan Java programları bu Matrix içinde dış dünya ile ilişkileri kesik bir şekilde yaşayıp giderler. Onların ihtiyaç duyduğu herşeyi Matrix onlara sunar. Java programları hangi işletim sistemi ya da hangi donanım üzerinde koştuklarını bile zorda kalmadıkça bilmezler. Onlar için her donanım üzerinde bir integer 4 byte yani 32 bittir. Bu sebepten dolayı Java'da C/C++ dan tanıdığımız unsigned int bulunmaz. İnteger hangi donanım olursa olsun 32 bittir, işletim sistemi ya da donanıma göre değişmez. Bu Java dilini tasarlayan mühendislerin Java programcılarının hayatını kolaylaştırmak için aldıkları bir tasarım kararıdır. Bir Java programının gördüğü alt yapı her zaman aynıdır. Java programcıları bunu bildikleri

için Matrix haricinde olup bitenlere pek önem vermeden kodlarını yazarlar ve Matrix içinde yaşamaya devam ederler, **taki Morpheus gelip programcıya mavi ve kırmızı hapları taktim edene kadar.**

Klasik kurumsal Java projelerinde çalışan Java programcısı Matrix'in dışında olup bitenlerle ilgilenmez. JVM ona ihtiyaç duyduğu herşeyi sağlar. O yüzden programcının tercihi mavi hap ve Matrix içinde yaşamaya devam etmek olur.

Java sadece kurumsal projeler için kullanılmaz. Sıra dışı projelerde de Java'ya rastlamak mümkündür. Şu an çalıştığım proje bunun en iyi örneklerinden birisidir. Navteq/Nokia firmasının GeoCoder (http://maps.nokia.com) isminde, harita üzerinde lokasyon arama yapan bir servisi bulunmaktadır. Bu servis aynı zamanda Bing ve Yahoo tarafından kullanılmaktadır. Kısa bir zaman sonra Facebook tarafından kullanılma planları yapılmaktadır. Ben lokasyon arama işlemlerinin programlandığı ekipte çalışıyorum. Bu gibi projelerde Morpheus'un verdiği kırmızı hapı yutup, Matrix'in dışına çıkmak gerekiyor, aksi taktirde Matrix, yani JVM içinde kalarak uygulamanın tabiyatındaki sıra dışılığı anlamak ve kodlamak mümkün değildir. Bu sıra dışılık programcıyı çok daha değişik kategorilerde düşünmeye ve değişik disiplinlerde çalışmaya zorlamaktadır.

Nokia'nin lokasyon arama servisi için dünya çapında altı değişik hosting lokasyonunda 300 den fazla VM (virtual Machine – sanal sunucu) kullanılıyor. Aşağıdaki resimde de görüldüğü gibi her JVM işletim sistemi bünyesinde 40 GB'den daha fazla yer kaplıyor. Neden JVM için bu kadar büyük bir hafıza alanının kullanıldığını anlamak için, uygulamanın tabiyatını anlamak gerekiyor.

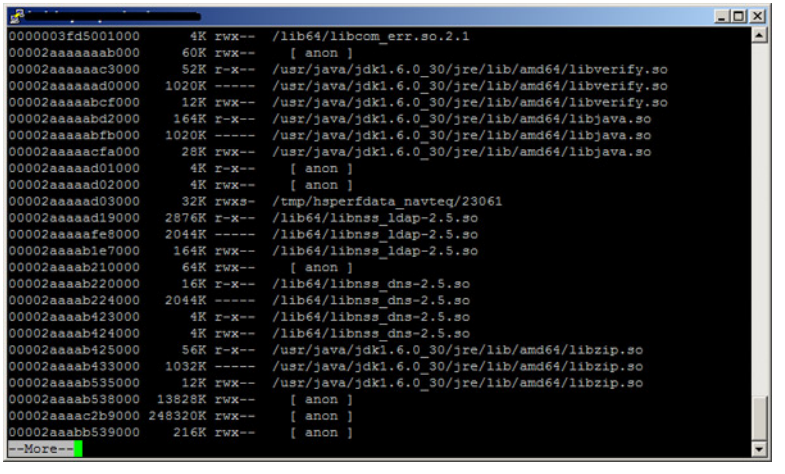


Yukarıda yer alan resimde işletim sistemi (RedHat Linux) JVM için yeni bir işlem (process) oluşturmuştur. Bu işlemin işletim sistemi bünyesindeki hafıza büyüklüğü toplamda (VIRT – Virtual – sanal) 44GB'dir. JVM için kullanılan hafıza (Heap size) 5 GB'dir. Heap ayarı -Xmx5g ile yapılmaktadır. Peki geriye kalan 39 GB neyin nesidir? Bunu anlamak için Matrix'in dışına çıkmamız gerekiyor, çünkü geriye kalan 39 GB Matrix'in dışında olup bitenleri temsil

etmektedir.

Lokasyon arama servisi için klasik veri tabanı sistemi kullanılmamaktadır. Klasik bir veri tabanı sisteminin kullanılması ve lokasyon arama işlemlerinin bu veri tabanı üzerinden yapılması arama süresini dakikalara çıkarabilir. Arama sonuçlarının 100 ms (100 milisecond bir saniyenin onda biridir) gibi bir zaman diliminde oluşturulması şartı bu projede klasik veri tabanlarının aksine bir tercihi zorunlu kılmıştır. Veri tabanından çekilen verilerle dosya tabanlı yeni bir veri tabanı oluşturulmakta ve bu dosyalar onlarca GB büyüklükte olabilmektedir. Bu veri tabanında bulunan verilere index dosyaları üzerinden erişilmektedir. Örneğin kullanıcı İstanbul Kadıköy lokasyonunu aradığında, arama işlemi önce index dosyasında yapılmaktadır. İndex dosyasında İstabul Kadıköy için bir veri bulunduğunda, bu veri adres nesnesinin dosya veri tabanındaki gerçek adresini (storage id) ihtiva etmektedir. Bu şekilde index üzerinden dosya veri tabanındaki adres nesnesine ulaşmak mümkün olmaktadır. Arama işlemlerinin hızlı yapılabilmesi için bu index ve diğer veri tabanı dosyalarının topluca hazıfaya yüklenmesi gerekmektedir. Aksi taktirde arama işlemleri çok uzun sürebilmektedir, çünkü arama esnasında hafızada yüklü olmayan bir adres nesnesi bulundu ise, bu nesnenin disk üzerinde bulunan veri tabanı dosyalarından yüklenmesi gerekmektedir. Bu gibi IO (Input/Output) işlemleri zaman aldığı için, genel olarak arama işlemi bu gibi durumlarda uzamaktadır. Bunun önüne geçmek için tüm dosyaların hafızaya yüklenmesi gerekmektedir. Bu işlemi yapmak için de Java Memory Mapped Files yapıları kullanılmaktadır.

-bash-3.2$ pmap PID komutunu girdiğimizde 39 GB alanın ne için kullanıldığını görebiliyoruz.

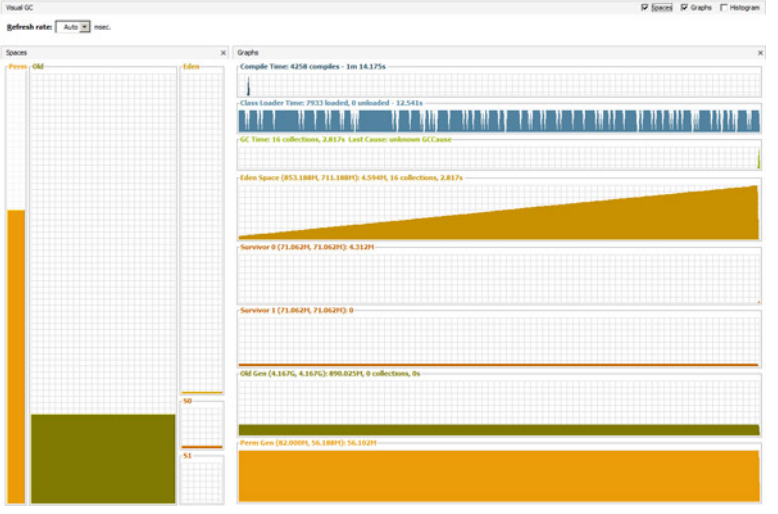


Yukarıda yer alan resimde görüldüğü gibi uygulama bünyesinde kullanılan tüm dosyalar işletim sistemi tarafından oluşturulan işleme (process) dahil edişmiştir. Bu dosyalara Java terminolojisinde Memory Mapped Files (hazıfada yüklü dosyalar) ismi verilmektedir. Yüksek performansın önemli olduğu durumlarda bu dosyaların %100 hafızaya yüklenmiş olmaları büyük önem taşımaktadır. Aksi taktirde dosyaların kullanılmak istendiğinde hafızada olmamaları performansı kötü etkilemektedir. Kullanılan hafıza alanının arkasında bir dosya

yoksa yani bir memory mapped file kullanılmıyorsa, bu alanlar [anon] (anonim) olarak listede gözükmektedir.

Bizim örneğimizde ihtiyaç duyulan tüm dosyaların hepsinin %100 hafızaya yüklenmediğini görüyoruz. İlk resimde yer alan RES (resident – aktüel işgal edilen hazıfa alanı anlamında) kolonuna göre JVM'in aslında 36 GB hafıza alanı işgal etmektedir. İşletim sistemi biz zorlamadıkça kullanılan tüm dosyaları %100 hafızaya yüklemez. Bu şekilde örneğin kullanılmayan fonsiyon kütüphanelerinin hafızada boş yere yer işgal etmesi engellenmiş olur. Bunun yanısıra işletim sistemi her program tarafından kullanılan ortak fonksiyon kütüphanelerini sadece bir kez hazıya yükleyerek, bu dosyaların değişik işlemler (process) tarafından ortaklaşa kullanılmasını sağlar. Birinci resimde SHR (shared – ortaklaşa kullanılan hafıza alanı anlamında) kolonuna baktığımızda bu değerin 12 GB oldugunu görmekteyiz, yani başka programlarla paylaştığımız 12 GB büyüklügünde dosyalar işlemci hafızamıza (Java Process Heap) yüklenmış durumdadır.

Belirttigim gibi bu dosyalar 5 GB'lik Java Heap içinde yer almamaktadırlar. Bu dosyalar daha önce bahsettiğim 39 GB'lik hafaza alanında yer almaktadir. Bu hafıza alanına Java Process Heap, normal Java nesnelerinin yer aldığı alana ise Java Heap ismi verilmektedir. Java Heap ve Java Process Heap bir araya geldiğinde 44 GB'lik, ilk resimde gördüğümüz Java işleminin tüm hafıza alanı ortaya çıkmaktadır. Java Process Heap alanı işletim sisteminden malloc() sistem fonksiyonu kullanılarak oluşturulan hafıza alanıdır. Java'da bu hafıza alanına genel olarak native memory ismi verilir. JNI (Java Native Interface) kullanılarak Java uygulamaları için native memory alanı oluşturmak ve kullanmak mümkündür. Java içinde ise hafıza alanı new operatörünü kullanarak tedarik edilir. Bu hafıza alanı oluşturduğumuz nesne için direk Java Heap bünyesinden gelir ve tamamen JVM tarafından, daha doğrusu Garbage Collector (kullanılmayan nesnelere garbage ismi verilir) tarafından yönetilir.



Yukarda yer alan resime baktığımızda JVM tarafından yönetilen hafıza alanının toplamda 5 GB

olduğunu, bunun 1 GB'lik gibi bir kısmının EDEN Heap space, 150 MB'sinin iki SURVIVOR Heap Space ve 4 GB'sinin OLD Heap space tarafından kullanıldığını görmekteyiz. Toplamda JVM'in yönettiği ve içinde Java nesneleri oluşturabileceğimiz alan 5 GB büyüklüktedir.

Bahsettiğim index ya da veri tabanı dosyalarının normal heap içinde bulunmaları JVM'e fazladan yük getirmekte ve Garbage Collection işlemini uzatmaktadır. Bu tür dosyaları normal Heap içinden Java Process Heap'e (native memory alanına) taşımak için ByteBuffer sınıfı kullanılmaktadır. ByteBuffer.allocateDirect() metodu ile en fazla 2 GB büyüklüğünge, Java Heap dışında hafıza alanı rezerve etmek mümkündür. Bu hafıza alanı Java Process Heap içinde yer alacaktır. allocateDirect() metoduna baktğımızda native hafıza rezervasyonu için sun.misc.Unsafe sınıfının allocateMemory() metodunun kullanıldığını görmekteyiz. Bu native olarak tanımlanmış ve C++ dilinde kodlanmış bir metotdur. Buradan da anlaşıldığı gibi Java bünyesinde bu sınıfı kullanmadan native memory rezervasyonu mümkün değildir.

JVM bünyesinde tüm hafıza otomatik olarak JVM ve Garbage Collector tarafından yönetilir. Programcının bu konuda yapması gereken fazla birşey yoktur. Java Heap dışında işlem yapmak zorunda kalındığında durum farklıdır. Native hafıza alanını JVM bünyesindeki Garbage Collector yönetmez. Programcının C/C++ dillerinde olduğu gibi native hafıza alanını kendisi yönetmesi gerekir. Bu yüzden Morpheus'un verdiği kırmızı hapı yutup, Matrix'in dışına çıkması gerekir. Sadece Matrix'in dışında olan programcılar gerçekte olup bitenlerden haberdardırlar.

Ucuz Etin Yahnisi

http://www.mikrodevre.com/2013/10/21/ucuz-etin-yahnisi/

Ucuz etin yahnisi yavan olur derler. Yiyenler bilir, tat vermez. Ne yediğini anlamassın. Aynı şey kolaya kaçılan işler için de geçerlidir. Elde edilen netice insanı tatmin etmez, doyurmaz, geride akılda ucuşan bir sürü soru bırakır.

Geçenlerde ucuz et yahnisini yeme, ya da yemeyip, pahalı etin bedelini ödeme seçimini yapmak zorunda kaldım. Ucuz et Arduino idi. Evde iki ya da üç adet Arduino var. İlk mikrodenetleyiciler hakkında bir şeyler işitmeye başladığımda, internette yaptığım araştırmalar devamlı karşıma bu ürünü çıkardı. Ben de bu konuda çaylak olduğumdan bir şey zannetip piyasada ne kadar Arduino kitabı varsa topladım, 2-3 adet te Arduino board sipariş ettim.

Arduino ile basit bir LED yakıp, söndüren bir devreyi beş dakika içinde kurabilir ve programlayabilirsiniz. Akabinde olay bumuymuş ya sorusunu kendinize sormaya baslarsınız. Ha güzelmiş deyip, birkaç proje yapmaya çalışırsınız. Çok kısa zamanda duvara toslayıp, temel elektronik bilginiz olmadığı için hiçbir başarı elde edemez ve yavan et yahnisi yemiş gibi olursunuz. Bunlar başımdan geçtiği için yazıyorum. Arduino'yu küçümsemiyorum, nedir, ne değildir konusuna daha sonra değineceğim.

Ben hızlı netice almaktansa, nedenlerini araştırıp, konuyu iyice anlamak isteyen, yani kendi tabirimle öğrenim ve anlama güdümlü olan, netice güdümlü olmayan bir yapıya sahibim. Ufacık bir soruya bulamadığım cevap benim günlerce olduğum yerde çakılıp, santim ilerleyememe sebep olabilir. O sorunun cevabını bulmak ve nedenini anlayabilmek için yarı ömrümü feda edebilirim. Arduino ile yapılacak çalışmaların böyle bir bünye ile barışık olmayacağı ortada. Bu tür çalışmaların beni tatmin etmeyeceğini anladığım için, konuyu baştan sarmak istedim. Baştan dediğim de, mikrodenetleyiciyi alıp, breadboard üzerinde devre kurma değil! Daha da başına sarmam gerekliliğini hissettim, yani temel elektronik hakkında eğitim alma gerekliliğini anladım.

Bu çalışmam benim üç, dört gece sabahlamama sebep oldu. Üç adet ATMega8 mikrodenetleyici yaktım, neden çalışmıyor bu, bana garezi mi var diye kafamı duvarlara vurdum, başarısızlık üzerine başarısızlık yaşadım, devreyi elli kere breadboard üzerinde kurdum, lakin en sonunda devreyi doğru kurmayı ve lambanın yanıp, sönmesini sağlayan programı mikrodenetleyiciye aktarmayı başardım. Alt tarafı bir LED'i yakıp, söndürmüşsün diyecekseniz. Dediğim gibi Arduino ile bu işi beş dakikada yaparsınız. Lakin 5 dakika ile dört gece çekilen çilenin insanı nasıl etkilediğini ve öğrenim sürecini nasıl şekillendirdiğini düşünebiliyor musunuz? Aradaki fark bu. Yani ucuz olmayan etin yahnisini yemek işte böyle bir şey.

Coşkun Taşdemir'in bu yazısında mühendislere Arduino hakkında güzel tavsiyeleri var. Arduino çok güzel bir fikir. Mikrodenetleyiciler dünyasına adım atıp, hızlı netice almak ya da mevcut bir fikri prototiplemek isteyenler için biçilmiş bir kaftan. Hızlı netice almanın bedeli ise mikrodenetleyiciler dünyasına tam anlamıyla vakif olamamak ve akılda bir sürü cevabı bulanamamış soru taşımak. Bu durum mikrodenetleyiciler dünyasına şöyle bir göz atmak isteyenler için sorun teşkil etmiyor. Lakin elektronik mühendisliği okuyan bir öğrenci için büyük

bir problem, çünkü Arduino temelindeki bütün konseptleri maskelemek (sırlarını vermemek) için tasarlanmış adeta. Bir mühendisin Arduino ile bu dünyaya giriş yapıp, Arduino ile devam edip, bu konuya bir zaman sonra vakıf olacağını düşünmesi büyük bir kandırmacadır. Arduino mikrodenetleyici dünyasına yedi kat gökten aralanmış bir kapıdır. Oradan bir şeyler görünüyor gibi olsa da, tam bakıldığında egoların hızlı neticelerle kendilerini avuttukları bir dünya görülür. Ne olup, bittiğini anlamak için işin temeline inmek gerekir. Bu bazen sıfırdan başlamak anlamına gelse de, önemli olan buna katlanabilmektir, çünkü acı çekmeden üstad olunmaz.

Yanlış anlaşılmasın. Ben süperim, üç gece uğraştım, aslanlar gibi devreyi kurdum, benden daha iyisi yok demeye getirmiyorum. Estağfurullah, ne haddimize! Hızlı netice alma ile istikrarlı ama bir o kadar da acılı öğrenim patikasının nasıl farklılık gösterdiğini aktarmaya çalıştım. Herkes neyi nasıl ögreneceğinin seçimini yapmakta serbest :) Öyle olmasaydı zaten Arduino gibi sistemlerin var olmaları mümkün olmazdı.

Ödünç İnternet

http://ozcanacar.com/odunc-internet/

Son zamanlar aşağıdakine benzer birkaç e-posta aldım:

[Bilisim-Internet] Gruplar Kapaniyor ve Sponsor Arayisi

Posteingang x

FARUK farukka@yahoo.com über yahoogroups.com 13:45 (vor 48 Minuten)

an Bilisim-Intern.

Bilder werden nicht angezeigt. Bilder unten anzeigen - Bilder von farukka@yahoo.com immer anzeigen

Gruplar Kapaniyor ve Sponsor Arayisi

Degerli Uyeler,

Oncesinde eGroups.com olan yahoo gruplarda 12 yildir paylasimda bulunuyoruz fakat Yahoo'nun yeni CEO'sunun radikal planlari cercevesinde yaptigi son aciklamalardan anlasildigina gore Yahoo bu hizmeti yakinda kapatacak ve gruplardaki iletisim ve paylasim da duracak.

Zaman zaman alevlenmeler olsa da bu gruplarin cok onemli bir islev gordugu de acik. Fakat kuresel krizin de etkisiyle artik Internet'te bedava donemi geride kaliyor. Hala bedava olan urunler de kullaniciyi o kadar yakindan analiz ediyor ki reklamla ucreti aliyor zaten.

Bu noktada eli tasin altina koymak zamani gelmistir. Ya gruplar tamamen yok olacak ya da burada yakalanan bu sinerjinin devami icin kisiler ve kurumlar birtakim desteklerde bulunacaklar.

Böyle olacağı belliydi. Şimdiye kadar hep ödünç alınmış bir internet kullandık. Facebook'ta yüklenen onca resim, Twitter üzerinden atılan kısa mesajlar, Blogspot'da yazılan bloglar... Bunların hepsi sizin mi zannediyorsunuz? Hepsini ödünc kullanıyoruz. Facebook kafası bozulduğu zaman üyelik hesabını kimseye sormadan kapatabiliyor. Kapattı diyelim. Buna karşı ne yapabilirsiniz? Hiçbir şey! Ödünç verdi, geri aldı!

Yahoo örneğinde olduğu gibi zorda kaldıklarında bu firmaların hepsi tek tek kullanıcılarından para isteyecekler. Bunu doğal olarak karşılayabiliriz. „Para istiyorsa, kullanmam" diyenler olabilir. Bende aynı fikirdeyim. Ama interneti ödünç kullandığımızın farkında mıyız? İnternete girmeden bahsetmiyorum. Kullandığımız servislerden bahsediyorum. Kullandığımız her büyük servisin arkasında bir firma var. Bunların çogu Amerikan firmaları. Kapitalist bir ülkenin bize verdiği ödünç bir interneti kullanıyoruz. İstedikleri zaman kullandığımız servisleri düşünceleri doğrultusunda değiştirebilirler. Genelde bir gurup kullanıcı servislerin paralı olmalarında şikayetçi. Ben gerçek problemin başka bir yerde olduğunu düşünüyorum. Bu firmalar bizim internette özgürlüğümüzü kısıtlama imkanlarına sahip. Yazdıklarımız birilerine battığı zaman özgürce yazmamız kısıtlanabilir. Buna karşı birşey yapmamız da mümkün değil, çünkü ödünç bir internet kullanıyoruz.

Bundan kurtulmanın tek yolu, internete kendi sunucumuzu koymamız ve kendi kişisel servislerimizi oluşturmamızdan geçer. Blog yazacaksak kendi sunucumuza WordPress kurarak bunu yapabiliriz. Bunun gibi daha birçok açık kaynaklı (open source) proje yardımı ile kendi servislerimizi oluşturabiliriz. Bunu herkes yapamayabilir. O zaman dernekler kurarak, kendi internetimize sahip çıkalım. Ödünç internet kullanıcısı olmak yerine, internetin sahibi olalım. Oluşturacağımız dernekler üye aidatı ile gerekli sunucuları ve servisleri internete koyabilir. Kimseye boynumuz eğik olmadan bu servisleri kullanabiliriz. Bu konuda biraz düşünelim derim.

Java Hotspot, Assembler Kod, Hafıza Bariyerleri ve Volatile Analizi

http://www.kurumsaljava.com/2013/11/08/java-hotspot-assembler-kodu-ve-volatile-analizi/

Java kodu Java derleyicisi javac (compiler) tarafından byte koduna dönüştürülür. Bu makina kodu Assembler değildir. Bu yüzden Java byte kodunu mikroişlemci koşturamaz. Java kodunu koşturabilmek için mikroişlemci ile Java byte kodu arasında, byte kodunu mikroişlemci koduna dönüştürebilecek bir ara katmana daha ihtiyaç duyulmaktadır. Bu JVM (Java Virtual Machine) ismini taşıyan sanal makinadır. Sanal makina Java byte kodunu mikroişlemi gibi koşturur. Java byte kodu için mikroişlemci sanal makinadır. JVM bünyesinde Java byte kodu makina koduna dönüştürülerek mikroişlemci üzerinde koşturulur. Bu yazımda Assembler makina koduna dönüştürülmüş Java kodundan örnek sunmak istiyorum.

JVM bünyesinde byte kodu doğrudan Assembler makina koduna çeviren birim Hotspot JIT (Just In Time) derleyicisidir. JIT olmadan JVM sadece Java byte kodu yorumlayıcısıdır (interpreter). Hotspot çok sık kullanılan kod bölümlerini, mikroişlemci üzerinde daha hızlı koşturulabilmeleri için doğrudan Assembler koduna dönüştürür. Bu işlemi yapabilmesi için belli bir süre kodu analiz etmesi gerekmektedir. Bu sebepten dolayı Java uygulamaları belli bir ısınma aşamasından sonra daha hızlı çalışmaya başlarlar, çünkü byte kodun belli bir kısmı ya da hepsi JIT tarafından Assembler makina koduna dönüştürülmüştür.

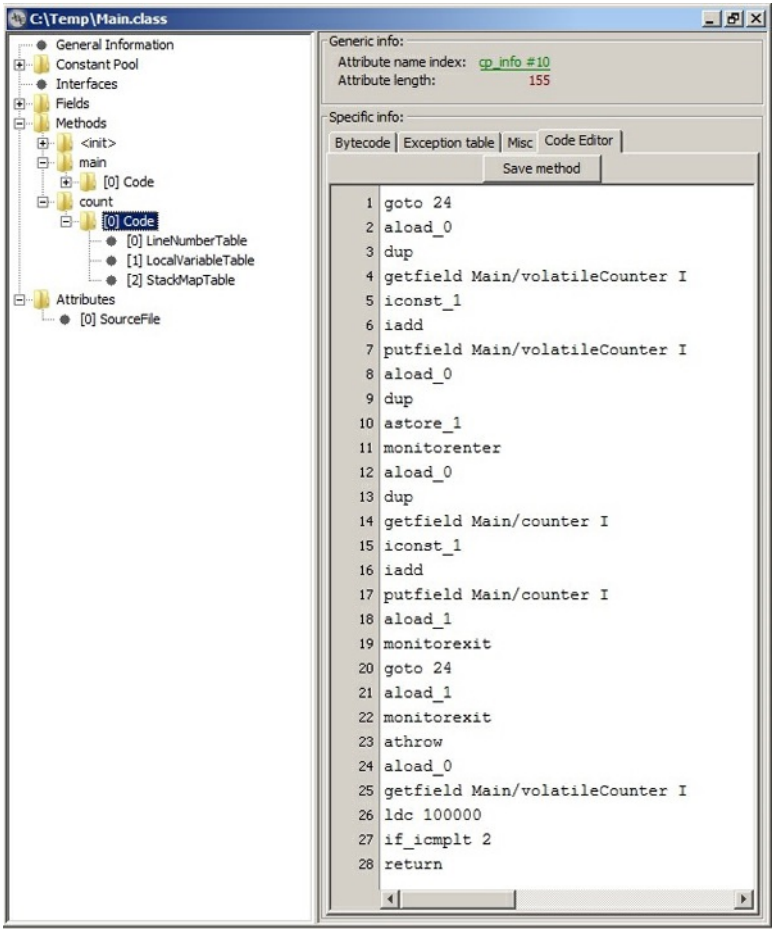
JIT tarafından oluşturulan makina kodunun çıktısını alabilmek için bir Hotspot Disassembler plugin kullanmamız gerekmektedir. Kenai base-hsdis projesi bünyesinde böyle bir plugin yer almaktaktadır. Ben bu yazımdaki örnekler için Linux altında 64 bit JDK7 (jdk1.7.0*45) kullandım. Kullandığım disassembler plugin linux-hsdis-amd64.so ismini taşıyor. Bu dosyanın libhsdis-amd64.so ismini taşıyacak şekilde jdk1.7.0*45/jre/lib/amd64/server dizinine kopyalanması gerekiyor. Bu işlem ardından çalışan bir Java uygulamasının makina kodu çıktısını alabiliriz.

Önce koşturmak istediğimiz Java koduna bir göz atalım. Main sınıfı bünyesinde volatile olan volatileCounter ve counter değişkenleri yer almaktadır. count() metodu bünyesinde bir for döngüsünde bu değişkenlerin değerleri artırılmaktadır. For döngüsü volatileCounter değişkeni 100000 değerine ulaştığında son bulmaktadır. Hotspot JIT kodu 100000 sefer koşturulduğundan dolayı makina koduna dönüştürmektedir. Bu uygulamanın çok sıkca kullanılan alanlarının (hotspot; sıcak alan anlamında) makina koduna dönüştürülerek, daha da hızlı koşturulabilmeleri için gerekli bir işlemdir. Hotspot sadece sıkca koşturulan kodları makina koduna dönüştürür. Sıkça kullanılmayan kod bloklarının JVM tarafından yorumlanma hızı yeterlidir. Bu tür kodlar için makina kodunun oluşturulması çok maliyetli bir işlemdir. Elde edilecek kazanç çok az olacağından, sıkça kullanılmayan kod blokları makina koduna dönüştürülmez.

Eğer count() metodunda bir for döngüsü kullanılmasaydı, JIT tarafından daha sonra göreceğimiz Assembler kodu oluşturulmazdı. Bunu sağlayan döngünün 100000 adet olmasıdır. JIT hangi kod bloğunun ne kadar koşturulduğunu takip ettiği için hangi kod birimi için makina kodu oluşturacağına karar verebilmektedir.

```
public class Main {

    private volatile int volatileCounter;
    private int counter;

    public static void main(final String[] args) {

        new Main().count();
    }

    private void count() {

        for (; this.volatileCounter < 100000;) {
            this.volatileCounter++;

            synchronized (this) {
                this.counter++;
            }
        }
    }
}
```

Java derleyicisi (javac) tarafından oluşturulan byte kodunu aşağıdaki resimde görmekteyiz.



Şimdi gelelim Hotspot JIT tarafından oluşturulan makina koduna. Makina kodunu görebilmek için JVM'i aşağıdaki şekilde çalıştırmamız gerekiyor:

java -cp . -XX:+UnlockDiagnosticVMOptions -XX:+PrintAssembly Main

Aşağıda oluşan Assembler makina kodu yer almaktadır:

```
1   0x00007f206906015c: jne    0x00007f2069060202  ;*monitorenter
                                                   ; - Main::count@16 (line 16)
2   0x00007f2069060162: incl   0x10(%r13)
3   0x00007f2069060166: mov    $0x7,%r10d
4   0x00007f206906016c: and    0x0(%r13),%r10
5   0x00007f2069060170: cmp    $0x5,%r10
6   0x00007f2069060174: jne    0x00007f2069060226  ;*monitorexit
                                                   ; - Main::count@28 (line 16)
7   0x00007f206906017a: mov    0xc(%r13),%r11d    ; OopMap{rbp=NarrowOop r13=Oop r14=Oop o
                                                   ;*if_icmplt
                                                   ; - Main::count@41 (line 13)
8   0x00007f206906017e: test   %eax,0xa91ee7c(%rip)        # 0x00007f207397f000
                                                   ;   {poll}
9   0x00007f2069060184: cmp    $0x186a0,%r11d
10  0x00007f206906018b: jge    0x00007f20690602ef  ;*aload_0
                                                   ; - Main::count@3 (line 14)
11  0x00007f2069060191: mov    0xc(%r13),%r11d
12  0x00007f2069060195: inc    %r11d
13  0x00007f2069060198: mov    %r11d,0xc(%r13)
14  0x00007f206906019c: lock addl $0x0,(%rsp)     ;*putfield volatileCounter
                                                   ; - Main::count@10 (line 14)
15  0x00007f20690601a1: mov    0x0(%r13),%rax
16  0x00007f20690601a5: mov    %rax,%r10
17  0x00007f20690601a8: and    $0x7,%r10
18  0x00007f20690601ac: cmp    $0x5,%r10
19  0x00007f20690601b0: jne    0x00007f2069060106
20  0x00007f20690601b6: mov    0xb0(%r12,%rbp,8),%r10
21  0x00007f20690601be: mov    %r10,%r11
22  0x00007f20690601c1: or     %r15,%r11
23  0x00007f20690601c4: mov    %r11,%r8
24  0x00007f20690601c7: xor    %rax,%r8
25  0x00007f20690601ca: test   $0xffffffffffffff87,%r8
26  0x00007f20690601d1: je     0x00007f2069060162
27  0x00007f20690601d3: test   $0x7,%r8
28  0x00007f20690601da: jne    0x00007f2069060100
29  0x00007f20690601e0: test   $0x300,%r8
30  0x00007f20690601e7: jne    0x00007f20690601f6
31  0x00007f20690601e9: and    $0x37f,%rax
32  0x00007f20690601f0: mov    %rax,%r11
33  0x00007f20690601f3: or     %r15,%r11
34  0x00007f20690601f6: lock cmpxchg %r11,0x0(%r13)
35  0x00007f20690601fc: je     0x00007f2069060162
36  0x00007f2069060202: mov    %r14,0x8(%rsp)
37  0x00007f2069060207: mov    %r13,(%rsp)
38  0x00007f206906020b: mov    %r14,%rsi
39  0x00007f206906020e: lea    0x10(%rsp),%rdx
40  0x00007f2069060213: callq  0x00007f206905e320  ; OopMap{rbp=NarrowOop [0]=Oop [8]=Oop o
                                                   ;*monitorenter
                                                   ; - Main::count@16 (line 16)
                                                   ;   {runtime_call}
```

Java kodu sequentially consistent değildir, yani Java byte kodu sahip olduğu sıraya göre satır satır koşturulmaz. Hem Java derleyicisi hem de mikroişlemci performansı artırmak için birbirine bağımlı olmayan kod satırlarının yerlerini değiştirerek işlem yapabilirler. Bu aslında oluşan Java

kodunun programcının yazdığı şekilde koşturulmadığı anlamına gelmektedir. Paralel çalışan programlarda bunun çok ilginç bir yan etkisi var: bir threadin bir değişken üzerinde yaptığı değişikliği başka bir çekirdek üzerinde koşan başka bir thread göremeyebilir. Örneğin t1 (thread 1) a isimli değişkenin değerini bir artırdı ise ve t2 a belli bir değere sahip iken bir for döngüsünü terk etmek istiyorsa, t2 belki bu for döngüsünden hiçbir zaman çıkamayabilir, çünkü a üzerinde t1 tarafından yapılan değişiklikleri göremeyebilir. Bunun sebebi t1 in üzerinde çalıştığı çekirdeğin (core) a üzerinde yaptığı değişiklikleri kendi ön belleginde (L1/L2 cache) tutmasıdır. Mikroişlemciler yüksek performansta çalışabilmek için hazıfa alanları üzerinde yaptıkları değişiklikleri ilk etapta kendi ön belleklerinde tutarlar. Gerek duymadıkça da bu değişiklikleri diğer çekirdeklerle paylaşmazlar. Her çekirdek sahip olduğu önbelliği tüm hafiza alanıymış (RAM) gibi gördüğü için kendi performansını artırmak adına kod sırasını değiştirebilir. Bu belli bir sıraya bağımlı olan diğer threadlerin düşünüldükleri şekilde çalışmalarını engelleyebilir. Bu genelde paralel programlarda program hatası olarak dışarıya yansır.

Bu tür sorunları aşmanın bir yolu volatile tipinde değişkenler kullanmaktır. Volatile tipinde olan değişkenler üzerinde işlem yapıldığında mikroişlemci kodu programcının yazdığı sırada koşturmaya zorlanır. Bunu gerçekleştirmek için hafıza bariyerleri (memory barrier) kullanılır. Mikroişlemci bir hazıfa bariyeri ile karşılaştığında bir çekirdeğin sahip olduğu önbellekteki değişiklikleri doğrudan hafızaya geri yazar (write back) ve bu hazıfa alanını (cache line) kendi önbelleklerinde tutan diğer çekirdeklere mesaj göndererek bu hafıza alanını silmelerini (cache invalidate) talep eder. Böylece herhangi bir çekirdek üzerinde koşan bir threadin yaptığı değişiklik hemen hafızaya, oradan da diğer çekirdeklerin önbelleklerine yansır. Bu t1 tarafından a üzerinde yapılan bir değişikliğin t2 tarafından anında görülmesi anlamına gelmektedir. Bunun gerçekleşmesi için a isimli değişkenin volatile olması gerekmektedir.

Main sınıfında yer alan this.volatileCounter++; satırı ile bahsettiğim hafıza bariyerinin kullanımı gerekmektedir. JVM bunu sağlamak için makina kodunun 14. satırında mikroişlemci için lock addl komutunu kullanmaktadır. 13. satırda yer alan mov komutuyla %r11d registerinde yer alan volatileCounter isimli değişkenin değeri doğrudan hafızaya (RAM) aktarılır. Hafiza alanının adresi %r13 registerinde yer almaktadır. 14. satırda yer alan lock addl ile tüm mikroişlemci bünyesinde global bir hafıza transaksiyonu gerçekleştirilir. Atomik olan bu işlem ile tüm çekirdeklerin üzerinde değişiklik yapılan hafiza alanını önbelleklerinden silmeleri ve yeniden yüklemelerini sağlanır. Böylece diğer threadler yapılan değişiklikleri anında görmüş olurlar.

Java kodunu sequentially consistent yapmanın diğer bir yolu synchronized kelimesinin kullanımıdır. Synchronized kullanılması durumunda mikroişlemci değişikliğe uğrayan değeri önbellekten alıp hafızaya geri aktararak, diğer çekirdeklerin kendi önbelleklerini tazelemelerini sağlar.

Main sınıfında yer alan counter isimli değişken volatile olmadığı için üzerinde yapılan değişiklikler diğer çekirdeklere yansımaz. Bunu sağlamak için count() metodunda değer atamasını synchronized bloğunda yaptım. Bu JVM tarafından tekrar bir hafıza bariyeri kullanımı gerektiren bir işlemdir. JIT makina kodunun 34. satırında lock cmpxchg ile gerekli hafıza bariyerini oluşturmaktadır. Bu üzerinde işlem yapılan çekirdeğin counter isimli değişkenin değerini tekrar hafızaya geri aktarmasını ve diğer çekirdeklerin bu değeri tekrar

hafızadan kendi önbelleklerine çekmelerini sağlamaktadır.

Çöplerin Efendisi

http://www.kurumsaljava.com/2012/05/17/coplerin-efendisi-garbage-collection/

Java programcısının çok sadık bir hizmetçisi var. Her türlü çöplüğü, pisliği arkasından devamlı toplar, hiç sesini çıkarmaz. Çöplerin efendisidir, ama bir o kadar da mütevazidir. Kimseye belli etmeden işini görür. Bu yüzden birçok Java programcısı onun farkında bile değildir. Ama o işini yapmasa Java programcısının hali çok vahim olur, C/C++ ile kod yazan meslektaşlarından bir farkı kalmaz, bilgisayarın hafızası denilen kara delikte kaybolur gider, yazdığı programlar devamlı sallanır.

Garbage Collector'dan bahsediyorum. Çoğu Java programcısının varlığından bile haberdar olmadığı, bazılarının çekindiği, bazılarının ne yaptığını tam olarak anlamdığı, bazılarının ise yaptığı işe hayran olup, ince ince hayranlık duyduğu Garbage Collector... Java'yı Java yapan özelliklerinden bir tanesi bir Garbage Collector'e sahip olmasıdır. Şimdi yakından tanıyalım bu kahramanı.

Java dilinde herhangi bir sınıftan bir nesne oluşturmak için new operatörü, Class.forName().newInstance(), clone() veya readObject() metodu kullanılır. Örneğin new operatörü sınıfın konstrüktörünü kullanarak JVM bünyesinde yeni bir nesne oluşturur. Nesne inşa edildikten sonra Java Heap içinde konuşlandırılır. Nesnelerin hepsi kullanıldıkları sürece heap icinde mutlu, mesut hayatlarını sürdürürler. Ama her canlı gibi onlar da birgün ölürler. Öldüklerinde onları toplayıp, ebediyete intikal ettiren Garbage Collector'dür.

İngilizce'de garbage çöp anlamına gelmektedir. Garbage Collector JVM bünyesinde hem yeni nesnelerin doğmasına yardımcı olan, hem de ölen nesneleri ortadan kaldıran modüldür. new operatörü ile doğan yeni nesneler için Garbage Collector hafıza alanını temin eder. Ölen ve çöp olarak isimlendirilen nesnelerin işgal ettikleri hafıza alanını Garbage Collector boşaltır ve yeni nesnelere tayin eder. Garbage Collector kısaca Java'da otomatik hafıza yönetiminden sorumludur. Onsuz Java'nın C/C++ dilinden bir farkı kalmazdı.

Garbage Collector'ün nasıl çalıştığını görebilmek için aşağıda yer alan videoyu hazırladım. Collector'ü kızdırmak için bolca çöp (garbage) üreten minik bir Java programını şu şekilde oluşturmak yeterli:

```
package com.kurumsaljava.jvm.garbage;

import java.util.ArrayList;
import java.util.List;

public class Heap {


    public static void main(String[] args) throws Exception {
        List<String> temp = new ArrayList<String>();

        Thread.sleep(10000);
        while (true) {

            temp.add("xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
                              xxxxxxxxxxxxxxxxxxxxxxxxxx");
            //System.out.println(temp.size());

            if(temp.size() > 3000000) {
                temp = null;
                temp = new ArrayList<String>();

            }
        }
    }
}
```

Heap ismini taşıyan sınıf bünyesinde x'lerden oluşan uzunca bir String nesnesini sonsuz bir döngü içinde temp ismindeki bir listeye yerleştiriyoruz. Liste bünyesinde yer alan String nesne adedi 300.000'e ulaştığında listeye null değerini atayarak siliyor ve yeniden bir liste oluşturuyoruz. Döngü içinde tekrar String nesneleri yeni listeye ekleniyor. Buradaki amacım hafızanın tükenmesi ile java.lang.OutOfMemoryError oluşmasını engellemek ve Garbage Collector'ün değişik heap alanları üzerinde nasıl çalıştığını gösterebilmekti. Java'da herhangi bir nesneye null degeri atandığında Garbage Collector'e "bu nesne ölmüştür, yok edilebilir" mesajı verilmektedir. Eğer bu nesneye herhangi başka bir nesneden referans yoksa, Garbage Collector bu nesneyi öldü kabul edip, temizleyecektir. Şimdi beraber videoyu izleyelim.

> Youtube video

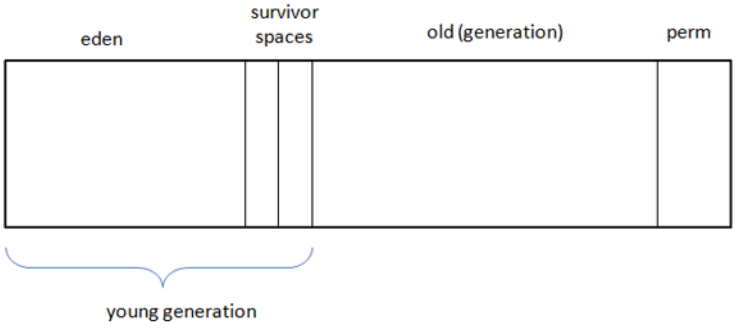
Videoda yer alan program için 64 MB büyüklüğünde bir heap alanını -Xms64m -Xmx64m JVM parametreleri kullanarak tahsis ettim. Eğer heap alanı 10 MB olursa OutOfMemoryError hatası oluşmaktadır. Aynı şekilde liste içindeki String nesne adedini 600.000'e çıkardığımda hafıza yetersizliğinden dolayı OutOfMemoryError hatası oluşmaktadır.

Video ve oluşan Garbage Collection aktiviteleri hakkında detaya girmeden önce, değişik Garbage Collector mekanizmalarını ve algoritmalarını gözden geçirelim.

Generational Garbage Collection

Tipik bir Java uygulaması gözlemlendiğinde, uygulamanın hayatı süresince oluşan birçok nesnenin kısa ömürlü oldukları görülmektedir. Oluşturulan nesnelerin çogu bir metot son

bulduktan sonra, sadece metot bünyesinde kullanıldıkları için hayata gözlerini yumarlar. Bunun yanısra bir Java uygulaması bünyesinde nesneler yaşlanarak belli bir yaşa erişebilirler. Bir nesnenin yaşlanma süreci uygulama bünyesinde kullanılma derecesini yansıtmaktadır.



Resim 1

Java uygulamarında oluşturulan nesnelerin hayat döngülerini yönetmek icin Sun firması tarafından JDK 1.3 versiyonu ile Generational Garbage Collection geliştirilmiştir. Resim 1'de görüldüğü gibi Generation Garbage Collection ile Java uygulamasının kullandığı hazıfa iki jenerasyona bölünmektedir. Yeni doğan nesneler genç jenerasyon (young generation), orta ve ileri yaşlı nesneler yaşlı jenerasyon (old generation) bölümünde konuşlandırılırlar.

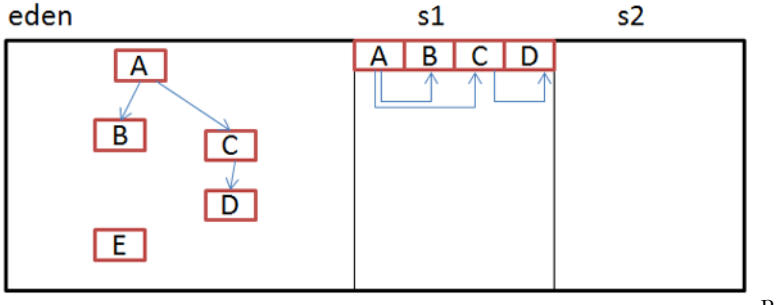
Nesnelerin ait oldukları yaş gurubuna gore ihtiyaç duyulan hafıza yönetim mekanizmaları farklılık gösterir. Yeni jenerasyon bünyesinde nesne doğum ve ölümleri çok hızlı gerçekleşir. Bu bölümde kullanılan Garbage Collection algoritmasının yeni doğan nesnelere hızlı bir şekilde hazıfa alanı tahsis etmesi, ölen nesneleri ortadan kaldırması gerekmektedir. Bu jenerasyon için kullanılan Garbage Collection algoritması bu ihtiyaca cevap verecek şekilde geliştirilmiştir. Yaşlı jenerasyonda durum farklıdır. Bu bölümde Formel 1'de olduğu gibi hızlı haraket etme gerekliliği yoktur. Bu jenerasyonda nesneler daha yavaş bir hayat temposuna sahiptir. Bu jenerasyonda kullanılan Garbage Collection algoritması buna uygun şekilde geliştirilmistir.

Young Generation Garbage Collection

Young generation üç hafıza alanından oluşmaktadır. Bunlar Eden, Survivor 1 ve Survivor 2 hafıza alanlarıdır. Bir Java uygulamasında nesnelerin hepsi gözlerini Eden hafıza alanında dünyaya açarlar. Survivor 1 ve Survivor 2 alanları geçici ara hafıza alanı olarak kullanılır. Eden hafıza alanı dolduğunda, Garbage Collector hayatta kalan nesneleri önce boş olan Survivor alanlarından birisine kopyalar. Bu işlemin ardından Garbage Collector Eden ve kullanımda olan Survivor alanını boşaltır. Eden hafıza alanı tekrar dolduğunda, hayatta kalan nesneler ve dolu olan Survivor alanındaki nesnelerin hepsi tekrar boş olan Survivor alanına kopyalanır. Her zaman bir Survivor alanı boştur ve bir sonraki Garbage Collection işlemi sonunda hayatta kalan nesneleri bünyesine alacak şekilde pasif olarak bekler. Videoda da Eden ve Survivor hafıza alanlarının Garbage Collector tarafından doldurulma ve boşaltılma işlemleri görülmektedir. Young generation bünyesinde çalışan Garbage Collection mark & copy (işaretle ve kopyala) algoritmasını kullanmaktadır. Şimdi bu algoritmanın nasıl çalıştığını yakından inceleyelim.

Mark & Copy Algoritması

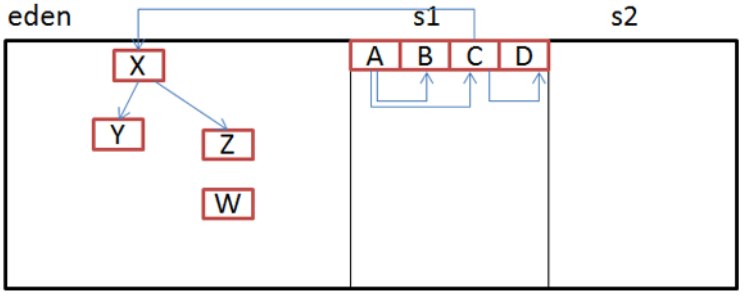
JVM bünyesinde kullanılan tüm Garbage Collection mekanizmaları hayattaki nesneleri işaretleme (mark) işlemi ile işe başlarlar. Garbage Collector hangi nesnelerin hayatta olduğunu anlamak için kök setini (root set) oluşturan nesnelerden yola çıkıp, diğer nesnelere olan bağlantıları takip eder. Bu işlemi bir ağacın kökünden yola çıkarak, dallarına kadar erişme olarak düşünebiliriz. Garbage Collector bu şekilde hayatta olan tüm nesneleri işaretler. Erişemediği diğer nesnelerin hepsini ölü kabul eder ve bir sonraki Garbage Collection ile ortadan kaldırılmalarını sağlar. Kök setinde kendisine doğru nesne referansı olmayan, ama kendisinden başka nesnelere referans giden nesneler yer alır. Bu tip nesneler genelde metot parametreleridir ya da global geçerliliği olan statik olarak tanımlanmış nesnelerdir.



Resim 2

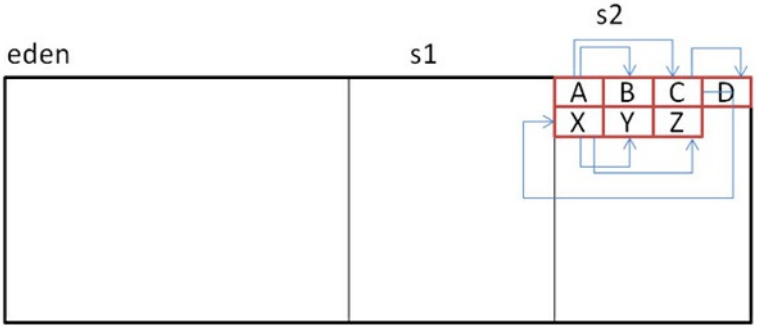
Resim 2'de kök setin A ve E nesneslerinden oluştuğu görülmektedir. Garbage Collector işaretleme işine önce A nesnesinden yola çıkarak başlar. B, C ve dolaylı olarak D nesnelerine erişir ve bu nesne kümesini (A,B,C,D) hayatta olan nesneler olarak işaretler. İşaretleme işine E nesnesi ile devam eder. E nesnesi kök setinde yer alan bir nesnedir, lakin E'den yola çıkarak başka bir nesneye erişmek mümkün değildir. Ayrıca başka bir nesne de E nesnesine işaret etmemektedir. Bu yüzden E nesnesi artık ölmüş bir nesnedir ve bir sonraki Garbage Collection ile E nesnesinin işgal ettiği hafıza alanı temizlenir.

Garbage Collector işaretleme (mark) işlemini tamamladıktan sonra hayatta kalan nesneleri aktif olan Survivor alanına kopyalama işlemine başlar. Garbage Collector resim 2'de görüldüğü gibi hayatta kalan nesne kümesini (ABCD) aktif olan Survivor 1 alanına kopyalar. Nesneler, işgal ettikleri adres alanları arka arkaya gelecek şekilde Survivor 1'de konuşlandırılır. Garbage Collection işlemi sonunda hayatta kalan nesnelerin birbirlerine olan referansları ve sahip oldukları adres alanları değişeceği için Garbage Collector işaretleme ve kopyalama işlemi öncesinde uygulama bünyesinde çalışan tüm threadleri durdurmak zorundadır. Bu işlem JVM terminolojisinde dünyayı durdurma (stop the world) olarak adlandırılır. Belli bir süre boyunca uygulama durur ve uygulama bünyesinde aktif olan sadece Garbage Collector'dür. Eğer Garbage Collector işine başlamadan önce dünyayı durdurmamış olsa, aktif olan uygulama threadleri nesne referansları üzerinde değişiklik yaparak, Garbage Collector'ün yanlış nesneler üzerinde işlem yapmasına sebep olabilirler. Bu sebepten dolayı Garbage Collection öncesi dünyanın durması gerekmektedir.



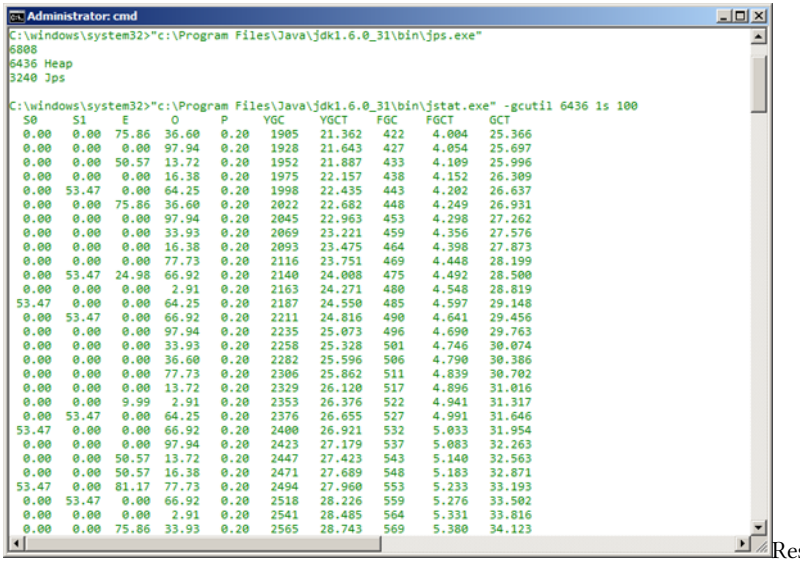
Resim 3

Resim 3'de Garbage Collection sonrasınra Eden bünyesinde yeni nesnelerin oluştuğu görülmektedir. Kök set içinde X ve W nesnesi yer almaktadır. Garbage Collector X,Y,Z ve Survivor 1'de yer alan ABCD nesnelerini Survivor 2'ye kopyalar. Garbage Collection işlemi sonunda young generation hafıza alanındaki son durum resim 4'de yer almaktadır.



Resim 4

Görüldüğü gibi her Garbage Collection sonunda Survivor alanları rolleri değiştirmektedir. Survivor alanlarının her Garbage Collection sonrası yer degiştirmeleri resim 5'de de görülmektedir.



Resim 5

Resim 5'de yer alan birinci kolon Survivor 1'in, ikinci kolon Survivor 2'nin, üçüncü kolon Eden'in, dördüncü kolon Old hafıza alanının doluluk oranını göstermektedir. Görüldüğü gibi Garbage Collector düzenli olarak hayatta kalan nesneleri kopyalamak için pasif olan Survivor alanını kullanmaktadır.

```
{Heap before GC invocations=12099 (full 2688):
 def new generation   total 19648K, used 14182K [0x2fb10000, 0x31060000, 0x31060000)
  eden space 17472K,  81% used [0x2fb10000, 0x308e9b50, 0x30c20000)
  from space 2176K,   0% used [0x30e40000, 0x30e40000, 0x31060000)
  to   space 2176K,   0% used [0x30c20000, 0x30c20000, 0x30e40000)
 tenured generation   total 43712K, used 1270K [0x31060000, 0x33b10000, 0x33b10000)
   the space 43712K,   2% used [0x31060000, 0x3119d8e8, 0x3119da00, 0x33b10000)
 compacting perm gen  total 12288K, used 24K [0x33b10000, 0x34710000, 0x37b10000)
   the space 12288K,   0% used [0x33b10000, 0x33b16308, 0x33b16400, 0x34710000)
    ro space 10240K,  51% used [0x37b10000, 0x3803e318, 0x3803e400, 0x38510000)
    rw space 12288K,  55% used [0x38510000, 0x38baa088, 0x38baa200, 0x39110000)
[GC 15453K->7160K(63360K), 0.0089426 secs]
Heap after GC invocations=12100 (full 2688):
 def new generation   total 19648K, used 0K [0x2fb10000, 0x31060000, 0x31060000)
  eden space 17472K,   0% used [0x2fb10000, 0x2fb10000, 0x30c20000)
  from space 2176K,   0% used [0x30c20000, 0x30c20000, 0x30e40000)
  to   space 2176K,   0% used [0x30e40000, 0x30e40000, 0x31060000)
 tenured generation   total 43712K, used 7160K [0x31060000, 0x33b10000, 0x33b10000)
   the space 43712K,  16% used [0x31060000, 0x3175e2e0, 0x3175e400, 0x33b10000)
 compacting perm gen  total 12288K, used 24K [0x33b10000, 0x34710000, 0x37b10000)
   the space 12288K,   0% used [0x33b10000, 0x33b16308, 0x33b16400, 0x34710000)
    ro space 10240K,  51% used [0x37b10000, 0x3803e318, 0x3803e400, 0x38510000)
    rw space 12288K,  55% used [0x38510000, 0x38baa088, 0x38baa200, 0x39110000)
}
```
Resim 6

Bir Java uygulaması -XX:+PrintHeapAtGC JVM parametresi ile çalıştırıldığv zaman, resim 6'da yer alan Garbage Collection log kayıtları oluşmaktadır. İlk kırmızı işaretli alan içinde Survivor 1 from, Survivor 2 ise to olarak isimlendirilmiştir. Survivor 1'in hafıza başlangıç adresi 0x30e40000, Survivor 2'nin hafıza başlangıç adresi 0x30c20000'dir. İkinci kırmızı işaretli alana baktığımızda from ve to'nun adres alanlarının değiştiğini görmekteyiz. Garbage Collection

işlemi oluşmuş ve Survivor alanları yer degiştirmiştir. Garbage Collector tarafından hep bir Survivor alanı reserve durumunda bir sonraki Garbage Collection içinde kullanılmak üzere beklemede tutulur. Bu dogal olarak belirli bir hafıza alanının kullanılmayarak israf edildiği anlamına gelir. Ne yazık ki bu Mark & Copy algoritmasının bir dejavantajıdır. Bunun haricinde bu algoritma hızlı bir şekilde Eden bünyesinde olup bitenlere cevap verebilecek yetenekte ve kapasitededir.

Minor & Major Garbage Collection

Garbage Collection minor (küçük) ve major (büyük) olmak üzere ikiye ayrılır. Minor Garbage Collection'ı kendi odamızı, Major Garbage Collection'ı evimizi temizleme gibi düşünebiliriz. Young Generation bünyesinde yapılan Garbage Collection işlemine minor Garbage Collection, JVM kontrolündeki tüm hafıza alanlarının temizlenme işlemine major Garbage Collection ismi verilir.

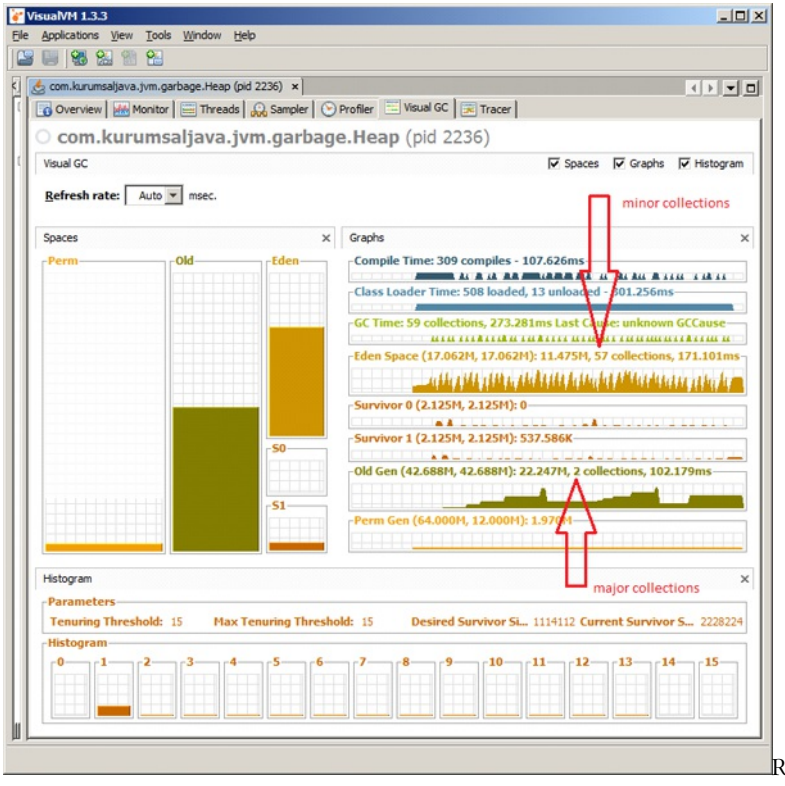
```
[GC 11994K->4048K(63360K), 0.0214741 secs]
[GC 19303K->12883K(63360K), 0.0344847 secs]
[GC 29655K->14047K(63360K), 0.0033241 secs]
[GC 28305K->18774K(63360K), 0.0156822 secs]
[GC 27609K->27609K(63360K), 0.0225508 secs]
[GC 44365K->42027K(63360K), 0.0363610 secs]
[Full GC 56241K->6011K(63360K), 0.0204354 secs]
[GC 14847K->14847K(63360K), 0.0238127 secs]
[GC 31595K->29264K(63360K), 0.0309380 secs]
[GC 43459K->33991K(63360K), 0.0084803 secs]
[GC 42827K->42827K(63360K), 0.0135701 secs]
[Full GC 59573K->1284K(63360K), 0.0063403 secs]
[GC 15471K->7175K(63360K), 0.0089063 secs]
[GC 16010K->16010K(63360K), 0.0132284 secs]
[GC 32755K->30427K(63360K), 0.0213428 secs]
[Full GC 44611K->6011K(63360K), 0.0123965 secs]
[GC 14847K->14847K(63360K), 0.0127060 secs]
[GC 31592K->29264K(63360K), 0.0222304 secs]
[GC 43447K->33991K(63360K), 0.0089595 secs]
[GC 42827K->42827K(63360K), 0.0132295 secs]
[Full GC 59571K->1284K(63360K), 0.0063264 secs]
[GC 15467K->7175K(63360K), 0.0086808 secs]
[GC 16010K->16010K(63360K), 0.0133747 secs]
[GC 32755K->30427K(63360K), 0.0209403 secs]
[Full GC 44610K->6011K(63360K), 0.0126356 secs]
```

Resim 7

Resim 7'de Garbage Collector tarafından yapılan minor ve major Garbage Collection işlemleri yer almaktadır. GC ile başlayan satırlar minor, Full GC ile başlayan satırlar major Garbage Collection yapıldığının ibaresidir. Major Garbage Collection işlemi için ayrıca Full Garbage Collection ismi de kullanılmaktadır.

İlk satırda young generation tarafından işgal edilen hafıza alanı toplamda 11994 kilobytetır (~ 12 MB). Minor Garbage Collection işlemi sonunda ölen nesneler temizlenmiş ve hayatta kalan nesnelerin kapladığı alan 4048 (~ 4 MB) kilobyte olarak değişmiştir. Garbage Collector minor Garbage Collection işlemi ile 8 MB civarında ölü nesneyi hafıza alanından kaldırmıştır. Bu minor Garbage Collection işlemi 0.02 saniye sürmüştür. Bu zaman diliminde JVM bünyesinde sadece Garbage Collection işlemi yapılmıştır. Parantez içinde yer alan 63360K (~ 64 MB) JVM tarafından kullanılan tüm hafıza alanlarının toplamını yansıtmaktadır. JVM parametreleri olarak -Xms64m -Xmx64m kullandığımız için, programın işgal ettiği toplam hafıza alanı 64 MB'dir.

Eden hafıza alanı, Old hafıza alanına göre daha küçük olduğu için bu alanda minor Garbage Collection işlemleri Old generation bünyesinde olan major Garbage Collection işlemlerinden sayıca çok daha fazladır. Resim 7'de Garbage Collector 20 minor Garbage Collection, buna karşılık sadece 4 major Garbage Collection gerçekleştirmiştir.



Resim 8

Yedinci satırda major Garbage Collection yapıldığını görmekteyiz. Tüm JVM bünyesinde 56241 kilobyte (~56 MB) hafıza alanı kullanımdadır (young + s0 + s1 + old). Görüldüğü gibi bu değer JVM hafıza büyüklüğü olan 64 MB'ye yakın bir değerdir. Resim 8'de young ve old generation hafıza alanlarının dağılım oranları yer almaktadır. JVM Eden için 17 MB, S0 için 2 MB, S1 için 2 MB ve Old icin 42 MB alan ayırmıştır. JVM parametreleri ile hafıza alanlarının büyüklüklerini değiştirmek mümkündür. Buna yönelik bir ayarlama yapılmadığında JVM otomatik olarak hafıza alanlarının büyüklüğünü ayarlar. Full GC ardından hayatta kalan nesnelerin kapladığı alan 6011 kilobyte (~6MB) olarak değişmiştir. Bu major garbage collection işlemi ile 50 MB büyüklüğünde nesneler Garbage Collector tarafından ebediyete uğurlanmıştır.

Durdurun Dünyayı

"Durdurun dünyayı başım dönüyor" diyor Ferdi Tayfur şarkısında. JVM mucitleri Ferdi Tayfur'un bu şarkısından esinlenerek dünyayı durduran Garbage Collector algoritmaları geliştirmişlerdir. Arabesk müziğin Java üzerindeki etkilerini başka bir yazımda detaylı olarak ele

alacağım. Arabeskin programcılık üzerinde etkileri düşünüldüğünden çok daha fazladır. (:)

Garbage Collector minor Garbage Collection için tek bir thread (Reaper Thread) kullanmaktadır. Garbage Collection işlemi esnasında Garbage Collector JVM bünyesinde çalışan tüm threadleri durdurur. Uygulama için dünya durmuş olur. Dünya ne kadar uzun dururs, uygulamanın kullanıcıya verdiği cevaplar o oranda gecikir. Buradan Garbage Collection işlemi esnasında verilen araların mümkün olduğunca kısa olması gerektiği sonucu çıkmaktadır. JVM mucitleri bunu göz önünde bulundurarak paralel çalışan minor Garbage Collection algoritması geliştirmişlerdir.

Paralel çalışan minor Garbage Collection algoritması, tekil threadle işini gören minor Garbage Collection ile teoride aynı yapıdadır. Aralarındaki tek fark, parelel minor Garbage Collection için birden fazla threadin çalışmasıdır. Bu şekilde stop the world duraklamaları daha kısaltılmış olur. Bu uygulamanın kullanıcıya karşı cevapkanlık oranını artırmaktadır.

Paralel minor Garbage Collection nasıl çalışmaktadır? Garbage Collector'ün temizleme işlemine hayatta kalan nesneleri işaretleyerek (mark) başladığını yazmıştım. Garbage Collector bu amaçla kök setinde yer alan nesneleri tarar ve bu nesnelerden yola çıkarak işaretleme işlemini yapar. Akabinde hayatta kalan tüm nesneleri aktif olan Survivor alanına kopyalar (copy). Paralel Minor Garbage Collection için kök setinde yer alan nesneler birden fazla Garbage Collection threadine atanır. Birden fazla thread paralel olarak kök setindeki bir nesneden yola çıkarak hayatta kalan nesneleri tararlar. Bu şekilde işaretleme performansı artırılır ve işaretleme için harcanan zaman azalır.

Hayatta kalan nesnelerin aktif olan Survivor alanına da kopyalanma işlemi buna benzer bir şekilde gerçekleşir. Her threade Survivor hafıza alanında PLAB (PLAB – Parallel Local Alocation Buffer) ismi verilen bir alan atanır. Garbage Collection threadleri birbirlerinden bağımsız olarak keşfettikleri ve hayatta kalmış olan nesneleri kendi PLAB alanlarına kopyalarlar.

Paralel Garbage Collection paralel çalışan threadlere rağmen dünyayı durduran bir algoritmadır. Tekil threadle çalışan Garbage Collection algoritmasına göre daha hızlı çalışır ve uygulamanın durma zamanlarını kısaltır. Birden fazla işlemcisi (CPU – Central Processing Unit) olan bir sunucu üzerinde paralel Garbage Collection algoritmasını kullanmak bu kazancı getirir. Lakin sadece bir işlemcisi olan sunucularda paralel Garbage Collection'ın kullanılması tam tersi bir etki yaratır.

Yazımın ikinci bölümünde major Garbage Collection'ın nasıl çalıştığını inceleyeceğim.

Neden Fonksiyonel Programlamayı Öğrenmek Zorundayız

http://www.kurumsaljava.com/2012/12/30/neden-fonksiyonel-programlamayi-ogrenmek-zorundayiz/

Daha dün gibi hatırlıyorum: Windows 95'in sahip olduğu işletim sistemi çekirdeğini (kernel) taskmanageri üzerinden şutlayabiliyordunuz. Akabinde tüm sistem çalışmaz hale geliyordu. Bu konularla ilgisi olmayanları kendine hayran bırakmak için fena bir yöntem değildi.

Bu bahsettiğim bilgisayarlar birkaç MB hafızası olan, tek bir işletim birimine (CPU) sahip, bir düğmesine basıldığında işletim biriminin çalışma hızını ikiye ya da üçe katlayan, bugünkü perspektiften bakıldığında taş devrinden kalma bilgisayarlardı. Geliştirilen uygulamalar da bu bilgisayarları limon gibi sıkıp, son hafıza hücresine ve işletim birimi döngüsüne (CPU cycle) kadar kullanmaya çalışırlar, ama kaynak yetersizliğinden dolayı bunda pek başarılı olamazlardı. Kısaca o zamanlarda kullanımda olan donanımın geliştirilen uygulamalar için yetersiz olduğunu söyleyebiliriz. Uygulamalar bir sonraki işletim birimi jenerasyonu ve daha fazla hafıza alanı ile hızlanır, bu donanımlarda geliştirilen uygulamalar aynı performansı göstermek için bir sonraki donanım jenerasyonunu beklemek zorunda kalırlardı.

Yazılımın hep böyle donanımı köşeye sıkıştırması, donanım mühendislerinin ışık hızı sınırlarına dayandıktan sonra, işletim birimlerinin hızını alıştıkları gibi artıramayacaklarını anlamaları ve çok çekirdekli donanım mimarisine yönelmelerine kadar devam etti. Bu noktaya gelene kadar yazılım mühendisleri donanım mühendislerine takılarak, "biz birden fazla işletim birimi varmış gibi programlarımızı yazarız, aynı işletim birimini paylaşarak çalışırlar ve en azından geliştirdiğimiz uygulamalar kullanıcıya birden fazla işletim birimi varmış gibi numara yaparlar" söylemleri ile yetindiler. Doğal olarak bu tür programlama tarzına alıştılar. Birden fazla işletim birimli sistemler çıktığında semaforlar ve threadler yardımı ile geliştirdikleri uygulamaları birden fazla işletim birimi üzerinde koşturmaya devam ettiler, eski program yazma stillerinden ödün vermeden, bunun gelecekte programcı olarak var olma ya da olmamanın cevabı olacağının farkına bile varamadan. Yazılımcıların çoklu thread (multithreading) uygulamaları geliştirmekte başarılı oldukları söylenemez. Günümüzde bile birçok programcı bir Java uygulamasında iki threadin senkronizyonunda zorluk çekmektedir.

Günümüzdeki kullanımda olan modern bir bilgisayarın en az iki ya da dört çekirdeği var. Bu yazıyı 2.8 GHz hızında, iki çekirdekli bir notebook üzerinde yazıyorum. Kullandığım işletim sistemi yükü eşit bir şekilde bu çekirdeklere dağıtmaya çalışıyor. Ne kadar başarılı olduğu tartışılır. Kullandığım bazı programlar birden fazla çekirdeğin üzerinde koşacak şekilde geliştirilmemiş. Bu yüzden beklenen performansı sağlayamıyorlar. Donanım mühendislerinin bir köşeden bizi izleyip "iki çekirdeği bile doğru dürüst programlayamıyorlar, kısa bir zaman sonra binli çekirdeklerle nasıl başa çıkacaklar acaba" dediklerini duyar gibiyim.

Bir uygulamanın optimal bir şekilde birden fazla çekirdeği kullanabilmesi için, uygulama parçalarının threadler yardımı ile birden fazla çekirdek üzerinde koşturulmaları gerekir. Bu çoğu zaman veri kaybını önlemek için threadler arası senkronizyonu zorunlu kılar. Java dilinde synchronized kelimesi ile birden fazla threadin aynı kod bloğunu koşturması engellenir. Bu tür bir sekronizasyon anlaşılması zor, komplike uygulamaların oluşmasına sebep verir. Tek bir

thread ile istenilen tarzda bir davranış sergileyen bir uygulama, ikinci bir thread devreye girdiğinde çok başka bir davranış biçimi sergileyebilir. Böyle bir uygulamanın bünyesinde barındırdığı hataları debugging yöntemi ile keşfetmek çok zor bir hal alabilir. Kısaca bu tür uygulamaları geliştirmek, koşturmak ve hataları gidermek çok zordur ve çok az sayıda programcı bu durumlarla baş edebilecek yeteneklere sahiptir.

Kullandığımız imperatif (Java, C/C++, Python, PHP, Javascipt) diller thread senkronizyonu problemi haricinde başka bir sorunla karşılaşmamıza neden olmaktadırlar. İmperatif dillerde yapılan işlemler hafızada yer alan verilerin değiştirilmesi üzerine kuruludur. Bunun en basit örneği aşağıda yer alan Groovy kodunda görülmektedir.

```
for(i in 1..10){
    println(i);
}
```

Groovy örneğinde for döngüsü başlatılmadan önce i ismini taşıyan bir değişken oluşturulur. Bilgisayarın hafıyasında i değişkeninin sahip olduğu değeri tutabilmek için 32 bit (bir int) uzunluğunda bir alan kullanılır. i nin başlangıç değeri 1 olduğu için, bu hafıza alanında yer alan değer 1 (00000000 00000000 00000000 00000001) olacaktır. Birinci döngü sona erdikten sonra i ye yeni bir değer atanır. Bu değer 2 rakamıdır. Değişkenin ismi değişmemiş olsa bile sahip olduğu değer değişmiştir. Bu imperatif dillerde sıkça karşılaştığımız bir değer atama işlemidir. Daha öncede belirttiğim gibi imperatif dillerde yapılan işlemler hafızada yer alan değerlerin değiştirilmesi üzerine kuruludur. Uygulama doğrudan hafızaya erişme (sadece kendi hafıza alanına) ve orada yer alan değerleri manipüle etme yetkisine sahiptir. İşlem gören her satır uygulamanın sahip olduğu anlık durumun (state) degişmesi anlamına gelmektedir.

Eğer i global bir değişken ve for döngüsü bir hata nedeniyle sonlanmış olsaydı, i kendisine atanan en son değeri korurdu. Örneğin i değişkeninin sahip olduğu değere bağımlı olarak başka bir threadin işlem yapmak için beklediğini farz edelim. i döngü içinde istenilen değere ulaşamadığı için diğer thread belkide hayatının sonuna kadar beklemek zorunda kalabilirdi. Tabi bazı programlama teknikleri ile bu gibi durumların önüne geçmek mümkün. Lakin görüldüğü gibi imperatif dillerde uygulamanın sahip olduğu anlık durumun koşturulan her kod satırı ile değiştirilebilir olması, bu durumun geçerliliğinin (consistency) korunmasını çok zor kılmaktadır. Aynı durum OOP'de kullanılan nesneler için de geçerlidir. Nesneler sınıf değişkenleri aracılığı ile belli bir duruma (object state) sahiptirler. Sahip oldukları metotlar aracılığı ile bu durum değiştirilebilir. Herhangi bir metot bünyesinde bir hatanın oluşması, sınıf değişkenlerine geçersiz değerlerin atanmasına ve böylece nesnenin korumaya çalıştığı durumun geçersiz hale gelmesine sebep verebilir.

Şimdi on ya da yirmi sene sonrasını hayal edelim. Binlerce çekirdeği olan bir işletim birimini geliştirdiğimiz imperatif tarzı bir uygulama nasıl optimal kullanabilir? Threadleri senkronize etmeye çalışarak bunu başarmamız mümkün değildir. Bu sebepten dolayı şimdiden binlerce çekirdeği olan bir işletim birimini optimal bir şekilde nasıl programlarız sorusuna cevap aramamız gerekmektedir. Aslında bu soruyu sorarak programcı olarak nasıl bir geleceğe sahip olacağımızın cevabını arıyoruz. Programcı olarak geleceğimizin nasıl şekilleneceği bu sorunun cevabında gizli.

Bahsettiğim problemleri aşmanın ve binlerce çekirdeği olan bir sistemi programlamanın yolu, değişkenlere tekrar, tekrar değer atamamaktan geçmektedir. Değişen değer senkronizasyonu mecbur kılmaktadır. Bu mecburiyet tüm çekirdeklerin aynı anda uygulama tarafından kullanılmasının önünde büyük bir engel teşkil etmektedir. Bu mecburiyeti ortadan kaldırmak için kullandığımız veri yapılarını değiştirilemez (immutable) hale getirmemiz gerekmektedir. Bu bir değişkene bir kere bir değer atandıktan sonra, bu değerin bir daha değiştirilemez olması gerektiği anlamına gelmektedir. Bu değişken uygulama son bulana kadar sahip olduğu değeri koruyabildiği taktirde, threadler arası senkronizasyon gerekliliği ortadan kalkar. Değişmeyen bir değer için senkronizasyon gerekli değildir.

Java gibi dillerde bir değişkene sadece bir kere değer atanması ve değişkenin bu değeri koruması final kelime ile sağlanabilir. final int i = 10; şeklinde bir tanımlama, daha sonra i=11; atamasının yapılmasına izin vermeyecektir. i her zaman 10 değerine sahip olacaktır. Bunun yanı sıra konstrüktör parametreleri aracılığı ile oluşturulan ve sınıf değişkenlerine set metotları aracılığı ile sonradan atama yapılmasına izin vermeyen nesneler değiştirilemez (immutable object) türdedir. Bu nesnelerin threadler arası paylaşımı hiçbir sorun teşkil etmez. Bu gibi yapılar thread güvenlidir (thread safe), çünkü içerikleri değişikliğe uğramaz/uğrayamaz.

Binlerce çekirdeği olan bir sistemi optimal şartlarda programlamanın yolu sonradan değiştirilemeyen (immutable) veri yapılarından geçmektedir. Peki bunun fonksiyonel programlama ile ne ilgisi var? Açıklayayım.

Fonksiyonel dillerde fonksiyonların sahip oldukları bazı özellikler şöyledir:

- Fonksiyonlar metot parametreleri aynı olduğu sürece hep aynı neticeyi geri verirler.
- Metot parametreleri fonksiyonlara pass by value mantığı ile verilir. Bu parametreler fonksiyon tarafından değiştirilemez. Fonksiyon yaptığı işlemin değerini geri verebilmek için yeni değişkenler oluşturur.
- Fonksiyonlar kendi bünyeleri dışındaki hiçbir veri ya da durum (state) üzerinde değişiklik yapmazlar. Bu yüzden fonksiyonların yan etkileri (side effects) yoktur. Aynı fonksiyon durmadan koşturulsa bile hiç bir durum değişikliği olmaz.
- Safkan fonksiyonel dillerde bir değişkene bir değer atandığı zaman, bu değer program son bulana kadar değişmez. Bu metot bünyesinde tanımlanan metot değişkenleri için de geçerlidir. Bu sebepten dolayı fonksiyonel dillerde for döngüsü içinde i değişkenine yeni bir değer atamak mümkün değildir. Değer atanmış bir değişkenin değeri değiştirilemeyeceği için yukarıda yer alan Groovy örneğini birebir fonksiyonel bir dilde yazmak mümkün değildir. Bunu gerçekleştirmek için fonksiyonel dillerde rekursiyon (recursive programming) kullanılır.
- Bir fonksiyon geri verdiği değer ile birebir yer değiştirebilir. Bunun ne anlama geldiğini aşağıdaki Clojure örneğinde açıklamaya çalışacağım.

```
(+ 1 1)
```

Yukarıda yer alan Clojure örneğinde 1+1 işlemi yapılmaktadır. + burada kullanılan fonksiyonun kendisi 1 ve 1 bu fonksiyona gönderilen parametrelerdir. Parantezler kullanılarak fonksiyonun nerede başlayıp, kullanılan parametrelerin nerede bittiği ifade edilir.

5 + 9 / 4 * 2 işlemini gerçekleştirmek için Clojure dilinde şu şekilde bir fonksiyon yazabiliriz.

```
(/ (+ 5 9 )  (* 4 2))
```

Bu Clojure örneğinde 3 değişik fonksiyon kullanılmaktadır. Birinci fonksiyon / fonksiyonudur. Bu fonksiyona iki parametre verilmektedir. Birinci parametre 5+9 un neticesi olan 14 değeri, ikinci parametre 4*2'nin neticesi olan 8 değeridir. Yani yukarıda yer alan fonksiyonu şu şekilde de yazabilirdik:

```
(/ 14  8)
```

Görüldüğü gibi bir fonksiyon geri verdiği değer ile yer değiştirebilmektredir. Bu sadece ve sadece fonksiyonun işlem yaparken yan etki oluşturmaması ve aynı parametreler ile aynı neticeyi geri vermesi durumunda mümkündür. Eğer fonksiyon işlem yaparken durum (state) değişikliğine sebep vermiş olsaydı, yapılan durum değişikliğine göre (* 4 2) işleminde 8 değeri yerine başka bir değer geri alınabilirdi ki bu da bölme işleminin yanlış değerler kullanılarak yapılmasına sebep olurdu.

Paralel programlamanın önündeki en büyük engel kullanılan programlama dilinin hafıyaya doğrudan erişerek mevcut değişkenlerin sahip olduğu değerleri manipüle edebilmesidir. Uygulamanın her parçası hafızada yer alan değerleri herhangi bir zamanda değiştirebilir. Bu hem parallel programlamayı hem de uygulamanın anlık hangi durumda olduğunu öngörmeyi zorlaştıran bir durumdur. Değiştirilemez veri yapılarının (immutable data structures) kullanılması uygulamanın sahip olduğu anlık durumu (state) öngörülebilir hale getirmekte ve paralel programlama için gerekli altyapıyı sunmaktadır. Fonksiyonel dillerin değiştirilemez veri yapılarını temel olarak almaları, bu dillerin parallel programlamada daha avantajlı bir konumda olmalarını sağlamaktadır.

Gelecekte imperatif dillerin yanısıra fonksiyonel dilleri de sıklıkla kullanacağız. Donanımın geleceği parallelikte, yazılımcı olarak bizim geleceğimiz de bu paralelliğe nasıl hükmedeceğimizde. Bundan sonra geliştireceğimiz uygulamalar donanımın bize sağladığı her türlü kaynağı en optimal şekilde kullanacak yapıda olmalı. Bunun için yeni dönemde bizi bekleyen yeniliklere hazırlanmalıyız.

Bir Java'cının Gözünden Ruby

http://www.kurumsaljava.com/2013/04/03/bir-javacinin-gozunden-ruby/

Son zamanlarda en çok merak edip, öğrenmek istediğim diller arasında geliyor Ruby. Ruby herkesin dilinden düşürmediği, dinamik veri tipli ve her şeyin istisnasız nesne olduğu bir programlama dili. 1993 yılında Yukihiro Matsumoto tarafından geliştirilmiş. 2006 senesinde Ruby on Rails çatısının oluşturulmasıyla popüler bir web programcılığı dili olmuş.

Bu yazımda bir Java programcısının, yani benim Ruby dilini nasıl algıladığını, bu dilin kendisini nasıl hissettirdiğini sizlere aktarmak istiyorum.

Java'da her zaman şöyle bir şey yapmak istemişimdir:

```
throw new Exception() if(abc....);
```

Böyle bir yapı Java'da mümkün değil. Onun yerine her zaman şöyle bir şey yazmak zorundasınız:

```
if(abc...)
{
    throw new Exception();
}
```

Tek bir satır ile ifade etmek istediğim bir düşünce için Java'da dört satır kod yazmam gerekiyor. Java'nın en büyük sorunlarından birisi bu. Bu zamanla Java'da yazılan programların okunamaz hale gelmesine sebep oluyor. İşin içine birde hata yönetimi (Exception Handling) girdi mi, kodu anlayana aşk olsun. Ruby'de dikkatimi çeken ve Java'da hasretini çektiğim if ve unless yapıları oldu. Ruby'de şöyle bir şey mümkün:

```
puts 'hello world' if x==4
puts 'hello world' unless x==4

unless x == 4
    puts 'hello world'
end

order.calculate_tax unless order.nil?
```

Belli bir düşünceyi Ruby'de olduğu gibi tek bir satır olarak ifade etmenin kodun okunabilirliği açısından çok büyük getirisi var. Bu satırı bir blok halinde yazmak elbette mümkün. Lakin bu ifade edilmek istenen düsüncenin kod arasında kaybolma rizikosunu artırıyor. Burada Ruby'nin takip ettigi filozofinin "ne kadar az kod, o kadar çok ifade edilmek istenen düsüncenin ön plana çıkması" olduğu belli oluyor.

Aynı sadelik Ruby'nin while ve until döngüleri için de geçerli:

```
x = x+1 while x < 10

x = x-1 until x == 10
```

Ruby'de her şey nesne:

```
>>  4.class
==> Fixnum
>>  4+4
==> 8
>>  4.methods
==> ["inspect", "%", "<<", "singleton_method_added",
        "numerator","*","+","to_i"......]
>>  true.class
==> TrueClass
>>  false.class
==> FalseClass
```

Java'da 4 rakamı int primitif tipine sahip iken, Ruby'de Fixnum tipinde bir nesne. 4.methods bu nesnenin sahip olduğu metotları sıralıyor. Ruby safkan bir nesneye yönelik programlama dili.

Ruby'de duck typing ismi verilen ilginç bir konsept uygulanmış. Bunun ne olduğunu açıklamadan önce bir Java örneği sunmak istiyorum.

```
public void printFileName(File file)
{
    if(file.exists())...
}
```

printFileName() ismini taşıyan metodun file isminde bir parametresi bulunuyor. Metot gövdesinde bu parametrenin sahip olduğu sınıf tipine göre değişik metotlar koşturabilirim. Bunlardan bir tanesi exists() metodu. Bu metodu kullanabilmem için file ismindeki metot parametresinin File sınıfından türetilmiş olması gerekiyor, aksi taktirde exists() metodunu kullanamam. Java'da bir nesnenin sahip olduğu tip o nesnenin sınıfı tarafından belirleniyor. File tipinde olan bir değişken sadece File sınıfının metotlarını kullanabilir. Ruby'de durum farklı. Aradaki farkı görmek için bir Ruby örneğini inceleyelim:

```
def copy_file(obj, dest_path)
    src_path = obj.get_path
    puts "Copying file \"#{src_path}\" to \"#{dest_path}\"."
end
```

copy*file() metodunun obj ve dest*path isminde iki tane parametresi bulunuyor. Bu iki parametrenin hangi tipte olduğu belli değil. Tipini, yani sınıfını tanımadığım bir nesnenin nasıl olurda obj.get*path() şeklinde bir metodunu koşturubilirim? obj nesnesinin ait olduğu sınıfta get*path() isminde bir metod olduğunu nereden biliyorum? Bilmiyorum, bilmek te zorunda değilim.

Ruby'de her şey nesne. Durum böyle olunca Ruby'nin kontrol etmesi gereken tek şey, kullanmak istediğim metodun nesnede var olup, olmadığı. Eğer kullandığım metodu Ruby nesnede

bulamassa NoMethodError hatası oluşturur. Bunu da yakalayıp, gerekli işlemi yapmak zor değil. Buradan çıkardığımız sonuç şu: Ruby nesneleri üzerinde istediğim herhangi bir metodu koşturabilirim. Eğer nesnenin böyle bir metodu varsa, nesne istediğim şekilde davranış sergileyecektir. Aksi taktirde NoMethodError hatası oluşacaktır. Bu Ruby'de nesne tipinin sahip olunan sınıfa göre değil, nesnenin ihtiva ettiği metotlara göre belirlendiği anlamına geliyor.

Ruby'de Java ya da C# da olduğu gibi bir sınıf tanımlamak zorunda kalmadan şu şekilde bir metot tanımlanabiliyor:

```
def  bugunHavaSicakMi
    true
end
```

Ruby'de her fonksiyon bir değer geri veriyor. Eğer metodun son satırında return komutu kullanılmadı ise, o zaman son satırda yer alan değer geri veriliyor. Yukarıda yer alan örnekte geri verilen deger true olacaktır.

Bir array şu şekilde tanımlanabiliyor:

```
>>  hayvanlar = ['Kedi', 'Fare', 'Yilan']
==> ["Kedi", "Fare", "Yilan"]
>>  puts hayvanlar
Kedi
Fare
Yilan
>>  hayvanlar[0]
==> Kedi
>>  hayvanlar[10]
==> nil
>>  hayvanlar[-1]
==> "Yilan"
>>  hayvanlar[0..1]
==> ["Kedi", "Fare"]
```

Var olmayan bir array elementine erişmek istediğimizde Ruby nil değerini geri veriyor. Java'da bu IndexOutOfBoundException hatası olurdu. Index olarak eksi değerler kullanıldığında Ruby arrayi sondan başa doğru görmeye başlıyor. Örneğin hayvanlar[-1] "Yilan", hayvanlar[-2] "Fare" değerini geri verir.

Ruby ile ismi olmayan anonim fonksiyonlar tanımlanabiliyor. Bu anonim fonksiyon başka metotlara parametre olarak ta vermek mümkün. Aşağıdaki örnekte hello world ekranda üç sefer görüntülenir. Küme parantezi içinde yer alan kod aninim fonksiyondur.

```
3.times {puts 'hello world'}
```

Anonim fonksiyonları parametrelendirmek mümkündür. Aşağıdaki örnekte each bir for döngüsü oluşturup, hayvanlar isimli listenin her elementini döngü içinde anonim fonksiyona parametre olarak vermektedir. Tek satırlık kod ile tüm listenin içeriğini ekranda görüntülemek münkündür.

```
hayvanlar = ['Kedi', 'Fare', 'Yilan']
hayvanlar.each { |a|  puts a}
```

Java'da anonim fonksiyon tanımlamak için bir anonim class implementasyonu oluşturmak gerekiyor. Bir listedeki değerleri anonim bir fonksiyon tarafından ekranda görüntülemek için aşağıdaki şekilde Java kodu yazardık:

```
package xxx;

import java.util.ArrayList;
import java.util.List;

public class Test {

    final static List<String> animals = new ArrayList<String>() {
        {
            add("dog");
            add("cat");
        }
    };

    private static abstract class AnimalLister {
        abstract void show(String animal);
    }

    public static void main(String[] args) {

        each(new AnimalLister() {
            @Override
            void show(String animal) {
                System.out.println(animal);
            }
        });
    }

    private static void each(AnimalLister animalLister) {

        for (String animal : animals) {
            animalLister.show(animal);
        }
    }
}
```

Java'da bir sınıfa yeni bir metot eklemek için ya o sınıfı yeniden derlemek, ya AOP kullanmak ya da sınıfı bytecode seviyesinde manipüle etmek gerekiyor. Ruby'de mevcut bir sınıfa yeni bir metot eklemek çocuk oyuncağı:

```
class Fixnum
  def my_times
    i=self
    while i > 0
      i = i-1
      yield
    end
  end
end

3.my_times {puts 'hello world'}
```

Yukarıda yer alan Ruby örneğinde Fixnum isimli sınıfa my*times isminde yeni bir metot ekledik. Oluşturduğumuz bu yeni metodu 3.my*times şeklinde kullanabiliriz. JDK içindeki bir sınıfa doğrudan yeni bir metot eklemek istediginizi düşünün. Ruby'de mevcut her sınıfa yeni bir metot eklemek mümkün.

Java gibi nesneye yönelik programlama yapılan dillerde ortak kullanılan metot ve değişkenler kalıtım yolu ile altsınıflara verilir. Ruby'de bu mekanizma modüller aracılığı ile işlemektedir. Bir modül içinde fonksiyonlar ve değişkenler yer alır. Herhangi bir Ruby sınıfı tanımlı bir modülü kendi bünyesine kattığında, modül içinde yer alan tüm metot ve değişkenler sınıf tarafından kullanılabilir.

Aşağıda yer alan Ruby örneğinde ToFile isminde bir modül yer alıyor. to*f metodu to*s metodundan gelen değeri bir dosyaya ekliyor. to*s metodu modül bünyesinde tanımlı değil. Bu metodun modülü bünyesine katan sınıf tarafından implemente edilmiş olması gerekiyor. Ruby daha önce bahsettiğim duck typing yönetimi ile nesnenin to*s metoduna sahip olup, olmadığını kontrol ediyor.

Person sınıfı include ToFile ile modülü bünyesine katıyor ve to_f metodunu kendi bünyesinde tanımlıymış gibi kullanabilir hale geliyor.

```
module ToFile
  def filename
    "object_#(self.object_id).txt"
  end

  def to_f
    File.open(filename, 'w') {|f| f.write(to_s)}
  end
end


class Person
  include ToFile
  attr_accessor :name

  def initialize(name)
    @name = name
  end

  def to_s
    name
  end
end

Person.new('özcan acar').to_f
```

Bu örneği Java'da aşağıdaki şekilde yazabilirdik. Java'da sadece tanımlı olan metotlar kullanılabilir. Ruby ToFile modülü örneğinde yer alan to*s metodunu kullanabilmek için bu metodun bir sınıf bünyesinde tanımlı olması gerekir. Aşağıda yer alan örnekte To*S isminde bir interface sınıf tanımlayarak, to*s metodunu bu sınıfa ekliyoruz. Soyut olan ToFile sınıfı bu interface sınıfı implemente ettiği için tos metodunu tanır hale geliyor. Person sınıfı ToFile sınıfını genişlettiği için tos metodunu implemente etmek zorunda. Person sınıfı ToFile sınıfını genişlettiği için to*f metoduna sahip oluyor. Person nesnesi tarafından to*f metodu koşturulduğunda, to*f metodu person nesnesi bünyesinde yer alan to_s implementasyona erişip, gerekli değeri edinebiliyor.

Ruby ile Java arasındaki en büyük fark, Java'nın kullanılan metotların hangi sınıflara ait olduklarından haberdar olması zorunluluğu. Ruby duck typing mekanizması ile bu zorunluluğu ortadan kaldırıyor.

```
public interface To_S {

    String to_s();
}
public abstract class ToFile implements To_S {

    public String filename() {
        return this.getClass().getSimpleName();
    }

    public void to_f() {
        FileUtils.writeStringToFile(new File("xxx"), to_s());
    }
}
public class Person extends ToFile{

    private String name;

    public Person(String name) {
        this.name = name;
    }

    @Override
    public String to_s() {
        return this.name;
    }

    public static void main(String[] args) {
        new Person("özcan acar").to_f();
    }
}
```

Bu kısa tanıtım umarım Ruby diline ilginizi çekebilmiştir. Ruby dilinin ifade edilmek istenen düşüncenin çok az satır ile, düşüncenin kod kargaşası arasında kaybolmadan implemente edilmesini mümkün kılması beni etkileyen en önemli özelliği oldu. Benim için ön planda olan kodun nasıl yazıldığı değil, düşüncenin kod olarak nasıl ifade edildiğidir. Programcılıkta zamanla ilgi alanı problem çözme, algoritma kullanımı vs. gibi konulardan, düşüncelerin sade ve çok kısa kod parçası olarak kodlanabilmesi yönüne kaymaktadır. Bu en azından benim için geçerli olan bir durumdur. Bu sebepten dolayı Ruby gibi düşüncelerin sade ve yüksek bir soyutluk seviyesinde ifade edilmesine izin veren dilleri tercih ederim.

Sosyal Medyada Paylaştığım Sözlerim

Ortaya koyduğu ürünü değiştirmeye korkan meslek sahibine programcı denir.

Kitap okumadan daha iyi olmak isteyenler birde uyuyarak denesin.

Bir devrin programcısı olmak icin uzmanlaş!

Bir okuma-yazma var, bir de okuyup, yazma var.

Nitelik arttıkça, gönül alçalır, çünkü farkındalık çoğalır.

Günlük yaşam sadece sizi en iyi olduğunuz ve çok sevdiğiniz şeylerden uzak tutmak için var.

Okumayı sevmeyenin, kod yazmayı sevmesi en büyük tezat!

Bir toplumun gelişmişliği sahip olduğu mutfak zenginliği ile ters orantılıdır.

Saygıyı veren alır.

Yaptığın işlerle değil de sahip olduğunu düşündüğün ünvanlarla anılmak, abesle iştigalin en belirgin ibaresidir.

Yedi adımda programcı olmanın sırrı demek, roketi icat edecek bilgi lazım değil, aya kement atıp, çıkarım demek gibi bir şey...

Bitirmek için başlamamışsan, henüz başlamamışsın.

Cahilliği süs gibi takınmak kolay olduğundan, süs olmayan bilgiye itibar az.

Çoğu Java kodu JVM Hotspot tarafından optimize edildiğinden hızlı, programcısı tarafından oyle tasarlandığından değil.

Bedeli ödenmeyem herşey emanettir.

Çok çalışmak sadece bir sendrom. Altında yatan ise doğru tutkuya sahip olmak.

Programcının kariyeri bilginin onu götürdügü yerdir.

Vücut harici derdi mülkiyet, sıkı dur geliyorum edebiyet.

Aslında hiç bir şey anlamamak çok iyiye alamet... Bknz. http://www.kurumsaljava.com/2012/08/12/aci-cekmeden-ustad-olunmaz

Teknik borcun (http://goo.gl/BU4rMj) en büyük maliyeti bir zaman sonra kaçan programcıdır.

Bir noktadan başka bir noktaya gitmenin en kısa yolu, işin ustasını ziyaret etmekten geçer.

Çok yazmak çok biliyorum değil, çok öğreniyorum demektir.

Yakından tanınmayan insanlar hakkında kafada olusan imajın %99 u öngörüşten ibarettir ve %99 oranında yanlıştır.

Programcılıktan yöneticiliğe terfi cüzdan için yükseliş, beyin için çöküştür. Bunu amaçlamak beyni paketleyip, rafa kaldırma rızasıdır.

Mister Miyagi programcı olsaydı, öğrencilerine kod katası yaptırırdı. #kodkatacom

Marifet egoya değil, vicdana tezahürat edildiğinde harekete geçebilmektir.

Her yazılımcının bilmesi gereken 20 konu... diye bir şey olamaz. Yazılımcının ihtiyacı olan tek şey meraktır, nasıl yapıldığını merak etmek....

Beraber çalışılan insanlardan bir şey öğrenilemiyorsa, orada kalmanın tek sebebi, statüyü koruma eylemidir.

Anlamayarak yapan sadece kullanıcı, anlayarak yapan bilginin gerçek sahibidir.

Birisi sana hocam, üstadım diye hitap ediyorsa, bu senin sahip olduğun değil, karşı tarafın sana verdigi değerdir. #havayagirmeme

At binenin, kılıç kuşananın, kod da onu yeniden yapılandırabilenin.

Kendinize verdiğiniz ünvanlar parayla satın aldıklarınız kadar kıymetsizdir.

Programcılığa olan sevdanın büyüklüğü, okumasanda sahip olmaktan haz aldığın senden yaşlı kitapların adediyle doğru orantılır.

Aslında hiç bir şey anlamamak çok iyiye alamet... http://www.kurumsaljava.com/2012/08/12/aci-cekmeden-ustad-olunmaz/ ...

*Bir ipte iki cambaz, bir kod biriminde birden fazla sorumluluk olmaz. *

Programcıya yol gösteren sadece mantığı ve kullandığı araçlardır.

Bilgiyi paylaşmayanın ya bilgisi yoktur ya da egosu vardır.

Sahip olduğunuz tecrübeyi tekrar ederek yeni tecrübe sahibi olamazsınız, yani tecrübe tecrübeyi doğurmaz.

Bilgi hisse senediyse, o zaman kazanmak için geniş çaplı çeşitliliğe (diversification) gidilmeli.

Yazılımda, pratik yapmak başarının annesidir. #kodkatacom

Sen kendine bir şey öğretemezsin, projelerin sana bir şeyler öğretir.

Java'yı zor kılan, JVM'in yazılımcılar tarafından kara kutu gibi görülmesidir.

Yazılımda iyi olmanın sırrı sürekli değişikliğe maruz kalmakta gizlidir.

Yazdığın kodu bir süre sonra beğenmiyorsan, öğreniyorsun demektir.

Hayat yuvarlayarak, götürmeye çalıştığımız tatlı bir küp şekeri. Marifet küpü küreye dönüştürebilmek.

Kafalarda yarım kalan projelerin en güzel şahitleri mezar taşlarıdır.

Örnek alındığında değil, rekabet edildiğinde öne geçilebilir.

Yaptığın işin arkasını çorap söküğü gibi getirmek istiyorsan, önce bilgi ve ilimle o çorabı örmen lazım.

Kişisel gelişim konusunda ilerleme kaydedemiyorum diyenler, aslında zora gelemiyorum demektedirler http://www.kurumsaljava.com/2012/08/12/aci-cekmeden-ustad-olunmaz

Çoğu yazılımcının diğer yazılım paradıgmalarını öğrenmelerindeki ana güçlük, imperatif (imperative) program geliştirmeye alışmış olmalarıdır.

Bilgi çağına paralel yaşıyor olmanın ibaresi nedir? Her sabah bugün ne öğrenirim sorusu yerine, bugün ne yerim sorusuna cevap aramak...

Nesneye yönelik programlama da kalıtım kullanmak iç güveysi olmak gibidir. Üst sınıf ne isterse, nasıl isterse, o olur!

Okuduğunda anlar, uyguladığında öğrenir, pratik yapmadığında unutursun... http://www.kurumsaljava.com/2012/04/07/kod-kata-ve-pratik-yapmanin-onemi

Kötü programcı yoktur. Kullandığı yazılım geliştirme paradigması yetersiz ya da uygun değildir. Berbere balta verseniz, o da adam akıllı saç kesemez. Ama bu onun kötü bir berber olduğu anlamına gelmez.

Nesneye yönelik programlamada bağımlılığın diğer bir ismi kaos.

İç bağımlılık (cohesion) ne kadar düşükse, dış bağımlılık (coupling) o kadar yüksek olacağından, uygulamayı geliştirmek zorlaşır.

Hayatı kolaylaştırmak için çözüm üreten herkes mühendistir.

Refactoring için yöneticiden izin istemek temel ihtiyaç olan uyku için izin istemek gibidir. Terzi de nasıl dikiş atacağını kimseye sormuyor.

Bir gün daha fazla yaşayabilmek için kırk yılda kazandığın parayı gözünü kırpmadan feda ederdin.

Çok maaş çok sorumluluk ve çok mesai anlamına gelebilir. Mutlu eden paranın fazlalığı değil, kendine ayırabildiğin zamanın çokluğudur.

Dünyayı ben yarattım edasının ilacı, bunu düşünmeyi mümkün kılan kendi yaratıcının olduğunu unutmamaktır.

İyi programcalarla değil, sadece iyi araç ve yazılım metotlarıyla bulgarın önüne geçilebilir.

Azim çalışır, tutku çalıştırır.

Çoğu insan azimliyim yerine hırslıyım diyor. Azim != Hırs. Azim başı göğe erdirir, hırs eninde, sonunda yerin dibine sokar.

Tutku başarının anasıdır, azim de babası.

Bir sınıfı test edebilmenin anahtarı new direktifinde yatar. New ile bir nesne oluşturamıyorsam, o sınıfı test etmek için kırk dereden su getiririm.

Merhamet olmadan maneviyat olmaz.

Önemli olan bilgi sahibi olmak değil, sahip olunan bilgi ile ne yapıldığıdır.

Maneviyat ihtiva etmeyen terbiyeden geri vermesini bilmeyen tüketiciler türer. Tüketimde sınır da tanımazlar.

Evde eli alet tutan eşinin, iş yerinde hatasız kod yazan patronunun gözüne girer ;-)

Devop'ların yanında birde elektronikle ugraşan programcılar var. Onlar Devronik.

Hedef mıknatıs gibidir, kendine doğru çeker.

Başarı kazanılan parayla ölçülseydi, spor toto milyonerleri dünyanın en başarılı insanları olurlardı.

Severek öğrendiğin her yeni bir şey zihnine yeni bir kanat takar.

Gerçek birim (unit) testleri ağ kablosu çekildiğinde bile çalışan testlerdir. Ağsız çalışmıyorsa, entegrasyon testidir.

Yazılımda fazla mesai yapılmayacak diye bir kural yoktur. Lakin fazla mesai bir kural haline gelmemelidir.

Başarı devamlı kurban vermektir.

Bilgi girdiğin karanlık oda için ışıktır.

Yapılması gereken işlerde en büyük sorun zaman yetersizliği değil, atılması gereken bir sonraki adımın bilinmemesidir.

Ne oldum değil, yarın ne öğreneceğim demeli.

Paranın kıymetini bir şeyler üretip, sattığında anlarsın, maaş alarak değil. Hele, hele ürettiğin şeyler geçim kaynağın olmak zorundaysa...

Yazılımda başarının sırrı başkalarının bireye bir şeyler ögretmesi değil, bireyin kendi kendine öğrenebilme kabiliyetine sahip olmasıdır.

Öğrenmenin en efektif yolu hizmet etmektir, çırağın ustasının ihtiyaçlarını görmesi gibi. Bu hizmet ustaya ya da insanlığa olabilir.

Sahip olduğu API'yi sunduğu sınıfları genişleterek kullanmaya zorlayan çatı bankadan kredi

çekmek gibi kullanıcısını kendine bağımlı kılar.

Clean code için şimdi vaktimiz yok. Ne için var? Projeyi adım, adım batırmak için mi?

Objektif kalabilmenin sırrı, acı verse de karşıt görüşe kulak vermeden geçer.

Amaç olmadan motivasyon balon gibi zamanla söner.

Aklında en az bir proje ile etrafta gezinmeyen şahsı sadece emeklilik hayalleri mutlu eder.

Başarının habercisi istikrarlı başarısızlık serisidir.

Gerçek özgürlük ve ayrıcalık tutku ile bağlı olduğun işi yapabilmektir.

Her insan potansiyel bir iyilik meleğidir.

Parlak ve yapıcı fikirler mutlu ve zinde akıllara gelir.

Sağlam kafa sağlam vucutta bulunur. Sağlıklı programcı iyi programcıdır.

Suratından gülücüğü düşüren, insan ilişkilerini yerden toplar.

Özür dileyebilmek.... Şismiş egoların çok uzağında olan hayattaki en önemli yetilerden bir tanesi.

Yazılım bir sanattır, yaşam biçimidir demiş. Deterministik olmak zorunda olan mühendis kafasıyla nasıl bağdaştırıyorlar, merak ediyorum!

Programcının hayatını zorlaştıran bilginin yarı ömrünün çok düşük olması değil,

mikroişlemcilerdeki çekirdek sayısının devamlı artmasıdır.

Arduino ile Scrum'ın ortak yanı nedir? İkisinin de var oluşlarının temellerindeki prensipleri maskelemede üstüne yoktur.

Başarılı olmak için daha çok çalışman gerektiğini ne zaman anlarsın? Başarılı olmak için daha çok çalışmaya başladığında.

Az para veriyorlar diye değil, bana bir şey öğretemiyorlar diye yeni iş aranır.

Başkası için çalışmak onun geleceğini inşa etmektir.

Mikrodenetleyicilerle uğraşmak girişimci ruhun beyin jimnastiği aktivitelerinin çalışma sahasını genişletiyor.

Mikrodenetleyici dünyasına adım atmanın bedeli: 3 adet yanmış ATmega8, breadboard başında sabahlamalar ve santim ilerlememiş olmak.

Toplumların gelişmişlik oranları bireylerinin bilgi playlaşımı için kurdukları dernek sayısıyla doğru orantılıdır.

Projenin sallantıda olduğunun en sağlam indikatörü nedir? Yazılımcıların dua edelim de olsun ya da insallah yarın çalışır demeleri.

İyi programcı karmaşayı koddan uzak tutandır, içine sokan değil!

Para tutanın, aşk sevenin, bilgi paylaşanındır.

ÜDD (Ümit Driven Development) nin diğer güzel bir örneği paralel çalışan bir programı logging kullanarak anlamaya ve debug etmeye çalışmaktır.

ÜDD (Ümit Driven Development) nin en guzel örneği, ana nedenini (root cause) anlamadan kodu değiştirerek, sorun çözmeye çalışmaktır.

Neden olduğunu anlamadan sorun çözmeye kalkmak, bir uçağın kokpitinde şu düğmeye de başalım, bakalim ne olacak demek gibidir.

Temel elektronik bilgisi olmadan mikrodenetleyicilerle ilgilenmek, nota bilmeden müzik yapmak gibi bir şeydir.

Eksi(k) olan elektron çekirdeğine gitme egilimindeyse, eksi(k) bilgili çırak ustasına gitme, onu arama eğiliminde olmalıdır.

Aklında en az bir proje ile etrafta gezinmeyen şahsı sadece emeklilik hayalleri mutlu eder.

Filozofa sormuşlar dürüstlük nedir diye. Beden için acı, ruh için ilaç demiş.

İnsan oğrendiği her şeyi çok çabuk unutur. Onların beyinde kalıcı olmalarını sağlamanın tek yolu pratik yapmaktır. #kodkata

Müşterinin yüzünü güldürmekdikten sonra en muazzam teknik çözüm bile hiçbir şey ifade etmez. #müşterigüdümlüyazılımcı

Olduğundan az görünmeyi zor kılan tersinin çok kolay olmasıdır. İnsan her zaman kolayı tercih eden bir mahluktur.

Ne zaman daha iyi bir programcı olurum? Kafamı koddan kaldırıp, programcılığın diğer yönleri ile ilgilendiğim zaman. #softskills

Büyük resmi görebilmek icin uzman değil generalist olmak gerekir.

Kitap okumayla programcı olunsa idi, dünyada iyi programcıdan gecilmezdi. #pratikyapanadam

Her sistem en zayıf yerinden patlak verir, yazılım sistemlerinde bu test edilmeyen kod birimleridir.

Müşterisini memnun edemeyen programcı gereksiz işler müdürü olarak sadece kendini mutlu etmiştir.

Refactoring kodu yoğurmaksa, yeni bir programlama dili ögrenmek beyni refactor etmektir.

Patronun değil, müşterinin ne istediği önemli.

Yemek seçenler ile progranlama dili seçenlerin ortak yanı: ilerki bir zamanda aç kalabilme ihtimaline yakın olmaları.

Benlik gütmek yerine benliği gütmek.

Senior olmayı bazıları sadece bilgi kûpü olmak ve daha fazla maaş almak olarak tanımlıyor.

Bir anlık öfke bin kalbi kırabilir, bir kalbi kazanmak bin yıl sürebilir.

Amaç olmadan motivasyon balon gibi zamanla söner.

Bi çalışmak için öğrenmek var, bi de öğrenmek için çalışmak...

Yeri gelince yazdığı kodu çöpe atabilen kodun efendisi, bunu yapamayan kodun kölesidir.

Egoist bir çırak zamanı gelince belki iyi bir programcı olabilir, ama ismi çırak yetiştirmediği için tarihin sayfalarında yer almaz.

Her gün bir kod katası yapmayan ya tembeldir ya da kata yapmayı küçümsüyordur. Demek oluyor ki tembellik ve kibir gelişimin önündeki engellerdendir.

Usta çırağını aramaz, çırak ustasını bulur. Çırağın ilk imtihanı budur.

Boş zamanlarımda yazılımla ilgilenmiyorum, sadece mesai saatlerinde demek yazılımı sevmiyorum, sadece ekmek kapısı olarak görüyorum demektir.

Esnek bağımlılığın en guzel hali.. EDA (Event Driven Architecture) ile SEDA (Staged EDA)

Yapacak iş kalmayınca ne iş yapabilirim diye sormak ta bir geribildirimdir.

Başarısızlık batak gibidir. Çırpınmadan batmak var, süt çanağına düşüp, çırpınarak sütü maya yapan fare misali tekrar bataktan çıkmak var.

Ümit Driven Development; Bir şeyin çalışacağını ümit ederek kod yazmak.

Yazılımcı olarak okudukların değil, uyguladıklarınsın.

BT yaş değil, baş işidir.

Ucuz yazılımcı peşinde olanlara duyurulur: if u pay peanuts, u get monkeys.

Yazdığı kodu Sonar gibi kod metriklerini gösteren araçları tatmin etmek için adapte eden makyajdan başka bir şey yapmamaktadır.

Mimarlar yazılım sisteminin enine, yazılımcılar boyuna doğru gider.

Yeni bir API'yi yapan değil, kullanan tanıtmalı. Yapan detaylarda kaybolur, kullanım konusunda kafaları karıştırır.

Nesneye yönelik programlamayı zor kılan, ne iş olsa yaparım, her şey elimden gelir abi diyen nesnelerdir. #teksorumlulukprensibi

Nesneye yönelik programlama nesnelerin gizemli olmaları ve sahip oldukları bilgileri ve davranışları ulu, orta göstermemeleridir.

Mevcut projede kod parçaları kayboluyor ve kimse bunun farkına varmıyorsa, test konseptlerinin tekrar gözden geçirilmesi gerekir.

Kendinize daha rahat bir iş bakıyorsanız, comfort zonu seviyorsunuz demektir http://goo.gl/Mj187

Başarısızlıklar elektrik kesilmesi gibidir, töm hafızayı resetler ve bitleri sıfırdan programlanır hale getirirler.

Tek yönlü beslenmek iyi değil, sağlığı tehdit eder, Java yanında C#, Python ve Ruby gibi nimetlerden de beslenmek lazım.

Kendi yazdığı kodu test edemeyen yazılımcı motor tamir edemeyen araba tamircisi gibidir. Eninde sonunda elinde patlar.

İyi programcıların azalmasının sebebi uzmanlaşma merağıdır. Uzmanlaşmak sonun başlangıcıdır. http://goo.gl/t2r8A

Bir programcının öğrenmesi gereken en önemli dil: Nesneye Yönelik İngilizce

Bitler saklanbaç oynuyor. Yedi bit saklanmış, sekizinci bit sayıyor: sıfır, bir, sıfır, bir, sıfır dersem çık, bir dersem çıkma....

On parmak yazı yazamayanı sekreter olarak işe almazlar. Programcıyım diyorsa iş değişir.

Nasrettin hocanin ya tutarsa dediğinin yeni türkçesi start-up kelimesidir.

Uçak bileti sağ varma garantisi ihtiva etmez, bilgisayar muhendisi olmakta iyi bir yazılımcı olmayı.

Birçok start-up fikri pazarda limon satmanın eline su dükemez, pazara çıkamadıkları da cabası.

Ana ile başlayan her seyi seviyorum: ana, anavatan, anadil, anadolu....

Gerçek özgürlük ve ayrıcalık tutku ile bağlı olduğun işi yapabilmektir.

Her bitin en büyük hayali usta bir programcının elinde yoğrulmak ve elektrik kesilene kadar kendisini özel hissetmektir.

Nerede o eski programcılar demiş bit diğer yedi kardeşine.

Bilginin en mükemmel hali mütevazilikle birleştiği andır.

Çok bildiğini sanmak kulakları tıkar, karşı tarafı dinletmez, yerin dibine sokar, yine de bunların farkına vardırmaz, çünkü iyi hissettirir.

Kişesel gelişimin en katıksız iki yol arkadaşı: eğlence ve acı. İkisinin de olmadığı yerde arpa boyu yol almak imkansız.

İşveren yerinde olsam yazılımcıları kontrol etmeye çalışmak yerine nasıl verimliliklerini artırabilirim sorusuna cevap bulmaya çalışırdım.

Ekip içinde fikir yürütüp müşteri ne istiyor sorusuna cevap aramak, müsteri gereksiminin analizinin yerini alamaz.

Geliştirilen çoğu program software değil, hardware, çünkü değiştirilmeleri çok zor ya da imkansız.

Müşterinin ne istediğini anlamadan kod yazmak, samanlıkta toplu iğne aramak gibidir.

Test edilmeden eklenen her yeni metot bugdır.

Refactoring yapmadan kod yazmak bakkaldan veresiye almak gibidir. Defter kabardıkça borç batağından kurtulmak imkansızlaşır.

Bu devirde 32 bitlik bir sistem ile yazılım geliştirmeye çalışmak verimli olmanın önündeki en büyük engeldir.

Yazılımda en önemli yetilerden birisi bakış açısını degiştirebilmektir.

Yeni programlama dilleri bir gün bir yerde belki kullanırım diye değil, ufku genişletip, bakış açısını zenginleştirsinler diye ögrenilir.

Ustalık sadece iyi kod yazmak değildir, efendilik, benlik gütmemek, usta yetiştirmek ve canlıya ve doğaya olan derin saygıdır.

Usta olmayı zor kılan teknik değil insani yönüdür.

Rus ruleti ile uzun bir metodun ortak yani nedir? Mutlaka patlarlar ama ne zaman patlayacakları belli olmaz.

İmpetatif tarz program yazma bir kelebeğin kanat çırpışının okyanusun öteki tarafında tsunami oluşmasına sebep olması gibidir. #donttouchstate

Neden Java'nın Closure yapılarına ihtiyacı var diyorsanız bir Swing uygulamasının koduna bakın. #anonimiçsınıflarclosuredeğil

Learning zone duvarları kafayla yarıp geçmek, panic zone denizde boğulmak, comfort zone çay bahçesinde çay içmek gibi gibi bir şey. http://goo.gl/Mj187

Tasarım şablonları gibi algoritmalar da programcılar için ortak kelime hazinesi oluşturur ve iletişimi kolaylaştırır.

Java'nın geleceğini görmek için JVM üzerinde çalışan Scala gibi dillere bir göz atmak yeterli. Bu dillerdeki yeni konsepler Java'ya eklenecek.

Yazılımda çoğu konseptin doğru anlaşılmamasının sebebi, konu hakkında bir kaynak okuyup ve anladığını sanıp devam etmektir.

Aynı metni ilk okuyuş gözü kelimelere aşına eder. İkinci okuyuş cümlelerin anlaşılmasını sağlar. Üçüncü okuyuş metnin anlanımı idrak ettirir.

Okuduğunu anlamıyorsan panic, anlam verebiliyorsan learning, anlıyorsan comfort zone'dasın. Anlıyorsan, öğrenmiyorsun! http://goo.gl/AHk3g

Pop metodu olmayan stack implementasyonu örneği nedir? Daldan dala atlama-öğrenme metodolojosi. Stackoverflow == BirseyOgrenilmediException

Hayat bir porselen dükkanı. Nihayi amaç kırıp, dökmeden dükkandan çıkabilmek.

Java'nın tip sistemi takim elbiseli bir bankacı ise, Ruby'ninki kot ve tshirt giyen genç ve dinamik bir delikanlı.

En sık kullandığım Eclipse kısa yol tuşu: alt+shift+r

Gözden ırak, gönülden ırak. Elektronik ortamda tutulan kullanıcı hikayeleri, duvarda asılı olanlara nazaran çok daha az irdeleniyor. #scrumpano

Uzmanlar comfort zone'larını terk etmeyi sevmez http://goo.gl/AHk3g

Bir dilin syntax'ını bilmek/anlamak, o dilin temsil ettiği konsepleri anlamış olmak anlamına gelmez. Arapça okuyup ama anlamamak gibi...

Kod yazmak inşaatın kendisi, refactoring iç dekorasyonudur.

Sağlam programcı kafası sağlam programcı vücudunda olur.

Uzmanlaşmak teknoloji korkusunu artırır http://goo.gl/AHk3g

Uzmanlaştığınızın en belirgin özelligi kullandığınız tek IDE'den kopamayıp, diğerlerine şans verememenizdir. #ustabilimumidelerikullanır

Bir uygulamanın bakımını zorlaştıran sebeplerden birisi copy/paste, diğeri sınıf hiyerarşileri oluşturup OOP'nin kalıtım özelliğini kullanmaktır.

Java'cılar için statik tip sistemi ne ise, agile'cılar için de Extreme Programming (XP) o. #WYSIWYG

Scrum iyi bir duck-typing örneği, istediğin yere çek.

Statik veri tipli bir dünyadan gelenler için duck-typing ilk başta tuhaf geliyor ama insan zamanla alışıp, sınıflara başka gözle bakıyor.

Fonksiyonel bir dili öğrenmek, pantalonu tersine çevirip, tersinden giymek gibi bir şey.

Dinamik dillerde veri tipi belli olmayan değişkenlere ne denir? Tipsiz ya da tipitip :)

Maaşın yüksekliği ustalık derecesini gösteren indikatör değildir.

Teknoloji avukatlığı yapanlar sözde uzmandır.

10 senede on değişik programlama diline hakim olmanın en kolay yolu, her sene iş yerini değiştirmektir.

Geleceğe en sağlam yatırım, her sene iyi seviyede yeni bir programlama dili öğrenmektir.

İç dünyanın en güçlü stabilizatörû uygulanan meslekten alınan hazdır.

Programcılık == dibi olmayan eğlence kuyusu

Yazılımclar arasında en mutsuz olanlar bu işi para için yapanlarla, daha fazla para kazanmak için yöneteciliğe terfi edenlerdir.

Program yazma paradigmaları düzenli aralıklarla değişiyor. Bir sonraki değişiklik için hazır olmak lazım.

Programcı olarak çok iyi bir iş ortaya koyabilmek için ufkun çok geniş olması, bunun için de yarım ömür okumak, öğrenmek ve pratik yapmak gerekir.

Programcılık yüzme gibi bir şey olsaydı, programcıların yüzde kaçı karaya çıkabilirdi acaba?

Kullandığınız programlama dili bir yemek çeşidi olsaydı, hergün aynı yemeği yemek ister miydiniz?

Bir programcı için beyin, soyutlama, algılama ne kadar önemliyse, sağlıklı parmaklar, göz ve agrısız bir bel de önemli. #öncesağlık

Çevik süreçlerin başarılı olma potansiyelinin önündeki en büyük engel, eski davranış biçimlerini terk edemeyen yazılımcılardır.

Profiline software enginar yazıp, arkasına çırak ibaresini eklemek, aranıp, bulunması gereken yazılımcı prototipine işarettir.

Yaşamınız bir fonksiyon olsaydı son bulduğunda return ile hangi değeri geri vermesini isterdiniz?

Ana gibi yar olmaz, C gibi dürüst programlama dili olmaz.

Sadece senior yazılımcıların olduğu bir ekipte son sözü söyleyecek bir merci olmadığı sürece neyin nasıl yapılacağı devamlı tartışılır.

Karşı taraf konuşurken ne soyleyeceğini kurmak, karşı tarafı değil, kendini dinlemektir.

Profesyonellik söylenenleri kişisel almamak, darılmamak, her zaman alçak ve sakin sesle konuşmak, karşı tarafın söylediklerini dinlemektir.

Kişinin canı gönülden programcılığı sevip, sevmediğini anlamak için bir milyon verseler bu işi bırakır mısın sorusunu sorun.

Hayat kocaman bir stack. Bütün günümüzü push ve pop yaparak geçiriyoruz.

Fonksiyonel dillerin kıymetini bilmek için imperatif dillerin cefasını çekmiş olmak gerekir.

Bir arı kovanıyla bir OOP uygulamasının ortak yani nedir? İkisinde de öğeler arasındaki ilişkileri kavramak çok zordur.

Sadece Java bilen birine dünyanın en iyi dili hangisidir diye sormayın. Objektif fikir beyanı için karşılaştırabilmek şarttır.

Sövme ve övme... Tek bir harfin fazlalığı ya da eksikliği insanlarla olan ilişkinizi nasıl etkiliyor, bir deneyin. #sövmeöv

Okumak != öğrenmek == pratik yapmak

En makbul ekip arkadaşı amansızca tartışıp, çekiştiğin, dövesin geldiği ama öğle ve iş aralarında hoş sohbet yapabildiğin şahıstır.

Kargaya yavrusu kuzgun görünürmüş. Herkes kendi yazdığı kodu beğenir. #kodunabirbakabilirmiyim

Kod yazarken iyi isimler seçemeyen bir yazılımcı bu işin ne kadar hakkını verdiğini sorgulamalı.

Yazılımcı olarak bütün bir gün boyunca sınıf/metot/değişken isimleri okuyoruz, onlara isimler veriyoruz ama yinede bu işi iyi yapamıyoruz.

Swing ile gui geliştirmek ne kadar zevkliymiş. Ön yargılarım gözlerimi kör etmiş. Özür dilerim Swing.

Birim testi piyade ise, entegrasyon testi tank, patriot. Birim testleri her zaman gerçeği söylemez,

entegrasyon testleti de yalan...

Mesai arkadaşlarınızın bir araya geldiklerinde sizin hakkınızda konuştuklarını düşünüyorsanız, yazdığınız koddan emin değilsiniz demektir.

Bilgi ve tecrübenin önünde saygıyla eğiliyorum. Yarım gün uğraştığım algoritmayı ustam 8 satırlık kodla yeniden yazdı. #herkesustaolabilir

Beyin kullanılmayan organların fişini çekiyor. Aynı şeyler kullanılmayan yetiler için de geçerli.

Ölümü en anlamlı kılan şey hizmet, en anlamsız kılan şey ise bir ömür buyu "hep ben" demiş olmaktır.

Sadece mimarların yaşadığı bir dünya hayal dünyasıdır.

Web girişimciliğinin en zevkli tarafı, kurulan platformun kısa sürede kullanıcı sayısı ve içerik olarak rakiplerini sollamasıdır.

Yazdığı teste bak, programcıyı al.

Uygulamayı yeniden yapılandırıp, testlere el sürmeyenler, testlerin kendilerini üvey evlat gibi hissettiklerini bilmezler mi? #canımtestlerim

Hata yönetimini köküne kıran girmiş gibi tek bir exception sınıfı ile yönetmek gırslığın danıskasıdır. #throw new SpecificException();

Saat 17 oldu, mesai bitti, ama sen bir türlü klavyeyi bırakamıyor musun? Program yazmak çok mu zevkli? Maddi/manevi dopdolu bir gelecek senin.

Girişimci yaptığı değil, kafasındaki bir sonraki proje ile yaşar.

Yazılımcılar tasavvuf gibi kavramları çok hızlı ve iyi kavrayabilirler. Kavramak != Yasamak #matrix

Tereyağından kıl çeker gibi sistemin herhangi bir tarafını değiştirebiliyorsan yaptığın tasarım o zaman tamamdır.

Hiç bir şey bilmediğini anlayıp, çırpındıkça, daha derine batıyorsun.

Güzel bir çözüm üreten çalışma arkadaşını övmesini bilmek gerekir.

Büyük edebiyat eserlerini okumadan edebiyat eseri yazılamayacağı gibi,usta yazılımcıların kodunu okumadan iyi programcı olmak mümkün değildir.

Rekursiyonu anlamak isteyenler Inception filmini seyretmeli. Her bir rüyanın içindeki rüya yeni bir stack frame, her kick bir return...

Okunması ve bakımı güç programların en belirgin özelliği nedir? 5 dakikada bulmanız gereken bir hata için bir gün boyu debug yapmak...

İyi bir yazılımcı olmak için edebiyat eserleri okumak lazım. Kazanılan kelime hazinesi ve ifade gücü koda ve diğer yazı türlerine yansır.

Yaptığın işte daha iyi olmanın tek gıdası devamlı yaptığın işte iyi olmadığını düşünmektir.

Deneme, yanılma tarzı kod yazmanın önüne geçmenin en kolay yolu, hemen bir test sınıfı yazmaktır.

Big Oh notasyonuyla yeni bir şey öğrenme hızın nedir?... O(n)? O(n^2)?

Programcının yeni CV'si Github.

Metot satır sayısı programcının kalitesini ölçmek için iyi bir metrik. Ne kadar az, o kadar iyi.

Usta olarak görmedim kendimi, ama kalfa bile değilmişim. Şimdi çırak olarak yeniden başlıyorum.

Bir konuda ustalaşmak için on sene gerekli olduğunu anlamak bile on sene sürüyor.

Yazılımda ustalaşmanın önundeki en büyük engel her gün işe gidip, sekiz saat çalışmaktır.

En az değer verdiğimiz, ama en değerli varlığımız nedir? Verimli bir şeyler yapmak için kullanabileceğimiz ama bosa harcadığımız zamanımız.

Nasıl daha iyi bir yazılımcı olabilirim?

Her fani ölümü tadar. Bazı programcılar her imkana sahip olmalarına rağmen ustalaşamadan fani olarak kalırlar.

Kişisel gelişimde multithreading... Paralel üç programlama dilini öğrenmek.

Ermiş meyve dalında durmaz. Ermiş programcı bildikleriyle yetinmez.

Acaba bunu doğru mu programladım sorusuna cevap arama süresi bir birim testi yazmak için yeterlidir.

Bir yazılımcının yaptığı işe verdiği değer yazdığı metodun satır sayısıyla ters orantılıdır.

Sınıflar egoist, apiler minimalist, modüller koherent, bağımlılıklar esnek olmalıdır.

Arabanın yakıtı benzin, programcının yakıtı bilgi. Yakıtın olmadan ilerleyemessin.

Programcı için popüler dil yoktur, işi için kullandığı dil(ler) vardır.

Programcılıkta bir üst seviyeye geçis, bu kodu nasıl daha kısa ve okunur yazarım sorusu ile başlar.

Neden ustalaşmak isteyen bir müzisyen pratik yapma harici müzik tarihi ile ilgili kitaplar okur? #büyükresmigörebilmek

Ufkun genişlemesi için python, scala, groovy ne varsa öğrenmek gerekiyor. Eskiden ne gereği var, Java yeter derdim. Ufkum ne kadar darmış!

Ufkunuzun genişlediğine canlı canlı şahit olmak istiyorsaniz fonksiyonel bir dil öğrenin. Beyin hücreleriniz size teşekkür edecektir.

Öğrenmeyi öğrenmiş olmanın ibaresi nedir? Başkalarına öğretebilmek.

Code reviewdan kaçan, programcı olarak sorumlu olduğu her şeyden kaçıyor demektir. #dörtgözprensibi

İçinde code review olmayan Definition Of Done yarımdır.

Definition Of Done ne kadar laçka ise, sürecin geri kalan kısmı da o kadar laçkadır. #balıkbaştankokar

İsmi üstünde "kişisel gelişim". Sizi başkası geliştiremez. Kişisel gelişiminizden kendiniz sorumlusunuz. #baskaşındanmedetumma

Fonksiyonel dillerden kaçış var mı? Nayır!

Bir dil, bir insan. Bir programlama dili, bir programcı.

Her Java'cının öğrenmesi, bilmesi ve kullanması gereken üç dil: scala,groovy,clojure...

Gerçek yazılım ustaları süslü, püslü ünvanlar kullanmazlar.

Yeni yazılımcı ünvanları icat edildikçe iyi yazılımcı sayısı azalıyor.

Yazılımda bazı ünvanlar peşinde koşanlar, yazılımın özünün ne olduğunu anlamışlar mı? #ünvandeğilbeceriönemli

Test edilemeyen kod vardır diyen kendini kandırır.

İki boyutlu yaratıklarla sadece oop bilen programcıların ortak yani nedir? Diğer boyutlarda olup, bitenleri kavrayamazlar.

Uzun, ince bir yoldayım, gidiyorum gündüz, gece. Yazılımcı bu türküyü söyler devamlı, ne yaptığını unutmamak için.

*Bugün çok para kazandıran bir programlama dili, yarın aynı şeyi sağlayamayabilir. #birdilbanayeter *

Lead olmayı kabul eden yazılımcı olmaya veda etmiş demektir.

"Yazılımda ustalaşmak ne kadar sürer?" diye sormuş öğrenci. En uygun cevap "ne kadar daha yaşarsın?" sorusu olur demiş usta.

Korkulan işler kişisel gelişim için nitrodur.

Kod katası yapmak, çiçek tohumlarını sulamak gibidir. http://Kodkata.com

Tekrar tekrar beyaz kuşağı kuşanmayı bilmeyen yazılımcı, ustalaşamaz, sadece bir konuda uzmanlaşır.

Tek bir dili savunan yazılımcılar uzman, çok dili kullananlar ustadır. Tercih sondakinden yana olmalı.

Ne olduğunu anlamaya çalışmak yerine soru sormak....

Beceri bilginin yaşayan halidir.

Kitabı sadece okumak; kitabı okumak ve kitaptaki kod örneklerini uygulamak. Bu ikisi arasındaki fark kişisel gelişim açısından gece ve gündüz gibi.

İki satır kod yazmayla girişimcilik olmaz.

Her yazılımcının böyle bir profil sayfası olmalı... http://goo.gl/1UY8L

Yazılımda ustalaşmanın sonu kendini programcılığa ve senden sonra gelenleri yetiştirmeye adama. Bu safhaya ulaş(a)mayanlar ne kadar usta olduklarını sorgulamalı.

Mingle gibi web tabanlı görev panosu kullanımı çevik sürecin tekerine çomak sokar. Gözden ırak, gönülden ırak!

Yazdığı koddan utanmayanlar el kaldırsın. #elimhavada

Kod inceleme (code review) seanslarının önündeki en büyük engel, yazdığı koddan utanıp, ne

yaptığını göstermek istemeyen yazılımcılardır.

Yazılımda verimlilik okunabilirlikle doğru orantılıdır. Bu sebepten dolayı temiz kod yazın diyorlar ya da dsl gibi icatlar yapıyorlar.

Dağ peygambere gelmesse, peygamber dağa gider. Veri koda gelmesse, kod veriye gider.... #hadoop

Lead dev arıyorlar, en az 3 yıl tecrübeli olması lazımmış. Ekip sadece bir kişiden oluşacaksa, o zaman sorun yok.

Kodu commit etmek altına imza atmak gibidir. Bildiğim en iyi şekilde yaptım, çalışır durumda ve test ettim demektir.

Tek bir assert kullanmadan %100 code coverage oluşturmak mümkün. %100 code coverage kodun en iyi şekilde test edildiği anlamına gelmiyor.

Testlerin kodu yüzde yüz kapsamasını kodun yüzde yüz test edildiğinin ibaresi olarak görenler var.

Yazılımda çeviklik sadece çevikliği kavramış mühendislerle mümkündür.

Yazılım zeka işidir demiş... Yüzde doksan bilmem kaç genetik örtüştüğümüz Orangutan'da da zeka var.

Trafik lambasında kırmızı yanınca bekleyerek küçüklere, temiz kod yazarak ekip arkadaşlarına örnek olmak....

Arı olup her çiceğin nektarını kapmak, freelance yazılımcı olup her projeden yeni bir şeyler ögrenmek...

Akıcı konuşabilmek için kelime hazinesi, akıcı kod yazabilmek için teknoloji ve bilgi hazinesi gerekir.

Test konseptleri içinde en anlaşılmayanı birim (unit) testleri. Entegrasyon testi yazıp, bunun birim testi olduğunu düşünenler var.

Birim testinin gerçek anlamda birim testi olduğunu nasıl anlarsınız? Elli tanesini aynı anda koştursanız bile yarım saniye sürmez.

.Net ya da JEE gibi teknolojilere hakimiyet şahsın iyi bir yazılımcı olduğunun kanıtı olamaz. Yazılımcılık başka bir şey!

Başkasının kodunu anlamıyorsanız, birim testi yazıp, kullanmaya çalışın. Birim testi yabancı kod hakkında hikayeler anlatmaya başlayacaktır.

Bugün kişisel gelişiminiz için ne yaptınız?

Bugün yazdığın koda baktığında onunla gurur duyabiliyor musun? Evet diyebiliyorsan, temiz kod yazanlar kulübüne hoşgeldin.

Yazılımcı olarak görevim yerime başkası konabilecek şekilde iş çıkarmaktır.

Değişkenleri private yapıp, set ve get metotları oluşturmak, evden çıkarken anahtarı kapıda bırakmak gibidir.

Yazılım yapmayı sadece bir iş olarak görenler doğru işi yapıp yapmadıklarını sorgulamalıdır.

Yazılımda paylaşım kültürünün en guzel örneğini açık kaynaklı projeler teşkil etmektedir.

Çok bilgi ve tecrübe sahibi olmak usta olmak mı demektir? Hayır! Bu bilgi ve tecrübeyi paylaşan gerçek ustadır. Diğeri sadece bilgi küpüdür.

Gerçek erdem yıllar süren yazılım serüveninde bir arpa boyu ilerlememiş olmayı görmek ve sil baştan yapabilmektir.

Bir sınıfı test edebilmek için sınıf üzerinde her türlü değişliği yapmak mübahtır.

Sınıf ve metotları final yaparak telif haklarınıza sahip çıkmış olursunuz, kimse değiştirip kendi malıymış gibi satamaz, sadece kullanabilir.

Yaşa değil bilgiye, bilene değil, öğretene saygım var.

Bir sonraki programlama dilini, dili kullanan toplumun kod paylaşma kültürüne göre seçin.

Balkalarının iyi bir yazılımcı olma hevesinizi kırmasına izin vermeyin. Meyva veren agaç taşlanır.

Yazılımcılara bilgi seviyesine göre değil, bilgi edinme heyecanına göre değer biçmeli.

Birim testi olmayan bir kod tabanını nereden test etmeye başlayacağını bilemiyorsun, dağınık bir odayı toplamak gibi.

Bir API'yi öğrenmenin en kolay yolu birim testi yazıp, API ile oynamaktır.

Oluşturduğunuz Java sınıflarını ne kadar iyi saklarsanız, o oranda dış dünyaya hesap verme sorumluluğunuz azalır. #protectedclass

Ustandan daha iyi olamassın. Kulağın boynuzu geçebilmesi için değişik ustaların yanında çalışmak gerekir.

Scrum buzdağının görünen kısmıysa, XP (Extreme Programming) buzdağının kendisidir.

Yazılımda çevikliğe en büyük ihanet bu konudaki cehalet ve bilip, bilmeden fikir beyan etmektir.

Yazılımda kıdem yoktur, matruşka gibi açtıkça içinden yeni dünyalar çıkar. Bir dünyanın kıdemi, diğer dünyanın başlangıcıdır.

Dervişin iki kefesi vardır. Birinde anladıkları, diğerinde anlayacakları. Yazılımcının bir kefesi vardır, içinde bildiğini sandıkları...

Şahıs yazılımcı olacaksa, en iyi yazılımcı olmayı hedeflemelidir. Bu meslekte vasatlığa yer yok.

Zeka sahibi olmanın diyeti bilgiyi paylaşmaktır.

Bilginin yarı ömrünü artırmak için paylaşarak çoğalmasını sağlamak gerekir.

Hello world ile başlayan bir kariyer yönetici olduktan sonra, hüzünle son buluyor. Yazılım sektöründe mutsuz insan sayısı çok yüksek.

Yazılımcının çalışma tarzını konuşarak değiştiremessin. Senin gibi güzel kod yazmaya imrendiğinde kendiliğinden değişecektir. #imrendir

Bilgi paylaşma kültürüne sahip olmayan bir yazılımcının söyleyecek neyi olabilir? Konuşsa, nesi ciddiye alınır?

Programcılar keşke armut gibi agaçta yetişebilse, ya da karpuz gibi tarlada.

Bir programcı günde sekiz saat ve haftada beş gün çalışır, NOKTA!

Üjbej vakitte usta yazılımcı olduğunu düşünenler var. 15 yıl dirsek çürütün, ondan sonra görüşelim.

Programcılık kursları hikaye. Bu işi dizini kırıp, ustanın yanına oturarak, ondan öğreneceksin.

On sene aynı şeyi yapmış olmak tecrübe olarak sayılmaz.

Bilginin sahibi yok, ama vicdanın var. Bilgiyi paylaşın, vicdanınız rahatlasın.

Öğrenileni hazmetmenin en güzel yolu ögrenileni başka birisine öğretmektir.

Kaç yazılımcı kitaba bakmadan insertion ya da merge sort algoritmasını programlayabilir?

Algoritmaları kapalı gözle programlayamayıp gerekirse kitaba bakarım demek böyük yanılgıdır.

*Google'da bir tarayayım bakalım kalp amelıyatı nasıl yapılıyor diyen bir doktor gördünüz mü? Biz yazılımcılar neden her şeyi googlelıyoruz? *

Yazılımcılık hep aynı iş, verileri A'dan B'ye aktarıyoruz, özünde başka bir şey değil.

Çevikliğin özü refactoring ve tdd http://goo.gl/zickX

Projenizin çabucak bitmesini istiyorsanız, usta yazılımcılara görev verin.

Kimin usta yazılımcı olduğu yazdığı koddan anlaşılır. Roman gibi yazar, Fadime deyze bile okuyup, anlar.

CRUD yapan ile roket uçuran yazılımcı arasındaki fark nedir? Hiç bir fark yok! Çalıştığın ortam neyse, sen o sun.

Yazılımcı için boş bir editor tual, klavyesi fırça, hayal gücü paletindeki renklerdir.

Davinci gibi her yazılımcı kod yazmadan önce eserine ismini yazmalı. O zaman bilir ki altında imzası olan bir yapıtı baştan savma yapamaz.

*Projeye illa kendi imzalarını atmaya çalışan yazılımcılar bu işin bir ekip işi olduğunu unutuyorlar. #bencilyazılımcı *

Bir yazılımcı ekibi ekip lideri olmadan belki işler ama her şeye hakim gayriresmi lideri/liderleri olmadan işlemez. #bilgeprogramcı

Marifet ustadan öğrenebilmek için senior bile olunsa tüm bilgi, tecrübe ve egoyu bir kenara koyup, tekrar çırak olabilmektir.

Yazılımcının ustadan öğrenmesini engelleyen en büyük düşmanı kendi egosudur.

İşin en trajikomedik tarafı her fırsatta agile kelimesini ağzından düşürmeyenlerin bir tanecik bile unit test yazmamaları. Altta db varmış!

Bazıları için kodun herkesin malı olduğu düşüncesi kendi kodları değişikliğe uğradığında anında geçerliliğini yitiriyor. #kodbabanınmalıdeğil

Kapitalizmin rüyası gerçekleşmedi. Program yazmayı fabrikada üretim yapmaya dönüştüremediler. #codegeneretionhikaye

Java bilen işsiz kalamaz. Şimdinin Java kodu 30 sene sonrasının şimdiki Cobol'u.

Kod birimlerini tekrar kullanamamak (component reuse) gayri safi milli hasılayı aşağı çeker. #sorumluluklarımızıbilelim

Bir yazılımcının itibarını artıran en önemli şey nedir? Müşterisine çalışır bir sürüm sunması. Bunu devamlı yapabilen kazanır, elması kızarır.

Neden bir Java'cı try/finally ile kullandığı bir kaynağı kapatmıyor da, evinde uyumaya gitmeden önce gaz vanasını kapatıyor?

Usta yazılımcı birim test koduna uvey evlat muamelesi yapmaz.

Projede 50bin satırlık kod yazılmış ama birim testi yok. Bu proje nedir? Saatli bomba... tık...tık... tık...tık...dagau...

Yazılımcının en iyi dostu Sonar. Dost (Sonar) acı söyler.

*Allahım beni baştan yarat ama ne olur tekrar yazılımcı olayım ;-) *

Gerçek ustada ego olmaz, hizmet aşkı olur.

Yazılım projelerinin başarısı insan gücüne değil, ekip elemanlarının ustalık derecesine bağlıdır.

Yazılım ekibini siz kurmayın, usta bir yazılımcı kursun. Kimin işin ustası olduğunu en iyi o bilir. Laf olsun, torba dolsun diye eleman aramaz.

Yazılımcıyı tartma birimi nedir? İtibar. İtibarının kaynağı nedir? Başarıyla sürümlediği projeleri.

Bir yazılımcının itibarini artıran en önemli şey nedir? Müşterisine çalışır bir sürüm sunması. Bunu devamlı yapabilen kazanır,elması kızarır.

Usta ne diyorsa dikkatle takip edeceksin, bir kitabı tavsiye ediyorsa okuyacaksın. Paylaşım yapıyorsa bir bildiği vardır http://goo.gl/iR7Cs

Öğrenim sahanız (learning zone) hareket etmiyorsa, kişisel gelişiminiz yerinde sayıyor demektir http://goo.gl/5Zq3l

İyi kod yazmak anadil hakimiyeti ile doğrudan orantılıdır.

Dao katmanı gereksiz. #activerecordkullanın

Bir şahsın yazılımcı olmaktan mutlu olmadığını nasıl anlarsınız? Ya mimar ya da yönetici olmak ister.

Mimarların çoğu yazılımcı değildir, yazılımcıların çoğu mimardır.

Ne zaman iyi bir yazılımcı oluruz? Yazılımcılığın çok uzun bir yol olduğunu kabul edip, bu yola düştüğümüzde. Amaç yolun kendisi.

Ebeveynin odak merkezi çocuğu, yazılımcının odak merkezi kişisel gelişimi...

Ne güzel... Twitter hep DRY (Dont Repeat Yourself) uyumlu yazmaya zorluyor. Demek ki her Java sınıfı 140 harften oluşmalı.

Bilgiye kedinin fareye baktığı gibi bakmak...

Yazılımcı bilgi ile balon gibi şişer, ayakları yerden kesilir. Önemli olan bilgiyi paylaşarak balonun havasını almak ve tekrar yere basabilmektir.

Blog yazmanın önündeki en büyük engel mikroblog yazmaktır.

Bilgiyi unutmak çok kolay. Beyne kazımak için devamlı pratik yapmak gerekiyor.

Bir işin temelini anlamadan tepesini anlayamamak...

Bir ipte iki cambaz, bir kod biriminde birden fazla sorumluluk olmaz.

Birden fazla sorumluluğu olan bir sınıfa ne denir? Şizofrenik ve çok kişilikli sınıf.

Keşke bazı programcıların kod yazarken düşüncelerini de kaydeden bir karakutu olsa. Program düşünce karakutuyu dinlerdik.

Eğer bir metot bir sınıf değişkeni ya da bir sınıf metodunu kullanmıyorsa, bu metodun bu sınıfta ne işi var? #srp

En zahmetsiz edinilen meslek? Yazılımcılık. En zor kavranan meslek? Yazılımcılık

Yazılımcı olarak ne zaman emekli olurum? Öldüğümde!

BizimAlem.com - Bir Sistemin Tasarlanış Hikayesi

Giriş

BizimAlem.com 2001 ocak ayında başlamış olduğum bir web projesi. Amacım, Avrupa'da yaşayan 5 milyondan fazla Türk kökenli vatandaşımız için bir araya gelebilecekleri bir sanal ortam oluşturmaktı. Bu doküman çok basit koşullarda başlanılan bir web projesinin hangi boyutlara ulaşabileceğini ve zaman içinde yapılan eklemelerle oluşturulan teknik mimarinin hangi evrelerden geçtiğini ihtiva etmektedir. BizimAlem.com zaman içinde Türkiye ile Avrupa arasında bir sanal köprü olmayı başarmış ender web projelerinden birisidir.

İsterseniz bu proje hakkında fikir edinebilmeniz için bazı teknik bilgiler aktarayım. BizimAlem hakkında bugün (14.1.2009) itibariyle bazı teknik veriler şu şekildedir:

- 550.000 kayıtlı üye,
- Günlük ortalama yeni kayıt olan üye adedi 1000-1500,
- 40 a yakın sunucu, switch, loadbalancer, firewall sistemleri,
- Tamemen J2EE (Java 1.5) ve Open Source tabanlı,
- Tüm sunucular Linux işletim sistemiyle çalışıyor,
- Günlük ziyaretçi sayısı 55.000 – 60.000 civarında,
- Günlük sayfa gösterim adedi 1.1 milyon civarında.

BizimAlem için oluşturduğum yazılım sistemi şu komponentlerden oluşuyor:

- Forum,
- Blog,
- Okey, Tavla, Batak, Bilardo gibi multiplayer oyunlar,
- Youtube vari video modülü,
- Gruplar,
- Haberler,
- Anketler,
- Sütun,
- Arkadaş listesi,
- Kişiselleştirilebilen profil sayfaları,
- Sanal hesap,
- VIP üyelik,
- E-Card,
- Limitsiz fotograf albümleri,
- Mesaj merkezi,
- Takvim,
- Yönetim paneli (Dashboard),
- Sanal hediyeler,
- Shop,
- Favori üye listesi,
- Ziyaretçi listesi,
- Online üyeler için alt panel mesaj gönderme,
- Detaylı arama modülü,
- Yorumlar,
- vb. diğer modüller

Bu teknik bilgiler BizimAlem hakkında bir fikir sahibi olmanızı kolaylaştıracaktır. İsterseniz gelin, bu projenin başladığı ilk güne, 2001 senesinin mart ayına dönelim ve projenin hangi şartlarda start aldığını görelim.

İlk Versiyon (v.1.0)

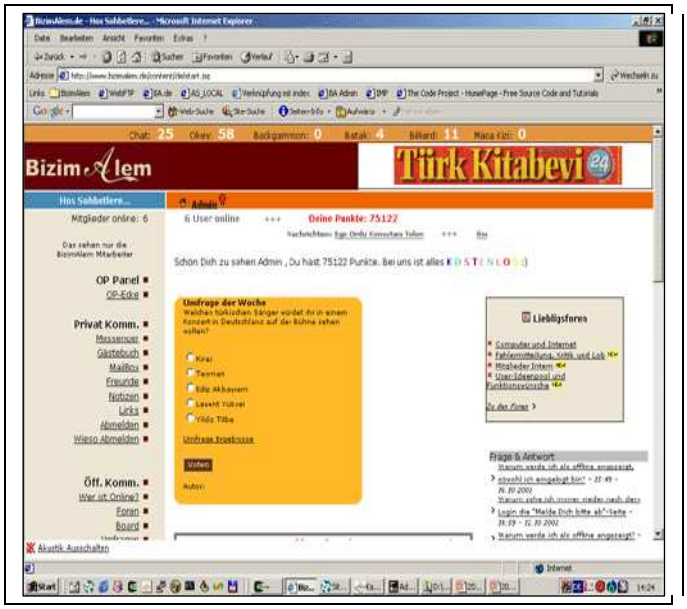
BizimAlem.com için çalışmalara Mart 2001 de başladım. Projenin ilk web adresi BizimAlem.de idi, çünkü bu web platformunu Almanya'da yaşayan Türk vatandaşları için planlamıştım. Bu sebepten dolayı BizimAlem'in ilk versiyonunda yer alan arayüzler Almanca dilinde idi. Almanya'da yaşayan genç Türkler Almanca ve Türkçe'yi hemen hemen aynı seviyede konuşabildiklerinden, BizimAlem'in ilk versiyonun da tamamen Almanca olması bir sorun teşkil etmedi. Platformun Almanca olmasının başka bir sebebi daha vardı. Almanya'da yaşayan 2.5 milyon Türk'ün yanısıra 80 milyonluk Alman toplumu vardı ve bir şekilde onlara da bu platform üzerinden ulaşmak istiyordum. Bu gibi nedenlerden dolayı ilk versiyon tamamen Almanca oldu.

1999 senesinden beri Java platformunu kullanarak freelancer olarak web projelerinde çalışma fırsatı buldum. Java'nın neler sağladığını yakından görme fırsatım olduğu için, Bizimalem'i tamamen Java platformu yardımıyla oluşturmaya karar verdim. 2000 senesinden sonra her yerde kullanılmaya başlanan Java EJB [1] teknolojisi BizimAlem için ideal gibi görünüyordu. Tabii daha sonralar acı tecrübeler yaparak ögreneceğim birşey vardı, oda bir web projesinin kesinlikle EJB gibi o tarihte henüz ermemiş bir teknoloji ile yapılmaması gerektiğiydi. Ne yazık ki çok ideal koşullardan yola çıkarak, programcı olarak tecrübesiz olmamın da etkisiyle yanlış bir teknolojiyi kullanmaya karar verdiğimi çok daha sonralar, birinci versiyonu tamamen çöpe atıp, herşeyi sil baştan yeniden yapmak zorunda kaldığımda anlamış oldum. EJB kullandığım ilk versiyon her açıdan performans problemleri ile karşılaştığım versiyondur. Bu ilk versiyon aslında gözümü yıldırmadı değil. Lakin daha iyi bir sistem ortaya koyabileceğimi düşünmem, devam etmemi kolaylaştırdı.

21.11.2001 tarihinde ilk BizimAlem versiyonu online oldu. Test sisteminde oluşturduğum ilk logo aşağıda yer almaktadır. Bu logo yerini kısa bir zaman sonra resim 2 de yer alan logoya bıraktı.

Resim 1 İlk BizimAlem logosu

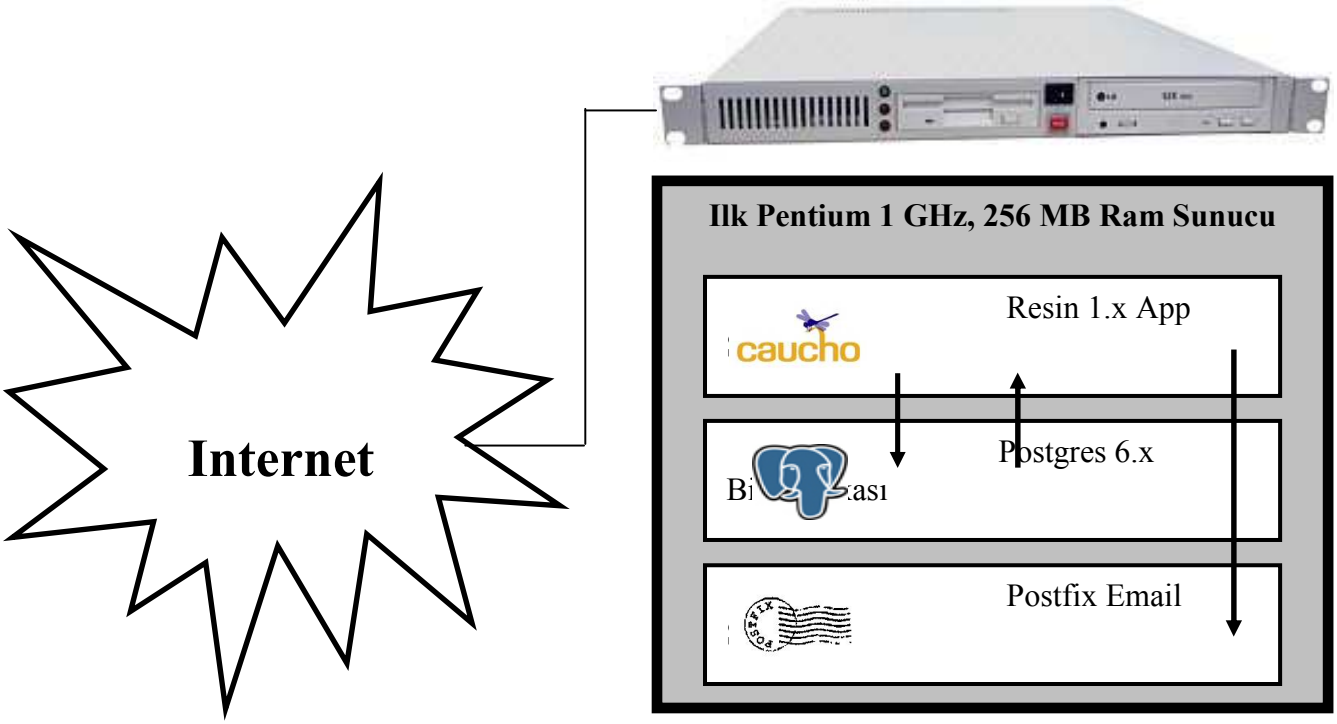
[1] Bakınız: http://java.sun.com/products/ejb/



Resim 2 İlk tasarım

Resim 2 de BizimAlem'in ilk web tasarımı yer almaktadır. İlk versiyonu oluşturmak için kullandığım teknolojik öğeler şöyledir:

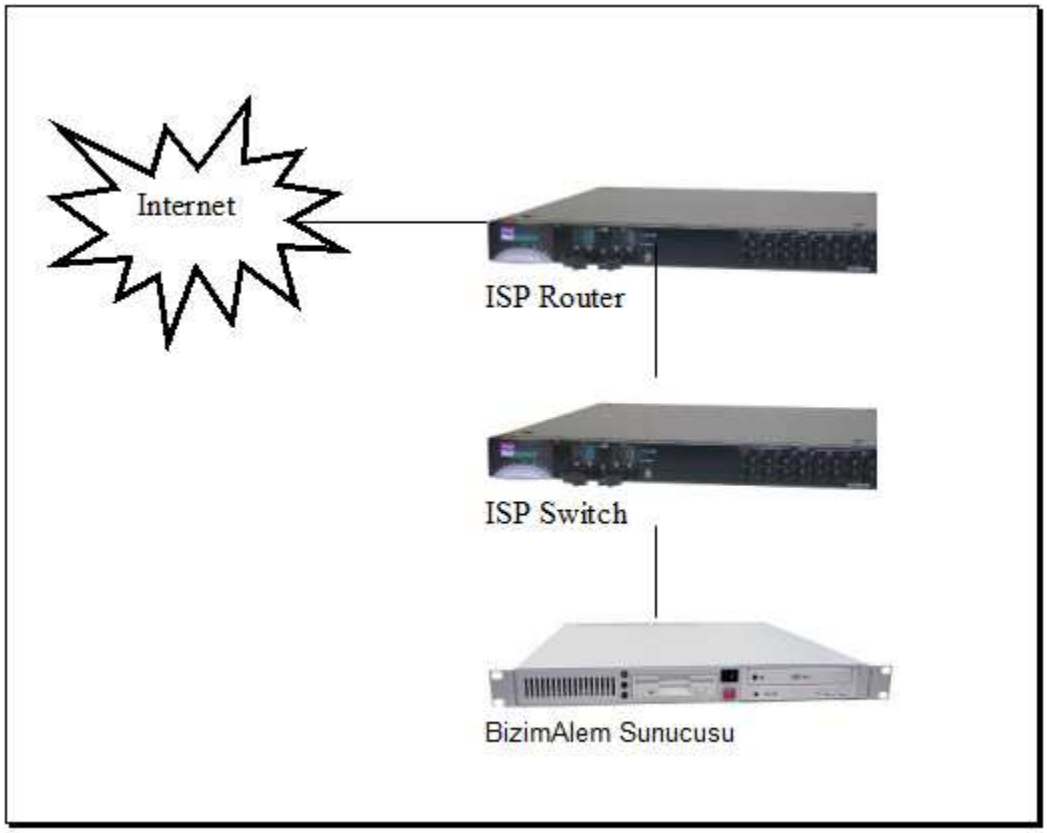
- Java 1.4
- EJB 1.1
- JBoss / Tomcat App Server
- Postres 6.x bilgibankası
- Suse Linux 6.x işletim sistemi
- Postfix email sunucu



Resim 3 İlk versiyonda kullanılan sunucu

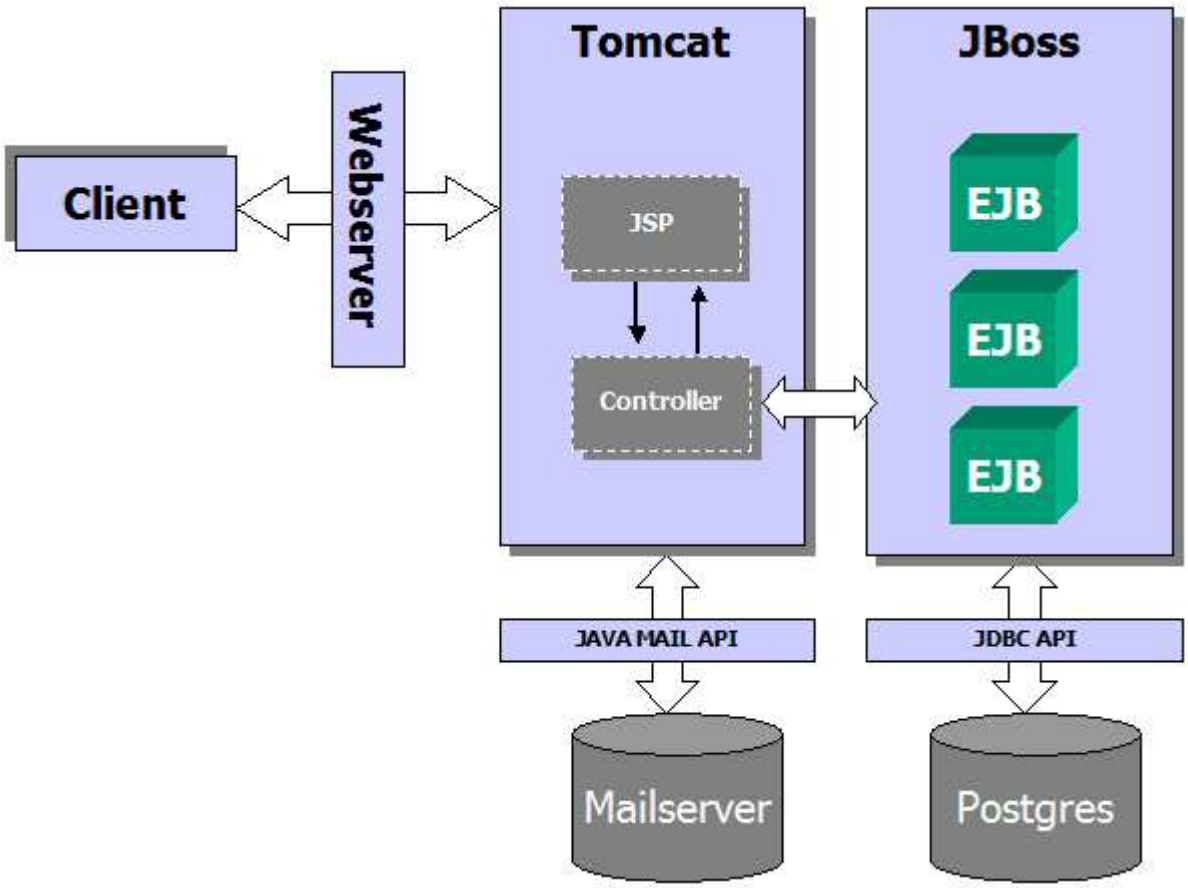
BizimAlem'in ilk versiyonunda kullandığım sunucu resim 3 ve 4 de yer almaktadır. İlk versiyon için kullandığım sunucuyu gerekli bilgisayar bileşenlerini satın alarak bir araya getirmiştim. Yazılımcı olmama rağmen, bu benim ileride server sistemlerini ve bileşenlerini daha iyi anlamamı kolaylaştırdı. Bu ayrıca benim iki şapkayı birden taşımamı sağladı. Bir şapkayı taktığımda yazılım mühendisi oluyordum ve sistem için gerekli yazılımı yapıyordum, diğer şapkayı taktığımda bilgisayar ağı ve server bileşenlerinden anlayan bir mühendis oluyor ve sunucu ve ağ üzerinde gerekli işlemleri uyguluyordum. BizimAlem aslında benim için kocaman bir laboratuar haline dönüşmüştü ve çok değişik disiplinlerde çalışma imkanı buluyordum. Bu makeleyi yazdığımda sahip olduğum teknik bilginin temelleri BizimAlem laboratuarında değişik konularda edindiğim tecrübelerle atılmış oldu. Artık sadece yazılımcı değildim. Mecbur olduğum için başka disiplinlerde de tecrübe edinmek ve oluşan sorunları çözmek zorundaydım. Şimdi geriye baktığımda, BizimAlem sayesinde çok değişik tecrübeler edinme firsatı bulduğumu görüyorum ve bunun için minnettarım.

Resim 4 İlk versiyonda kullanılan sunucu. Hala test sunucusu olarak kullanımda.



Resim 5 BizimAlem için kullanılan sunucunun internet bağlantısı

Doğal olarak bir sunucuyu internete bağlamak yeterli değil. Bunun yanısıra sistem yazılımının tamamlanmış olması gerekiyor. İlk versiyon için oluşturduğum mimari resim 6 de yer almaktadır.



Resim 6 BizimAlem teknik yazılım mimarisi

Şimdi geriye baktığımda EJB teknolojisini kullanmamı şu şekilde açıklayabiliyorum: EJB yeni oluşmuş bir teknolojiydi ve herkes herhangi bir şekilde kullanıyordu ya da kullanmak istiyordu. Doğal olarak sizde aynı trene binmek istiyorsunuz ve bu teknolojinin getirisini ve götürüsünü bilmeden kullanmaya başlıyorsunuz. Ne yazik ki benim EJB serüvenim o zamanlar hüsranla bitti, çünkü yüzlerce insanın aynı anda kullandığı bir sistemde EJB 1.1 tarzı aplikasyonlar ne yazık ki performans sorunları doğuruyor! Yaşadığım performans sorunlarını şu şekilde sıralayabilirim:

- Bir EJB komponent (1.x versiyonunda) ile sadece Remote Interface üzerinden iletişim kurulduğu için, bu Tomcat içinde bulunan JSP / Controller sınıfları ile JBoss içinde bulunan EJB komponentlerin bağlantı ve işlem süresini uzattı.
- Yüzlerce insanın online olduğu bir sistemde EJB komponentleri yetersiz kalmaya başladı. Pooling mekanizmaları kullanmama rağmen bir JBoss App Server içinde bulunan EJB (Stateless Session Bean) komponentler aynı anda sadece 25-30 Thread tarafından kullanılabilir oldular. Kapasiteyi yükseltmek için kullanılan server adedini artırmam gerekti.

Performans sorunlarının yanısıra yazılım esnasında da çok zorluklar çektim. EJB komponentler ne yazik ki test edilebilir yapıda değiller. Bu en azından EJB 3.0 öncesi geçerli olan bir durumdu. Test etmek ve hataları bulmak için debugging yapmak gerçekten çok sabır isteyen iştir.

Uzun bir müddet performans sorunlarıyla ugraştıktan sonra, EJB komponentlerine veda etmem gerektiğini anladım ve BizimAlem'in ikinci versiyonu için kolları sıvadım.

İkinci Versiyon (v.2.0)

2002 senesinin ortalarından itibaren BizimAlem.com v.2.0 için çalışmalara başladım. v.1.0 versiyonu benim için pekte tekrar kullanılır yapıda değildi, çünkü kodun merkezinde EJB komponentleri vardı ve ben EJB teknolojisinden tamamen uzaklaşmak istiyordum. Bunun yanısıra kodun bakımı ve geliştirilmesi kolay olmalıydı. Bu arada üye sayısı 2000 i aşmıştı ve aynı anda 100 ün üzerinde üye online oluyordu. Yeni sistemdeki performans sorunlarını ortadan kaldırmak için bir analiz yaptım. Analizin sonuçları şunları gösterdi:

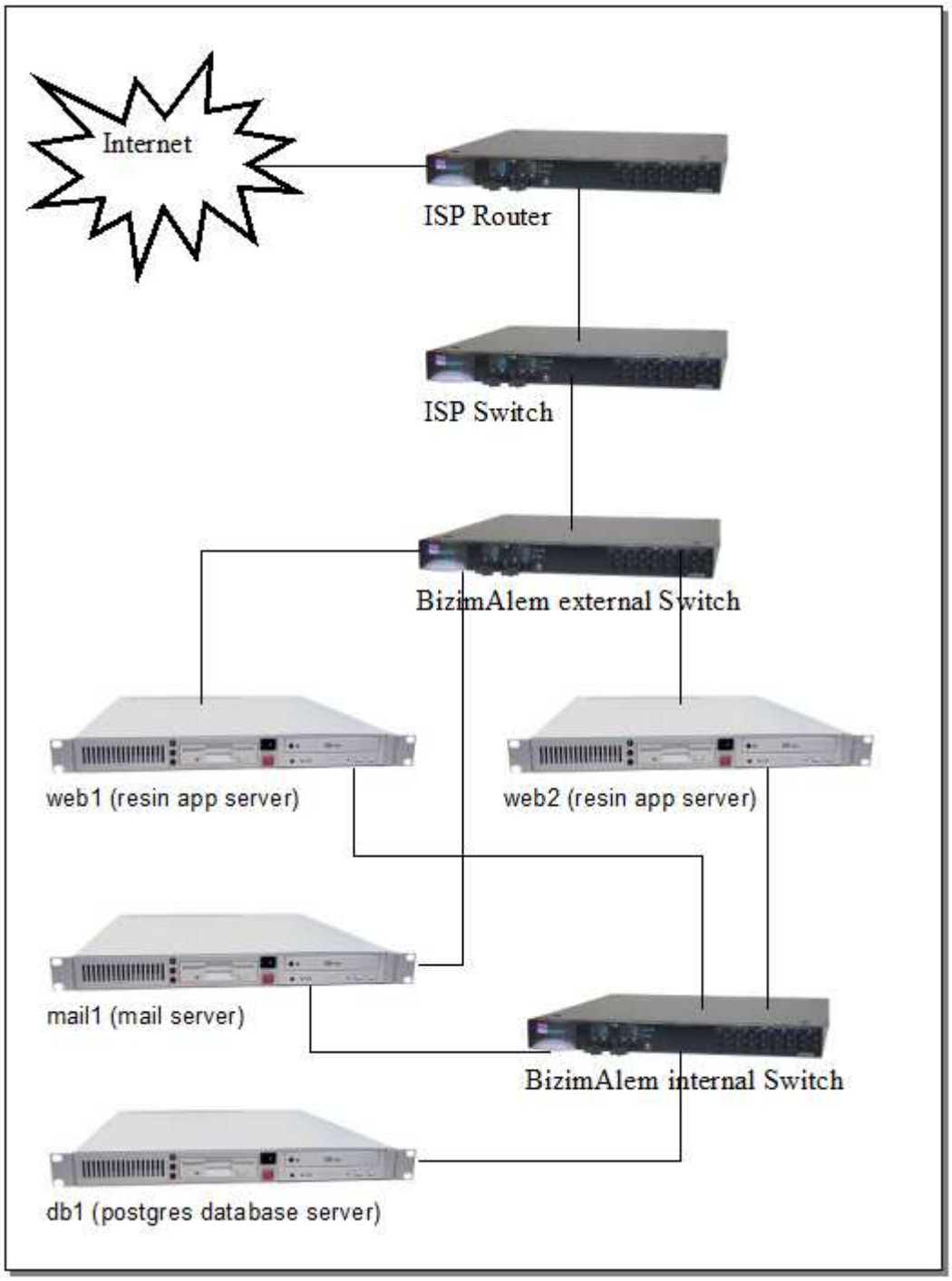
- Bilgibankası için ayrı bir sunucu kullanılması gerekiyor. Bu bilgibankası için yapılan işlemlerin performansını artırır.
- Email gönderimi için ayrı bir sunucunun kullanılması gerekiyor. Bu şekilde sistemin diger bölümlerini etkilemeden email gönderimi performansı artırılabilir.
- JBoss yerine yeni bir application server kullanılması gerekiyor, çünkü EJB kullanılmadığına göre JBoss gibi bir EJB containere olan ihtiyaç ortadan kalkmıştır. Yeni application serveri olarak Caucho Resin [2] 1.x serisini seçtim. Resin performansı yüksek olan bir Servlet / JSP containerdir, yani Tomcat gibi bir application server.
- Web aplikasyonu iki değişik Resin application server içinde paralel çalıştırabilirsem, gelen yükü bu iki server arasında paylaştırabilirim. Bu sebepten dolayı web aplikasyonu (v.2.0) için en az iki yeni sunucuya ihtiyacım var. Bu şekilde online olan üyelerin adedi 500 e kadar yükselebilir. Sunucu kapasitesi arttıkça, hizmet verilebilecek online üye adedi de artar.

İlk versiyonda sadece bir sunucu ile yetinirken, artan üye sayısıyla beraber genel sistem performansını yükseltebilmek için birden fazla ve sorumluluk alanları tanımlanmış olan sunuculara ihtiyacım olduğunu anladım. Artık BizimAlem projesi kendi bilgisayar ağını kurma aşamasına gelmişti. Birden fazla IP adresine ve çok ufak bir network adres alanına ihtiyacım vardı. Müşterisi olduğum ISP bana kendi C class network adres alanından aşağıdaki IP adreslerini tahsis etti.

- 213.221.93.10 – web1 isimli sunucu.
- 213.221.93.11 – web2 isimli sunucu
- 213.221.93.12 – mail1 isimli sunucu
- 213.221.93.1 – default route
- 213.221.9.255 – default netmask

Bilgibankasını da kendine özel bir sunucu üzerinde çalıştırdığım taktirde, genel sistem performansı çok daha iyi olabilecekti. Bu yüzden gerekli serverleri satın aldıktan sonra, aşağıda yer alan ağ planını oluşturdum.

[2] Bakınız: http://www.caucho.com



Resim 7 BizimAlem v.2.0 ağ planı

Resim 7 de görüldüğü gibi iki yeni ağ oluşturmam gerekiyordu, çünkü bilgibankasını dış dünyaya açmak istemiyordum. Bu yüzden her sunucuya iki ağ kartı takarak biri üzerinden dış ağa (BizimAlem external Switch) diğeri üzerinden bilgibankasının bulunduğu iç ağa (BizimAlem internal Switch) bağlanmalarını sağladım. web1 ve web2 sunucuları iç ağ üzerinden bilgibankası işlemlerini gerçekleştiriyorlar. İç ağ için aşağıdaki IP şemasını kullandım:

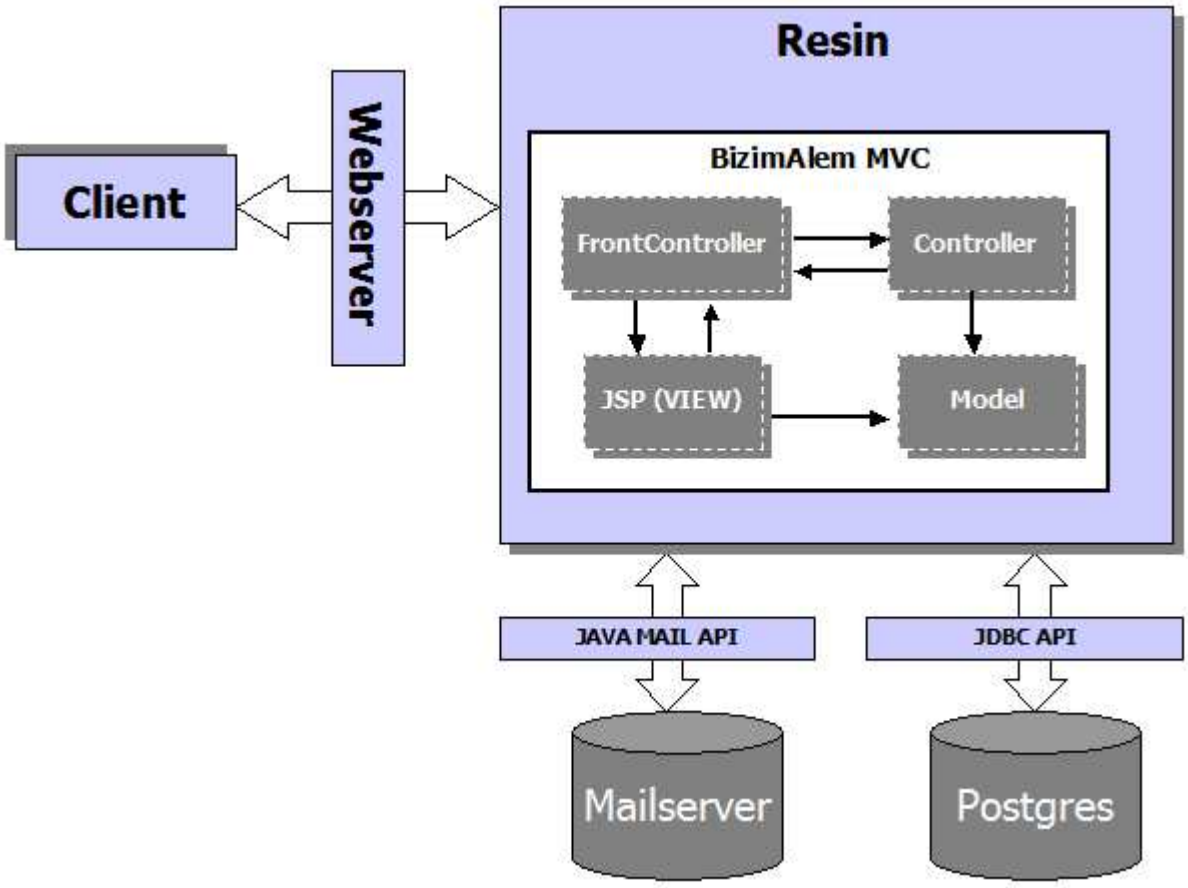
- IP Range: 192.168.1.1 – 213.221.93.255
- Networkmask: 192.168.1.255
- Default gateway: 192.168.1.1

Bu şemaya göre sunucuların sahip oldukları IP adresleri şu şekilde oldu:

- web1 dış ağ IP adresi: 213.221.93.10
- web1 iç ağ IP adresi: 192.168.1.10
- web2 dış ağ IP adresi: 213.221.93.11
- web2 iç ağ IP adresi: 192.168.1.11
- mail1 dış ağ IP adresi: 213.221.93.12
- mail1 iç ağ IP adresi: 192.168.1.12
- db1 iç ağ IP adresi: 192.168.1.2

BizimAlem v.2.0 için dört yeni sunucu (server) ve iki Switch kullanıldı. Bu şekilde network komponent adedi altıya çıkmış oldu.

Ağ oluşturma çalışmaları yanısıra v.2.0 için yazılım çalışmalarını da devam ettiriyordum. Kısa bir zaman sonra aşağıdaki mimari yapı oluştu.



Resim 8 BizimAlem v.2.0 yazılım mimarisi

BizimAlem v.2.0 ile yeni bir yazılım konsepti ile tanıştı. Web projelerinde yazılımı kolaylaştırmak için genelde Model View Controller (MVC) tasarım şablonu kullanılır. MVC tarzı çalışan web frameworklerde kullanıcı istekleri merkezi bir Controller (FrontController) sınıfı tarafından karşılanır. Controller sınıfı validasyon ve navigasyon gibi işlemlerden sorumludur. JSP sayfalarında gösterilmesi gereken veriler Model sınıflarında tutulur. Controller, verileri ihtiva eden model nesnelerini HttpServletRequet, HttpServletResponse ya

da HttpSession (Java Servlet API sınıfları) sınıfları aracılığıyla JSP sayfaları tarafından kullanılır hale getirir. Modellerin ihtiva ettiği veriler View olarak isimlendirilen JSP sayfalarında gösterilir. JSP sayfaları HttpServletRequest, HttpServletResponse ya da HttpSession aracılığıyla gerekli model nesnelerine ulaşır. Çoğu web framework JSTL taglerini kullanarak, edinilen verilerin JSP sayfalarında gösterimini gerçekleştirir. JSP sayfalarının model nesnelerine erişimi frameworkün implementasyon tarzına göre değişen bir durumdur.

Ben o zamanlar (2002) Struts[3] gibi piyasada bulunan bir MVC web framework kullanmak yerine kendi MVC frameworkünü implemente etmeyi uygun gördüm. Bu çalışmaların sonunda Struts gibi çalışabilen BizimAlem MVC web frameworkü oluştu. Çok basit bir şekilde implemente ettiğim BizimAlem MVC web framework validasyon ve navigasyon gibi işlemleri kolaylıkla yapabilecek yapıdaydı. Kullandığım Controller sınıfları basit POJO (Plain Old Java Object) tarzı sınıflardı. Artık v.2.0 bünyesinde hiçbir EJB komponenti kullanılmıyordu ve JBoss yerini Resin application servere bırakmıştı. web1 ve web2 isimlerinde iki Resin application server üzerinden BizimAlem'in iki kopyası paralel olarak çalışmaktaydı ve kullandığım BizimAlem external Switch üzerinden gelen kullanıcıları web1 ya da web2 ye otomatik olarak yönlendirebiliyordum. Bu şekilde sistem kapasitesini, gerekli olduğu durumlarda örneğin web3 isminde yeni bir sunucu ekleyerek artırabilecektim. Bu ileride Hardware Loadbalancer komponentleri kullanarak çok geniş bir kullanıcı kitlesine hitap edebilecek bir altyapının temellerini oluşturdu. Yazılım esnasında da, oluşan sistemin birden fazla server üzerinde paralel çalışabilmesine dikkat ettim. Oluşan sistem bulunduğu servere bağımlılık oluşturmadan çalışabiliyordu ve bu şekilde yeni sunucular ekleyerek sistem kapasitesini artırmak kolaylaşmıştı.

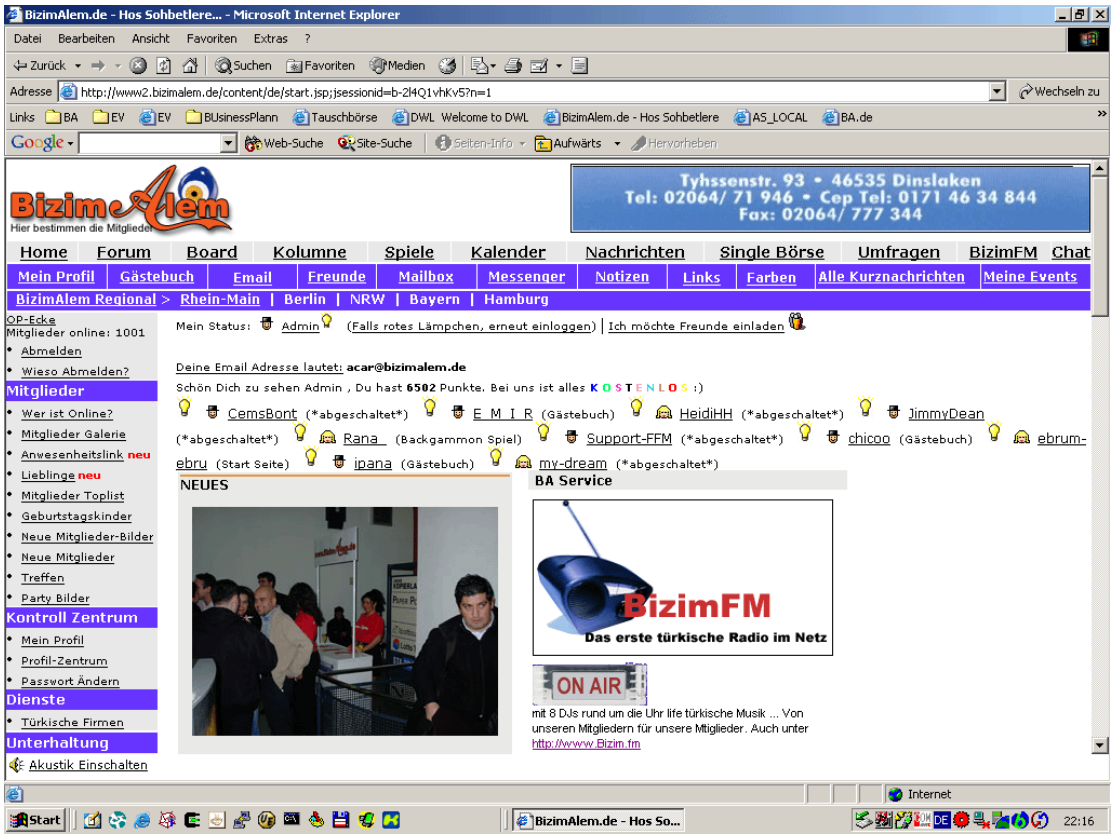
BizimAlem v.2.0 ile v.3.0 arasındaki versiyonlarda kullandığım bazı BizimAlem logoları ve tasarım çalışmaları aşağıda yer almaktadır.

Resim 9 Tarihi bir BizimAlem logosu

Resim 10 Tarihi bir BizimAlem logosu

[3] Bakınız: http://struts.apache.org/

Resim 11 Tarihi bir BizimAlem logosu



Resim 12 Tarihi bir BizimAlem tasarımı

Resim 13 Tarihi bir BizimAlem tasarımı

BizimAlem v.2.0 2002 sonunda online oldu ve performans sorunlarını büyük ölçüde giderdi. Kısa bir zaman sonra kayıtlı üye sayısı 40.000 i, Aynı anda online olan üye sayısı 1000 i geçti. Herşey yolunda gidiyor gibi görünüyordu ve platform günden güne devamlı büyüyerek, sahip olduğu altyapı kapasitesinin sınırlarını zorlamaya başladı.

Sistemin güvenliğini sağlamak için her sunucu üzerinde Linux Firewall sistemini aktive etmem ve bakımını yapmam gerekiyordu. Bu zaman içinde sunucu sayısı arttıkça zorlaşan bir işlem haline geldi. Merkezi bir firewall sisteminin gerekli olduğunu kısa bir zaman sonra BizimAlem Hackerlerin hedefi haline geldiğinde anlamış olacaktım.

Ne yazik ki ağ ve işletim sistemi yönetiminde tecrübesiz olduğum için, düzenli aralıklarla Linux işletim sistemi için gerekli yamaların (patch) yapılması gerektiğini çok geç öğrendim. Bir takım niyetleri iyi olmayan şahışların BizimAlem serverlerini ellerine geçirmelerine mani olamadım ve bir müddet bu şahıslarla beraber (co-existing) yaşamak zorunda kaldık. Bu şahıslar serverleri ellerine geçirdikleri için, sistemi istedikleri şekilde kullanabiliyorlardı. Bunun bu şekilde gitmeyeceğini ve BizimAlem için yeni bir güvenlik konsepti geliştirmem gerektiğini anladım ve çalışmalara başladım.

Öncelikle bilgisayar ağlarında bulunan sunucuların ele geçirilmeleri hakkında kitaplar okudum ve gerekli bilgiyi edindim. Eğer saldırı yöntemlerini bilmesseniz, sistemi nasıl korumanız gerektiği hakkında pek fazla bir bilginiz olmaz. Bu yüzden kendinizi saldırıcı şahıslar yerine koyarak, nasıl çalıştıklarını anlamanız gerekiyor. Bu şekilde sistemin açıklarını da keşfederek, kısa zamanda gerekli tedbirleri almanız mümkün olur.

Altı aya varan „Hacker nasıl olunur“, - „Sunucular hackerlere karşı nasıl korunur“ eğitimimden sonra BizimAlem güvenlik konsepti üzerinde çalışmaya başladım.

Bu yazının başında da belirttiğim gibi BizimAlem benim için büyük bir ihtisas alanı haline geldi. Normal şartlarda yazılımcı olarak ilgilenmeyeceğim konular benim için faaliyet alanı

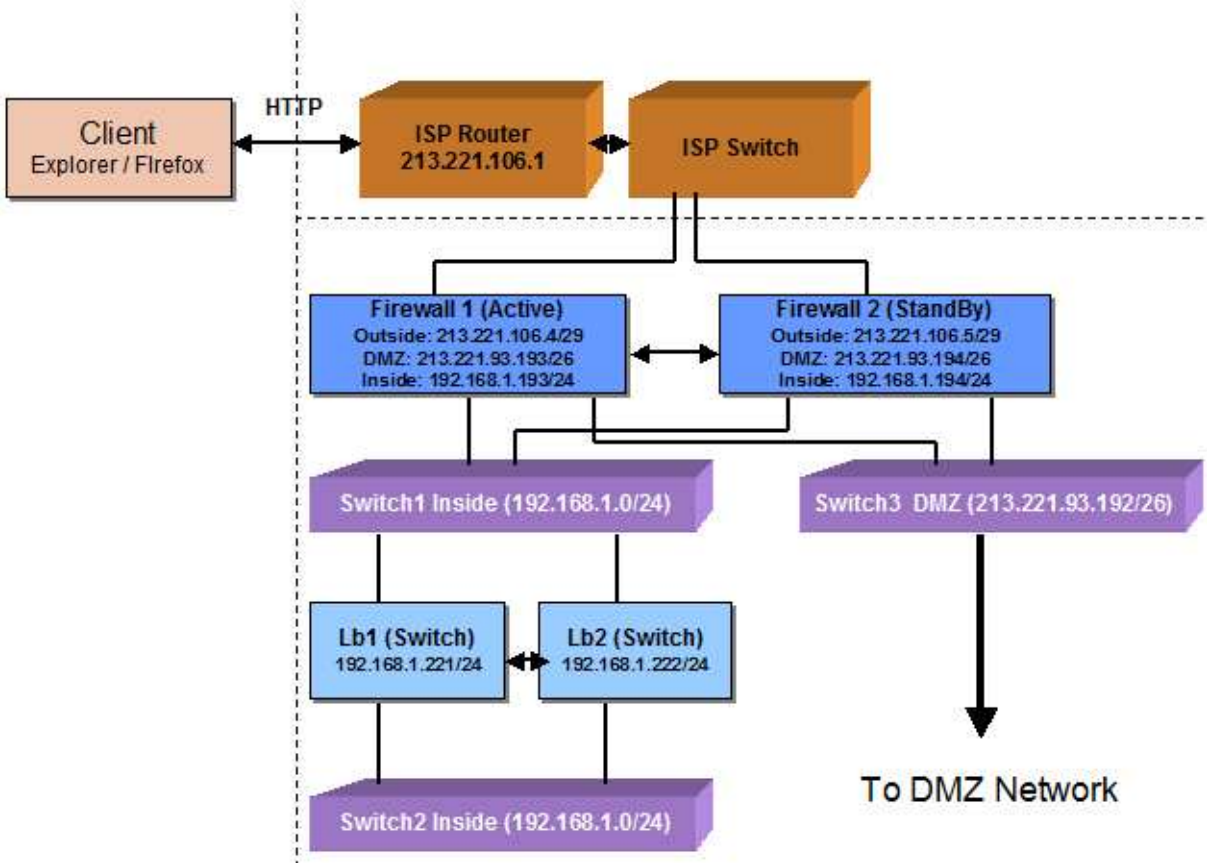
olmaya başladı, çünkü oluşan her türlü sorunu kendi başına çözmek zorundaydım. Aslında bunu da çok severek yapıyordum, çünkü bu çok zevkli bir uğraştı.

Yeni güvenlik konseptim aşağıdaki maddelerden oluşmaktaydı:

- Güçlü bir merkezi firewall sistemi ile tüm sistemin korunması gerekli. Her sunucunun tek başına lokal bir firewall sistemi tarafından korunması yeterli değil.
- İşletim sistemi düzenli olarak yamalanarak (patch) en aktüel hale getirilmek zorunda. Aksi taktirde üçüncü şahışlar sistem açıklarını kullanarak, sistemi sabote edeceklerdir.
- Kullanılan open source komponentlerin en son versiyonları serverlere kurulmalı. Eski versiyonlarda bulunan açıklar üçüncü şahışlar tarafından kullanılarak sistem sabote edilebilir.

İlk işlem olarak 25.000 EUR civarında yatırım gerektiren ve yüksek direnme ve koruma gücüne sahip firewall komponentlerini satın aldım. Firewall sistemi iki sunucudan oluşuyordu ve beraber devamlı çalışma özelliğine sahip bir ikili oluşturuyorlardı. Heart Beat olarak bilinen bir mekanizma ile firewall sistemini oluşturan komponentler birbirlerine bağlıdır ve hayatta olup olmadıklarını karşılıklı olarak kontrol ederler. Eğer aktif olan firewall komponenti devre dışı kalırsa, diğer komponent onun yerine geçerek, işlemlerin devam etmesini yani trafiğin akmasını sağlar.

Yeni güvenlik konsepti dogrultusunda BizimAlem için gerekli ağı yeniden yapılandırdım. Bir sonraki resimde yeni ağ planı yer almaktadır.



Resim 14 BizimAlem ağ planı

Yeni ağ planına göre, sistemin tümüne erişim yeni firewall sistemi üzerinden gerçekleşmekteydi. Tüm trafik firewall komponentlerinden transit geçiş yaptığı sürece, sistemin güvenliğini optimal sağlayabilirdim.

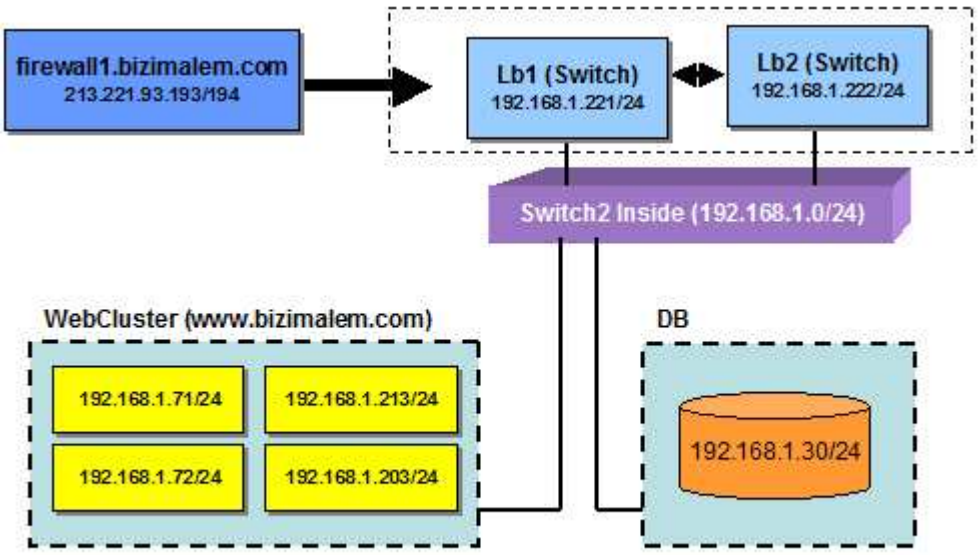
Yeni sistemi oluşturabilmek için bulunduğum ISP (Internet Service Provider) ile fikir alışverişinde bulunmaya başladım. Doğal olarak artık kendi kullanabileceğim bir alt ağa (subnetwork) ihtiyacım vardı ve ISP bana gerekli IP adres alanını tahsis etmek durumundaydı. Görüşmeler sonunda aşağıda yer alan IP adres alanı bana tahsis edildi.

- IP Range: 213.221.93.192 – 213.221.93.255
- Netmask: 213.221.93.192
- Default Gateway: 213.221.93.193
- DNS: 212.82.225.7

Bana tahsis edilen IP alanında 61 IP adresini kullanabiliyorum. Bunlar 213.221.93.193 ila 213.221.93.254 arasında olan IP adresleri. 213.221.93.193 nolu IP adresi firewall sisteminin bir ayağını oluşturuyor. Bu yüzden firewall arkasında kalan tüm serverler için 213.221.93.193 default gatewaydir.

Sunucular arasındaki yükü eşit bir şekilde dağıtabilmek için hardware loadbalancer komponentleri satın alarak sisteme dahil ettim (resimde LB1, LB2). Firewall sisteminde olduğu gibi Loadbalancer sistemi iki komponentten oluşuyor. Heart Beat mekanizmasıyla bu komponentler birbirlerine bağlı. LB1 devre dışı kalması durumda LB2 onun yerine geçerek, trafiğin akmasını sağlar.

Üye sayısı artıkça sistemin kapasitesini de artırmak gerekti. Bu yüzden yeni sunucular satın alarak bir web cluster oluşturmaya karar verdim. Cluster birden fazla sunucunun içinde bulundugu server topluluğudur. Bu sunucular aynı görevi yapmak ve oluşan yükü paylaşmak için berabar çalışırlar.



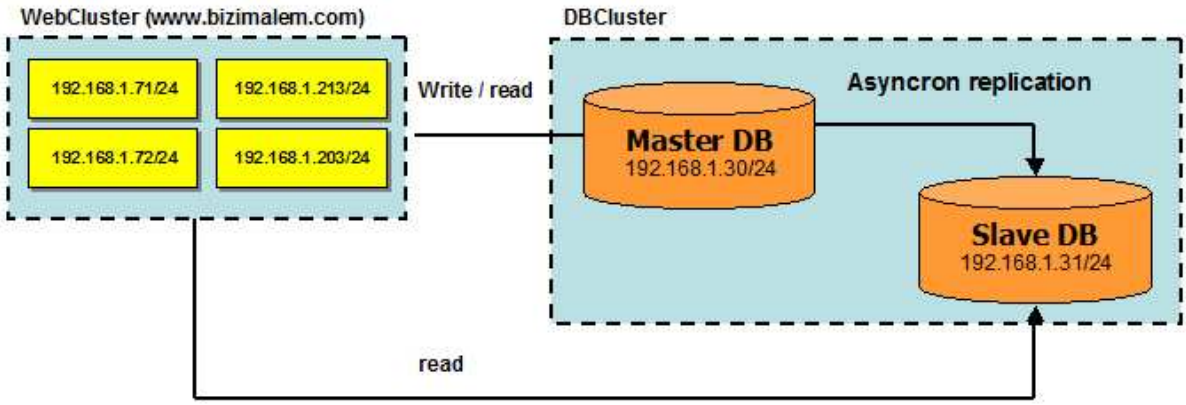
Resim 15 BizimAlem ağ planı

web1 (192.168.1.71), web2 (192.168.1.72), web3 (192.168.1.213) ve web4 (192.168.1.203) ismini taşıyan bu sunucular web clusteri oluşturuyor. Artık eski versiyonda olduğu gibi bu sunucuların geçerli IP adresleri yok, yani 213.221.93.10 yerine iç ağdan olan 192.168.1.71 gibi bir IP adrese sahipler. Bu şekilde bu sunucuların direk internet üzerinden erişimleri engellenmiş oluyor. Bu serverlere sadece Loadbalancer (LB1) komponenti üzerinden erişilebilir. Bu da sadece http://www.BizimAlem.com, yani 213.221.93.231 IP numarası üzerinden mümkün. Loadbalancer 213.221.93.231 nolu IP ye gelen istekleri otomatik olarak web cluster içinde bulunan herhangi bir sunucuya iletir. Bu sunucunun geçerli bir IP adresi olmak zorunda değil. Yaptığım ağ ayarları ile LB1 ve web cluster aynı ağ içinde olduklarından (Switch2 üzerinden) birbirleriyle temas kurmaları kolaylaştı. LB1 ve LB2 üzerinde oluşturduğum 213.221.93.231 nolu VIP (virtuelle IP) üzerinden web cluster içinde bulunan tüm sunucuları sadece bir tek sunucuymuşcasına kullanmak mümkün hale geldi. Bu durumda kullanıcı sadece http://www.BizimAlem.com yazar ve LB1 tarafından web cluster içinde bulunan bir sunucuya yönlendirilir. Kullanıcı hangi server üzerinde olduğunu bilmez. LB1 cookieler yardımı ile kullanıcı isteklerinin hep aynı sunucuya iletilmesini mümkün kılar.

Geçen zaman diliminde internet ağ oluşumu ve yönetimi hakkında detaylı bilgi edinme firsatı buldum. Sene 2005 ocak ayını gösterirken BizimAlem 200.000 den fazla üyesiyle Avrupa'nın en büyük Türk platformu olma yolunda ilerliyordu. Online üye sayısı 2500 lere dayanmış ve yine doğal olarak sistem sahip olduğu kapasite sınırlarını zorlamaya başlamıştı. Bunun yanısıra sistem üzerinde aynı anda bir çok üye işlem yaptığı için kullandığım bilgibankası performans sorunları yaratmaya başlamıştı. Bu sorunu çok kısa bir zamanda ortadan kaldırmam gerekiyordu, çünkü BizimAlem.com, sistem yükünün çok artığı saatlerde erişilemez hale geliyordu. Bu durum BizimAlem v.3.0 versiyonunun hazırlık başlangıcı oldu.

Üçüncü Versiyon (v.3.0)

2006 senesine gelindiğinde BizimAlem büyük bir platform haline gelmiş, lakin büyüklüğüyle oluşan problemler de büyük boyutlara ulaşmıştı. En büyük sorunlardan birisi bilgibankasının aplikasyon için dar boğaz (bottleneck) haline gelmesiydi. Bu sorunu çözebilmek için iki bilgibankasının bir cluster sistemi oluşturduğu bir yapıyı oluşturmaya başladım. BizimAlem için açık kaynaklı olan (open source) PostgreSQL bilgibankasını kullanıyorum. Birden fazla bilgibankası sunucusundan oluşan bir cluster oluşturabilmek için bilgibankaları arasında replikasyon yöntemleri kullanarak verilerin aynı seviyede ve senkron tutulması gerekiyor. Senkron veri tutulması PostgreSQL sistemlerinde hemen hemen mümkün değil. Ama verileri asenkron replikasyon yöntemleri kullanarak cluster içinde bulunan bilgibankalarında aynı seviyede tutmak mümkün. Bunun ne anlama geldiğini BizimAlem için oluşturduğum bilgibankası cluster yapısını göstererek, açıklamaya çalışayım.

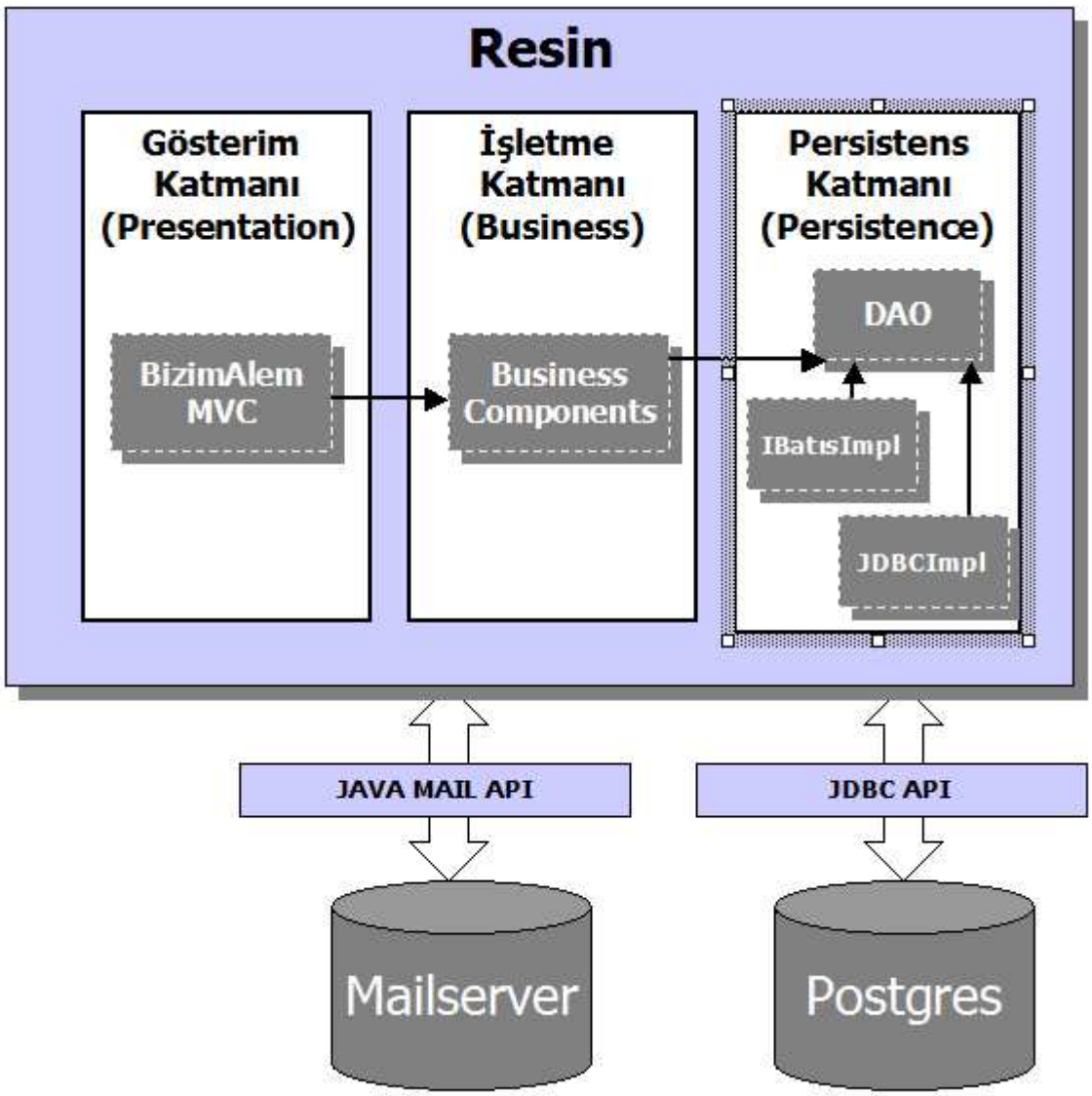


Resim 16 BizimAlem bilgibankası cluster sistemi (dbcluster)

Web cluster tarafından kullanılan ve master (usta) db ismini taşıyan bir ana bilgibankası sunucusu var. Tüm insert,update ve delete komutları master db tarafından işlem görür, yani bütün değişiklikler bu bilgibankası üzerinde yapılır. Açık kaynaklı olan Slony[4] replikasyon sistemi ile master db bünyesinde meydana gelen tüm değişiklikler slave (çırak) db ye aktarılır. Bu işlem asenkron gerçekleşir, bu yüzden bu replikasyon tarzının ismi asenkron replikasyondur. Insert ya da update yapıldıktan sonra Slony bu değişiklikleri birkaç saniye içinde lokalize eder ve otomatik olarak bu komutları slave db ye aktarır. Bu işlemin ardından master db ve slave db aynı seviyeye gelir. Web cluster insert, update ve delete komutları için master db yi, select komutları için slave db yi kullanır. Sistemde oluşan SQL komutlarının %80 den fazlası select komutları olduğu için ikinci bir db (database) nin sistem tarafından kullanılıyor olması, bilgibankası yükünü hafifletir ve bilgibankası dar boğaz olmaktan çıkar. Böyle bir db cluster sistemi ile her iki db de select komutları için kullanılabilir, lakin insert, update ve delete komutlarının devamlı master db ye gitmesi sağlanır, çünkü master db merkezi bilgibankası sistemidir. Slave1, slave2, slave3 şeklinde yeni bilgibankası sunucuları sisteme eklenerek, bilgibankası performansı daha da artırılabilir.

BizimAlem için v.3.0 oluşturma çalışmaları 2006 Eylül ayında başladı. Amaç 500.000 den fazla üyeye hizmet verebilen ve aynı anda 10.000 üyenin online olabildiği bir platform oluşturmaktı. Bunun için BizimAlem'in yazılım mimarisini değiştirmem gerekliydi. Yaptığım performans analizleri aşağıda yer alan yazılım mimarisinin gerekli olduğunu gösterdi.

[4] Bakınız: http://www.slony.info/



Resim 17 Üç katmanlı mimari

BizimAlem bünyesinde, üyelere sunulan hizmetler çoğaldıkça, kodun bakımı ve geliştirilmesi zor bir hal almaya başlamıştı. Bu sebepten dolayı üç katmanlı bir mimari oluşturarak, bu sorunu ortadan kaldırmaya çalıştım.

Üç Katmanlı Mimari Nedir?

Günümüzde yapılan kurumsal projelerin temelinde üç ya da daha fazla katmanlı mimariler yatmaktadır. Program yazılımının amacı, veri oluşturmak, bu verileri depolamak, istendiği zaman depolanmış verileri elde edip, değerlendirmek ve belirli sonuçlara ulaşmaktır. Buradaki sihirli kelime „veri“ dir ve bilgisayarın ve internetin icat edilmesinde başrolü oynamıştır.

Bir firmanın günlük faaliyetlerinde hergün birçok veri oluşur, bu veriler değerlendirilir ve firmanın bir veya birden fazla bilgibankasında depolanır. Her firmanın stratejik faaliyetlerinden birisi, bu veri oluşumunun kontrollü bir şekilde yapılmasını sağlamak olmalıdır. Veri kaybı, aynı zamanda firmanın kazanç kaybı anlamına gelebileceği için, birçok

firma, sahip oldukları verilere bilgisayar ve bu bilgisayarlarda çalışan bir takım programlar aracılığı ile sahip çıkmaktadırlar.

Program yazılım amacının veriler üzerinde işlem yapmak olduğunun altını çizdik. Peki program yazma kriterleri nelerdir? Her programcı istediği şekilde, gelen istekler doğrultusunda program yazabilir mi? Evet yazabilir, ama programcının profesyonelliği, oluşan kodun kalitesi ile direk orantılıdır. Edindiğim tecrübeler doğrultusunda, bir programın bakımı ve geliştirilmesi, yazılımından daha pahalıdır diyebilirim. Tasarım şablonlarının kullanılmadığı ve profesyonel olmayan yazılımcılar tarafından oluşturulan programların bakımı imkansız ya da çok zordur. Bir firma için sahip olduğu veriler ne kadar önemli ise, verileri işlemek için kullandığı programlar ve bu programların bakımı ve geliştirilmesi de bir o kadar önemlidir.

Günümüzde firmalar sahip oldukları verileri kullanmak, saklamak ve işlemek için web tabanlı programlar kullanmaktadırlar. Web tabanlı programların bakımı ve geliştirilmesi masaüstü programlardan daha kolay ve ulaşılan kullanıcı kitlesi daha büyük olduğu icin (kullanıcılar genelde bir web tarayıcı – browser ile çalışabilirler) tercih edilmektedirler. Web tabanlı programlar bugünkü standartlara göre üç katmandan oluşacak şekilde hazırlanır. MVC (Model – View – Controller) tasarım şablonunda da gördüğümüz gibi, sorumluluk alanları tanımlanmış katmanlar oluşturulur. Her katman kendisi için tanımlanmış görevi yerine getirmekle yükümlüdür ve işlevini yerine getirmek için diğer katmanlardan faydalanır.

Web projelerde uygulanan ilk katman gösterim (presentation) katmanıdır. Bu katmanda veriler üzerinde işlem yapılmaz. Veriler üzerinde işlem diğer katmanlar tarafından gerçekleştirilir. Gösterim katmanı başka bir katmanda hazırlanmış olan verilerin kullanıcıya gösterimi için kullanılır. Bu katmanda JSP ve Servlet gibi gösterim teknolojileri kullanılarak edinilen veriler sunulur.

Gösterim katmanına veriler işletme katmanı (business) tarafından sağlanır. İşletme katmanında veriler üzerinde yapılacak işlemler tanımlanır. Bunlar Java sınıflarında oluşturulan metodlardır. Business metodları olarak bilinen bu birimlerde, firmanın veriler üzerinde yapmak istediği işlemler implemente edilerek, istenilen neticeler elde edilir. Gösterim katmanı için gerekli veriler veri depolama / edinme (persistence) katmanı tarafından sağlanır. Bu katmanın görevi JDBC yada Hibernate gibi teknoloji ile bilgibankasında yer alan verileri edinmek ve istenilen verileri bilgibankasında depolamaktır.

Katmanlar arası komunikasyon tanımlanmış interface sınıflar üzerinden gerçekleşir. Örneğin gösterim katmanı işletme katmanında bulunan bir Facade interface üzerinden istediği verileri elde edebilir. Gösterim katmanı, işletme katmanı sadece bir interface sınıfından oluşuyormuş gibi düşünülerek, bu interface sınıfına karşı programlandığı taktirde, iki katman arasında esnek bir bağ oluşur. Bu durumda işletme katmanı, dış dünyaya sunduğu Facade interface sınıfını istekleri doğrultusunda implemente ederek gösterim katmanını etkilemeden çalışma tarzını tanımlayabilir. Facade interface sınıfında tanımlanmış metodlar değişmediği sürece, işletme katmanında yapılacak değişiklikler gösterim katmanını etkilemez. Sadece bu şekilde hazırlanmış bir program, gelecekte meydana gelen değişikliklere ayak uydurabilir yapıda olabilir. Tüm kodun bir katman içinde implemente edilmesi, kullanılan sınıflar arasındaki bağı yükselteceği gibi, bu kodun ilerideki bakımını güçleştirir.

BizimAlem Persistence Katmanı

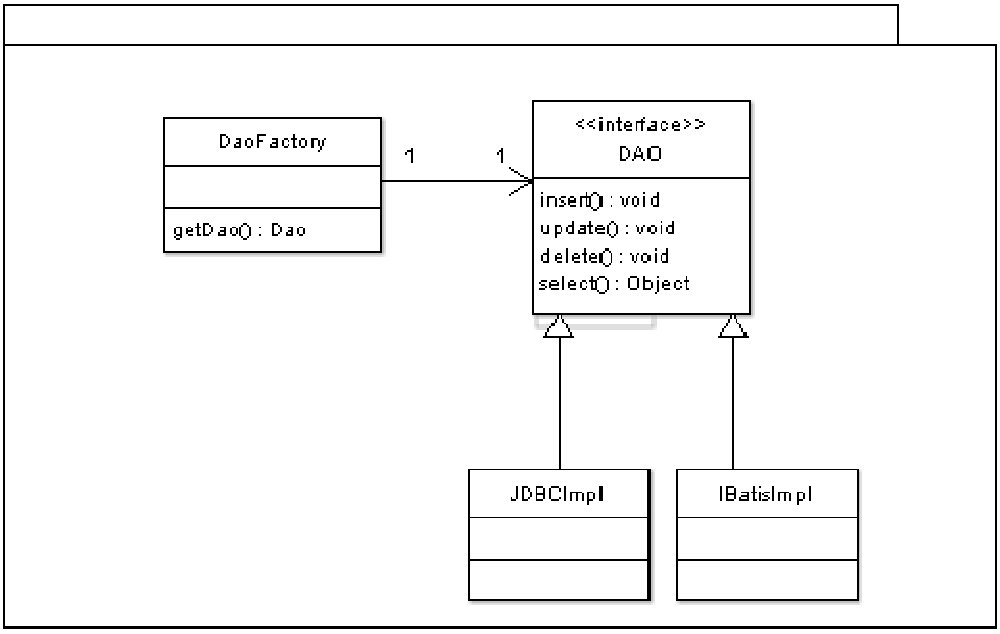
v.3.0 öncesi bilgibankası işlemlerini yapmak için sadece JDBC teknolojisini kullanıyordum. JDBC ile performans ve tuning işlemleri çok daha kolay bir hale geliyor. Ana derdiniz de performans olunca, o zaman doğal olarak JDBC teknolojisinde kalıyorsunuz, çünkü JDBC ile bilgibankası işlemlerine müdahale etmek çok daha kolay. Bu yüzden fırsat bulup, nesneleri bilgibankasında kalıcı hale getiren bir ORM (object relational mapping) aracı kullanmadım ve buna da açıkça gerek yoktu. Zamanla BizimAlem için geliştirdiğim yeni hizmetler ile tasarım nesnelere doğru kaymaya başlayınca, oluşan bu nesneleri JDBC ile bilgibankasına aktarmak çok kompleks bir hal almaya başladı.

Bugün kullanılmakta olan bir çok ORM aracı bulunmaktadır. Bunların başında Hibernate, Toplink, Cocobase ve IBatis gelir. Saf kan ORM aracı olan Hibernate ile nesneler birebir bilgibankası tablolarına yerleştirilir. Programcının oluşturulan JDBC koduna müdahale etme şansı pek yoktur. Tüm persistens işlemi Hibernate tarafından gerçekleştirilir. Bu doğal olarak performansın önemli olduğu alanlarda bir dezavantaj, çünkü SQL komutlarına direk müdahale etmek hemen hemen imkansız. Bu sebepten dolayı Hibernate gibi bir ORM aracını BizimAlem'de hiçbir zaman kullanma taraftarı olmadım. Benim aradığım ORM ile JDBC arasında olan bir araçtı. Hem nesneleri bilgibankasına kaydedebilmeliydim, hem de SQL komutları üzerinde değişiklik yaparak, nesnelerin bilgibankasına yerleştirilmesi ve tekrar edinilmesi işlemine müdahale edebilmeliydim. Bunu yapabilen IBatis isminde bir ORM aracı var.

IBatis herhangi bir sınıf için, o sınıfın nesneleri üzerinde işlem yapmak üzere sqlMap ismini taşıyan XML dosyalarında gerekli SQL komutların oluşturulmasını sağlamaktadır. Bir sonraki örnekte görüldüğü gibi resultClass olarak tanımlanan PortalVo sınıfından bir nesne oluşturmak için getPortalList isimli select kullanılmaktadır. Nesneyi oluşturmak için gerekli select komutu <select> tagı bünyesinde yer almaktadır. IBatis tanımlanan bu select komutunu kullanarak, PortalVo sınıfından bir nesne oluşturarak, aplikasyona bu nesneyi verir. Bu şekilde hem nesneler ile kullanım kolaylaştırılmaktadır, hem de kullanılan SQL komutlarına müdahale etmek mümkün olmaktadır.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sqlMap
PUBLIC "-//ibatis.apache.org//DTD SQL Map 2.0//EN"
"http://ibatis.apache.org/dtd/sql-map-2.dtd">
<sqlMap namespace="Poll">

      <select id="getPortalList"
resultClass="smart.core.cache.vo.portal.PortalVo">
            select
            id,username,subject,created,commentcounter from
community_portal order
            by created2 desc limit 6 offset #offset#
</select>

      <select id="getPortalById"
resultClass="smart.core.cache.vo.portal.PortalVo">
            select * from
            community_portal where id = #id#
</select>

</sqlMap>
```

Mevcut JDBC tabanlı kod yanı sıra IBatis'i kullanabilmek için DAO[5] tasarım şablonunu kullanmaya karar verdim.



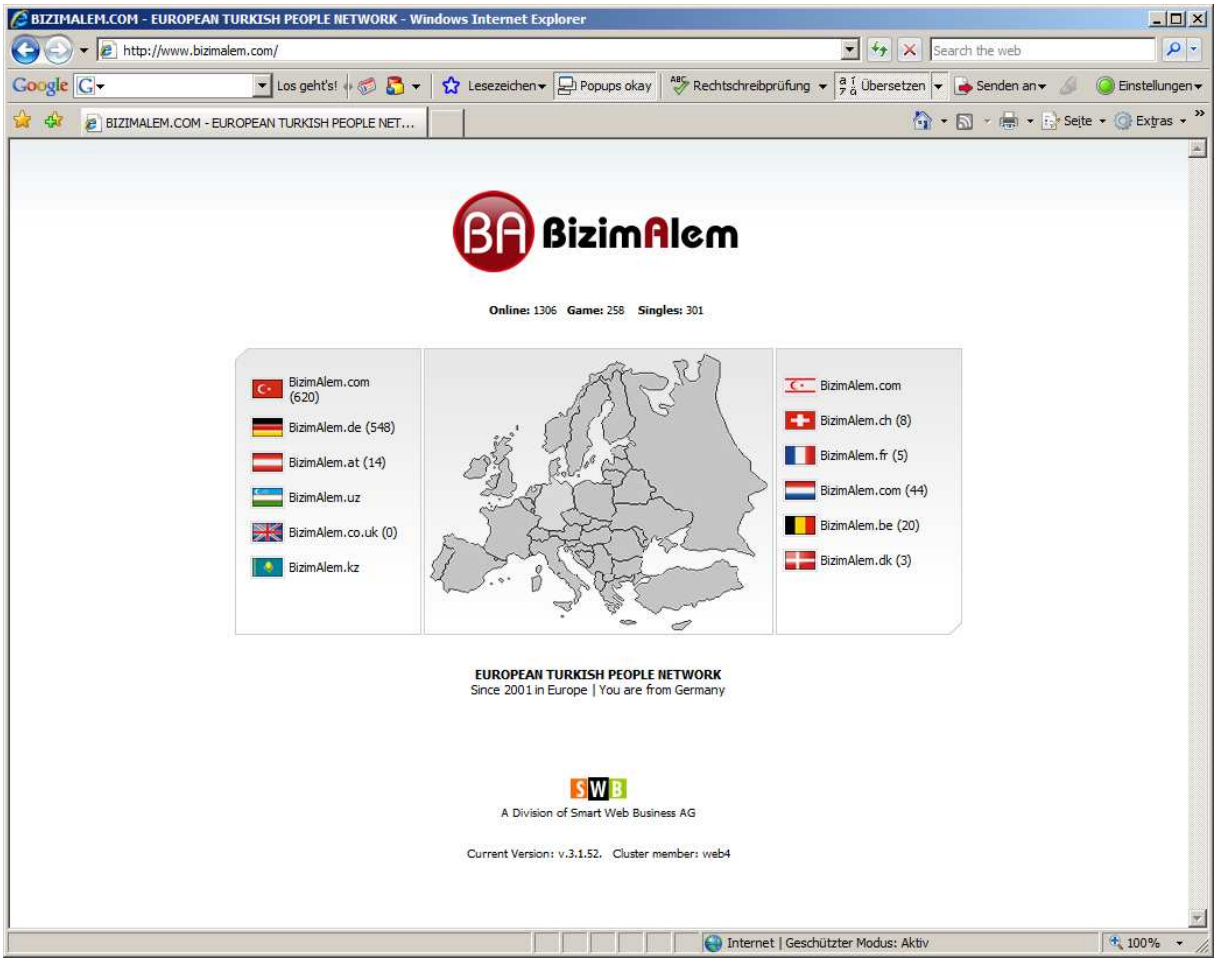
Resim 18 DAO tasarım şablonu

DAO bir interface sınıf ve içinde insert, update, delete, select gibi metotlar yer almaktadır. İki değişik teknolojiyi, yani JDBC ve IBatis aynı anda kullanabilmek için DAO interface sınıfının iki değişik teknoloji için implemente edilmesi gerekir. Bu sebepten dolayı tamamen JDBC koduna dayalı olan JDBCImpl ve IBatis ORM aracını kullanan IBatisImpl implementasyonlarını oluşturdum. DAOFactory üzerinden yerine göre JDBCImpl ya da IBatisImpl implementasyonlarını kullanmam kolay bir hale geldi. Yeni oluşturduğum BizimAlem servisleri için sadece IBatisImpl implementasyonunu kullanmaya başladım ve bu şekilde bu servislerin tamamen nesneler ile oluşturulması kolay bir hal aldı.

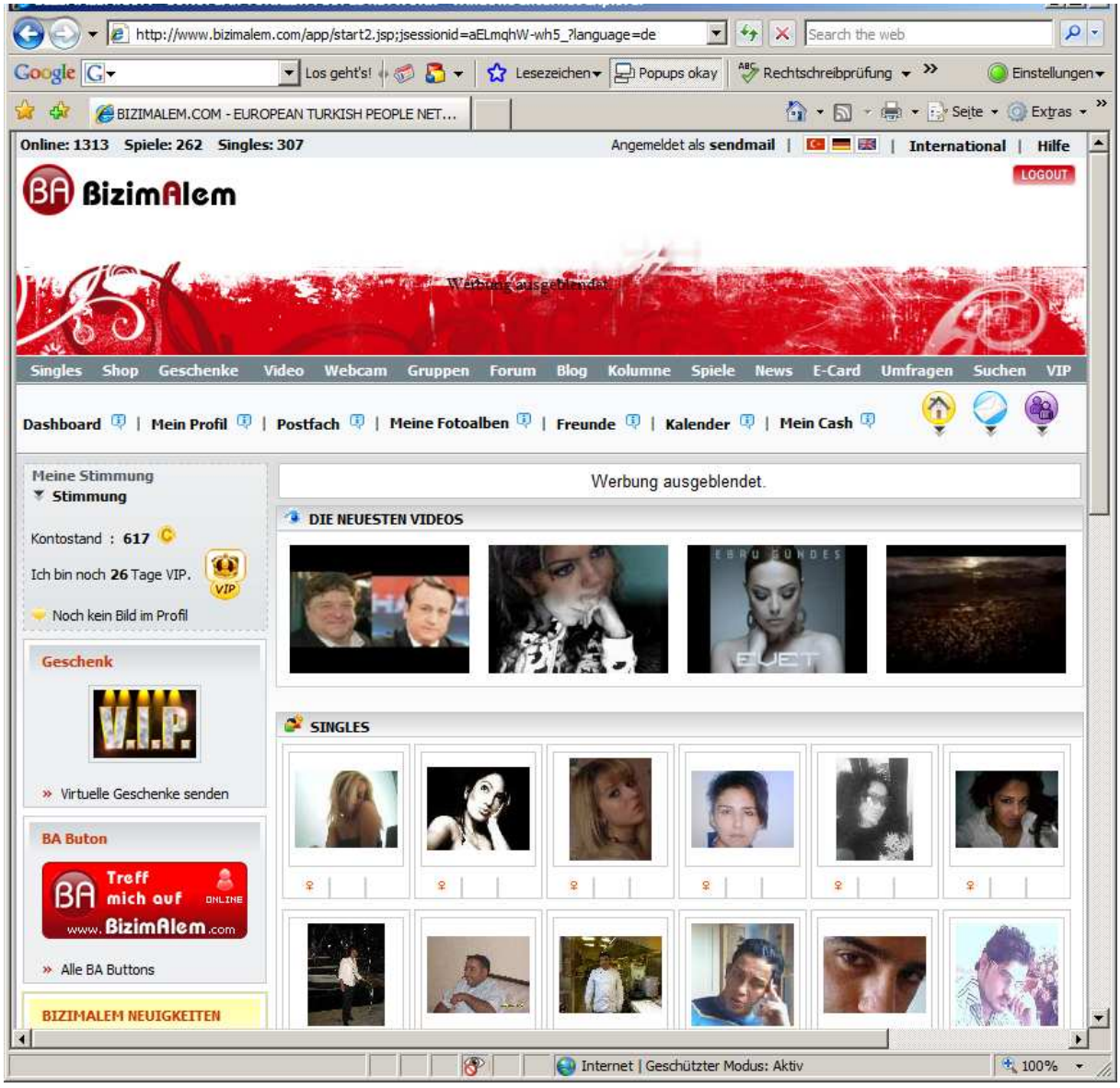
Yeni Tasarım

v.3.0 ile tasarım ve kullandığım BizimAlem logosu da değişti. Yeni tasarım resim 19,20 ve 21 de yer almaktadır.

[5] Bakınız: http://www.kurumsaljava.com/2008/12/01/data-access-object-dao-tasarim-sablonu/



Resim 19 BizimAlem v.3.1.52 versiyonu giriş sayfası



Resim 20 BizimAlem v.3.1.52 versiyonu ana sayfa

Resim 21 BizimAlem v.3.1.52 versiyonu oyun modülü

24.6.2007 tarihinde v.3.0 online oldu. Yeni versiyon online olmadan önce bilgibankasında geniş çaplı bir temizlik yaptım. İlk etapta 6 aydır kullanılmayan tüm kullanıcı hesaplarını sildim. Bu işlemin ardından aktüel kullanıcı sayısı 300.000 in altına düştü. Bu iyi bir rakamdı, çünkü bu üyelerin hemen hemen hepsi platformu aktif olarak kullanıyorlardı. v.3.0 online olduktan sonra tüm üyelere email aracılığıyla duyuru yaparak, yeni versiyonu tanıttım. Bu şekilde sistemi uzun süredir kullanmayanların da tekrar dikkatini çekme fırsatı buldum.

v.3.0 kullanıma açıldıktan sonra doğal olarak bazı sistem hataları ortaya çıktı. Bu hataları gidermeye çalışırken kullanıcıların sistem üzerindeki faaliyetleri artığından dolayı oluşturduğum bilgibankası cluster sistemi tekrardan dar boğaz haline gelmeye başladı. Yeni oluşturduğum BizimAlem servisleri ayrıca sisteme ek yük getirdiklerinden dolayı, bilgibankasının dar boğaz olması süreci hızlandırılmış oldu. Bu sorunu çözmek için artık elimde sadece bir silah kalmıştı: Caching!

Caching Nedir?

Ön belleğe alma olarak tercüme edebileceğimiz caching mekanizmaları ile kullanılmak istenen veri, verinin kaynağına ulaşmak zorunda kalmadan, bilgisayar hafızasında (ram) tutulur. Bilgisayar hafızasına erişim, verinin bulunduğu kaynağa (örneğin harddisk) erişimden daha hızlı olduğu için, verinin işlenme süresi kısaltılmış olur. Ayrıca verinin kaynağına erişim veri transferi açısından bir dar boğaz olabileceği için, caching mekanizmaları ile dar boğazlılık sorunu giderilir.

Caching mekanizmaları birçok alanda kullanılmaktadır. Bunlardan bazıları şöyledir:

- Ana işletim birimi (cpu) hafıza köprüsü (memory bus) üzerinden gerekli verileri elde ettikten sonra bu verileri kendi bünyesinde barındırdığı cache alanlarında saklar. Bu işletim biriminin çalışma hızını artırır.
- Harddiskler cache mekanizmaları kullanarak, diskler üzerinde bulunan verileri tekrar edinmek zorunda kalmadan kullanıcı sisteme aktarabilirler. Bu harddisklerin performansını artırır.
- Veriler bilgibankasından okunduktan sonra caching mekanizmaları kullanılarak hafızada tutulur. Bu verileri kullanan programın, sıkça bilgibankasını kullanmak zorunda olmadığı için performansını artırır.

Cache içinde tutulan veri şüphesiz orijinal verinin bir kopyasıdır. Er ya da geç veri asıl bulunduğu kaynak üzerinde (örneğin bilgibankası) değişikliğe uğrayacaktır. Bu durumda cache içinde bulunan kopyanın statüsü nedir?

Eskiye rağbet olsaydı, bit pazarına nur yağardı demiş atalarımız ☺ Kopya hiç bir zaman orijinalın yerini alamayacağı için kıymet taşımaz. Lakin bu bilgisayar sistemlerinde işlenen veriler için geçerli değil. Bir kopya veri, orijinalı değişmediği sürece onunla aynı değerdedir. Ne zaman orijinal veri değişikliğe uğradı, o zaman kopya için ölüm çanları çalmaya başlar. Orijinal veri değişikliğe uğradıktan sonra, cache içinde bulunan kopyanında değişikliğe uğraması gerekmektedir, aksi taktirde cache içinde bulunan veriyi kullanan program verinin en son haline sahip olmadığı için yanlış işlem yapabilir hale gelecektir. Buradan söyle bir sonucu çıkartabiliriz: "**Bir cache içinde tutulan verinin belli bir zaman diliminden sonra kendisini imha ederek, verinin tekrar asıl kaynağından edinilmesini sağlaması gerekmektedir**".

Cache içine atılan bir veri için geçerlilik süresi tespit edilmelidir. Aksi taktirde veri sonsuza dek (bilgisayar restart edilene kadar) cache içinde kalacak ve verinin tekrar asıl kaynağından edinilmesini engelleyecektir. Örneğin bir müşterinin bilgilerini ihtiva eden bir nesne cache içine atılmadan önce 30 saniye geçerli olacak şekilde yapılandırılabilir. Her 30 saniyenin bitiminde müşteri bilgileri bilgibankasından tekrar edinilerek, cachelenir. Buradan şöyle bir sonucu çıkartabiliriz: "**Caching mekanizmasının sağlıklı çalışabilmesi ve kullanıcı programı yanıltmaması için verilerin geçerlilik süresine (expiration) sahip olmasına gerekmektedir**". Geçerlilik süresi nesneyi cache sistemine atan program tarafından belirlenen bir özelliktir ve bir zaman birimini (saniye, dakika, saat) ihtiva eder. Örneğin sıkça değişikliğe uğramayan veriler için 1 gün, 1 ay gibi geçerlilik süresi tayin edilebilir. Sıkça değişikliğe uğrayan veriler için bu süre 30 saniyenin altında olacaktır.

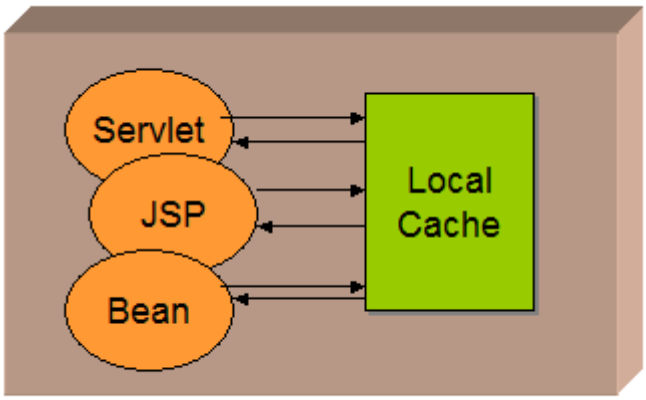
Bir veriyi caching sistemine atmamız ve geçerlilik süresini belirlememiz yeterli değildir. Veriyi tekrar edinebilmek için bu veriyi **adresleyebilmemiz** gerekmektedir. Bunu bir anahtar (key) kullanarak yapabiliriz. Bu anahtarın tüm sistem içerisinde eşsiz olması gerekmektedir. Sadece bu sayede istenilen veriye ulaşılabilir ve aynı anahtarı taşıyan iki verinin birbirlerini yok etmeleri engellenir.

Caching Türleri

Caching sistemleri **lokal** ve **global** olmak üzere iki gruba ayrılır.

Lokal Caching Sistemleri

Bu tür caching sistemlerinde caching sistemi kullanıcı program ile aynı hafıza alanında (in memory) faaliyet gösterir. Kullanıcı program ile aynı hafıza alanı ve JVM içinde bulunan caching sistemi, kullanıcı programa yakınlığından dolayı lokal caching sistemi olarak isimlendirilir.



Resim 22 Lokal caching sistemi

Lokal caching sistemlerinin performansı yüksektir, çünkü caching işlemleri kullanıcı program ile aynı hafıza alanını paylaşan lokal cache üzerinde hızlı bir şekilde gerçekleştirilir.

Lokal cache sistemlerinde sadece bir tane cache kopyası (instance) mevcuttur. Her aplikasyon için bir lokal cache kopyasının oluşturulması gerekmektedir. İki değişik adres alanında faaliyet gösteren aplikasyon aynı lokal cache kopyasını kullanamazlar.

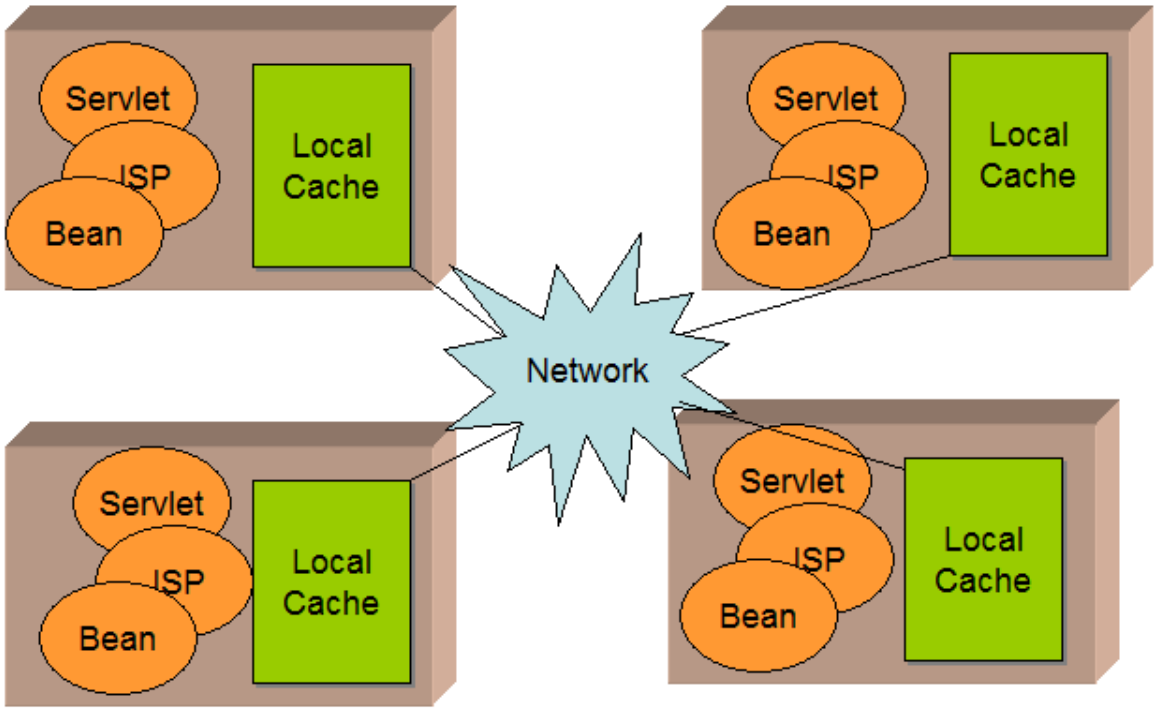
Global Caching Sistemleri

Birden fazla server üzerine dağıtılmış (distibuted) ama tek hafıza alanıymış gibi faaliyet gösteren caching sistemlerine global caching sistemi adı verilir. Global caching sistemleri kendi aralarında iki gruba ayrılır:

- Global tek hafıza caching sistemleri
- Global bölümsel (partial) hafıza caching sistemleri

Global Tek Hafıza Caching Sistemleri

Bu tür caching sistemlerinde her aplikasyon serveri kendi hafıza alanında bir lokal caching sistemine sahiptir. Birden fazla aplikasyon serveri ve lokal caching sisteminin kullanılması bir global cache oluşturulduğu anlamına gelmez. Global bir caching sisteminin oluşabilmesi için ağ içindeki lokal caching sistemlerinin birbirlerinden haberdar olmaları gerekmektedir. Kullanılan caching sistemleri ağ üzerinden birbirleriyle bağlantı kurarak, bünyelerinde meydana gelen değişiklikleri ağ içindeki diğer lokal cachelere bildirirler. Bu işlem için JGroups ya da JMS gibi komünikasyon teknolojileri kullanılır. Bu sayede bir global caching sistemi oluşur.



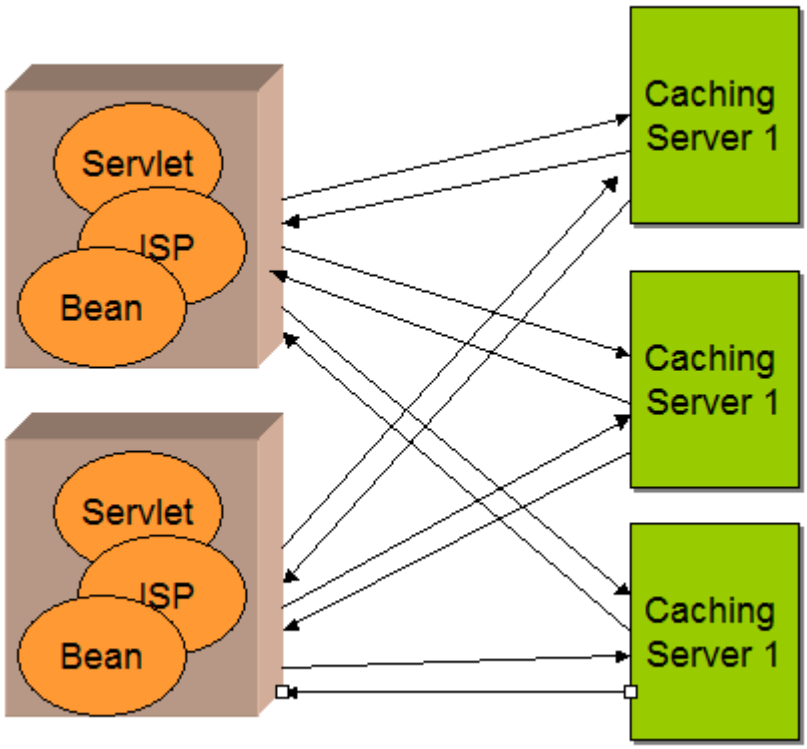
Resim 23 Global caching sistemi

Lokal cacheler arası verinin replikasyonu (kopyalanması) ile tüm lokal cacheler aynı veriye sahip olur.

Bu tarz global bir caching sistemi oluşturabilmek için verinin lokal cache sistemleri arası replike (kopyalanması) edilmesi gerekmektedir.

Global Bölümsel (Partial) Hafıza Caching Sistemleri

Bu tür global caching sistemlerinde caching komponentleri arasında replikasyon yapılmaz.



Resim 24 Global bölümsel caching sistemi

Global caching sistemini kullanmak isteyen her aplikasyon, sistemdeki tüm caching serverleri ile bağlantı kurarak, caching işlemlerini gerçekleştirir. Kullanılan özel bir caching API yardımı ile veriler değişik caching serverlerine dağıtılır. Bu yüzden bu tür faaliyet gösteren caching sistemlerine global bölümsel caching sistemleri adı verilmektedir, çünkü veriler değişik caching serverlerine dağıtılarak cachelenir. Daha sonra yakından inceleyeceğimiz MemCached bu tarz caching sistemleri kurmak için kullanılabilecek bir caching sistemidir.

Caching Konseptleri

Bu yazıda ismi geçen caching konseptlerini yakından inceleyelim:

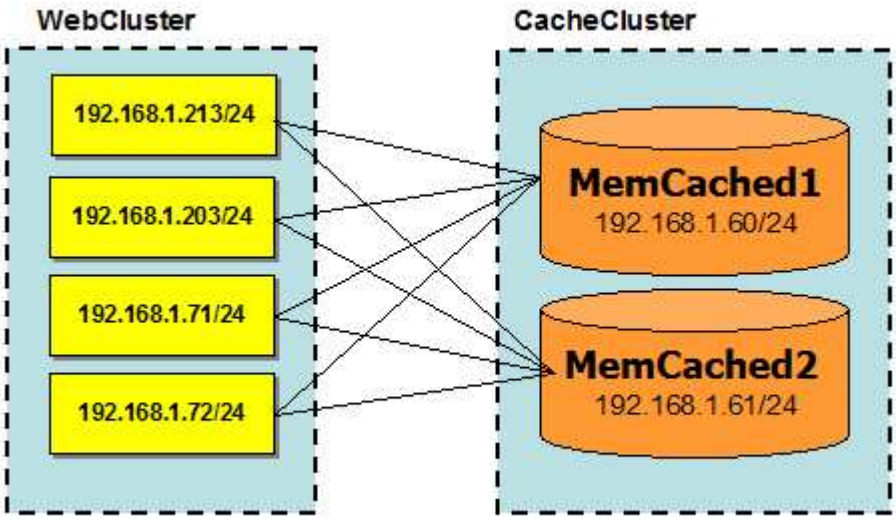
- Ön belleğe anılacak olan verinin kendisi (**cacheable object**) . Örneğin bir sınıftan oluşturulan bir nesne veri olarak ön belleğe alınabilir.
- Verinin ön bellekteki kalma süresi (**expire time**). Bu değer baz alınarak, verinin ön bellekte hangi zaman dilimi için kalacağı belirlenir. Verinin ön bellek içinde sonsuza dek kalmasını engellemek için limitli bir zaman diliminin seçilmesi gerekmektedir, örneğin 30 saniye.
- Veriyi ön belleğe yerleştirmek ve tekrar edinmek için kullanılan anahtar (**key**). Bu anahtarın sistem bünyesinde eşsiz (unique) olması gerekmektedir. Aksi

taktirde aynı anahtara sahip verilerin birbirlerini yok etmeleri ihtimali oluşmaktadır.

- Verileri ön belleğe almak için put olarak isimlendirilen işlem gerçekleştirilir. put işleminde ön belleğe alınacak verinin kendisi, verinin ön bellekteki kalış süresi ve veriyi adresleyen anahtar (key) parametre olarak kullanılır.
- Verileri ön bellekten edinmek için get işlemi gerçekleştirilir. get işleminde veriye ulaşmak için put işleminde oluşturulan anahtar (key) kullanılır.

BizimAlem Caching Sistemi

BizimAlem için global bölümsel hafıza caching sistemi olan MemCached serverini kullanmaya karar verdim. MemCached [6] client-server tarzı çalışan bir global bölümsel caching sistemidir. BizimAlem için oluşturduğum caching sistemi resim 25 de yer almaktadır.



Resim 25 BizimAlem Caching sistemi

Web cluster içinde bulunan her sunucu cache cluster içinde bulunan her memcached serveri ile bağlantı kurarak, caching işlemlerini gerçekleştirir.

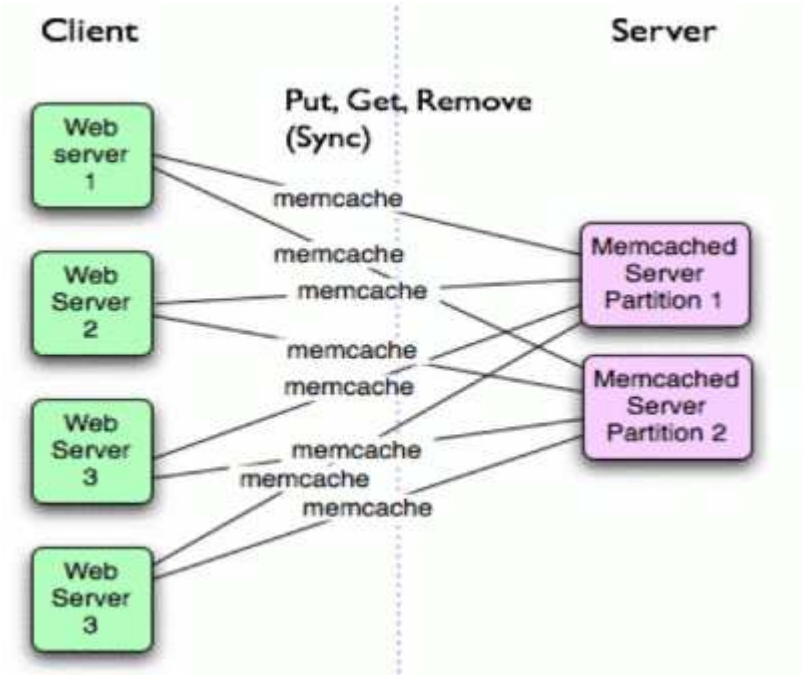
Fiziksel bir server üzerinden MemCached kurulduktan sonra, aşağıdaki şekilde çalıştırılır.

```
./memcached -d -m 2048 -l 192.168.1.10 -p 11211
```

Yukarda yer alan örnekte MemCached 192.168.1.10 IP numaralı server üzerinde 2 GB hafıza alanını kullanacak ve 11211 numaralı porttan erişilebilir olacak şekilde çalıştırılmıştır. Birden fazla MemCached ağ içinde aktif olabilir. MemCached serverler arasında veriler replikasyon (kopyalama işlemi) usulüyle senkron tutulmaz. MemCached serverlerine verileri transfer eden Client API (daha sonra detaylı olarak göreceğiz) cachelenmesi gereken verinin anahtarı yardımıyla eşsiz (unique) bir hash değeri oluşturarak, listeden bir MemCached

[6] Bakınız: http://www.danga.com/memcached/

serverini seçer ve veriyi direk bu MemCached serverine transfer eder. Veriyi edinmek için tekrar aynı anahtar kullanıldığında Client API yine aynı hash değerini oluşturarak, verinin hangi MemCached server üzerinde olduğunu belirler ve veriyi temin eder.



Caching sistemini kullanabilmek için Java dilinde implemente edilmiş bir MemCached Client API implementasyonuna ihtiyacımız var.

MemCached Client API

MemCached serveri ile bağlantı kurup, işlem yapabilmek için değişik dillerde Client API'ler[7] oluşturulmuştur. Java dilinde Greg Whalin[8] tarafından implemente edilen MemCached Client API'yi kullanıyoruz.

```
package smart.core.cache;

import java.util.Date;
import java.util.Map;
import com.danga.MemCached.MemCachedClient;
import com.danga.MemCached.SockIOPool;


public class MemcachedImpl extends BaseCache
{

      protected static MemCachedClient client =
            new MemCachedClient(Thread.currentThread().
```

[7] Bakınız: http://www.danga.com/memcached/apis.bml
[8] Bakınız: http://whalin.com/

```
                        getContextClassLoader());

    static
    {
      // get mcache servers
      String[] serverlist =
          getProperty( "memcached.servers" ).split( "," );

      // get server weights
      Integer[] weights      =
          new Integer[serverlist.length];

      String[] serverWeights =
          getProperty("memcached.servers.weights" ).split( "," );

      for ( int i = 0; i < serverWeights.length; i++ )
      {
          weights[i] = new Integer( serverWeights[i] );
      }


      // idle time set to 4 hours
      long maxIdle = 1000 * 60 * 60 * 6;

      // socket read timeout
      int readTO = 1000 * 3;

      // initialize the pool for memcache servers
      // see javadocs on this method
      // for tuning the connection pool
      SockIOPool pool = SockIOPool.getInstance();
      pool.setServers( serverlist );
      pool.setWeights( weights );
      pool.setInitConn( Integer.parseInt(getProperty("initcon")) );
      pool.setMinConn(  Integer.parseInt(getProperty("mincon")) );
      pool.setMaxConn(  Integer.parseInt(getProperty("maxcon")) );
      pool.setMaintSleep(  Integer.parseInt(getProperty("maintsleep")) );
      pool.setNagle( Boolean.getBoolean(getProperty("nagle")) );
      pool.setSocketTO( readTO );
      //pool.setSocketConnectTO( 0 );
      pool.setMaxIdle( maxIdle );
      pool.setFailover(Boolean.getBoolean(getProperty("failover")));
      pool.setFailback(Boolean.getBoolean(getProperty("failback")));
      pool.initialize();
      getClient().setCompressEnable(
                Boolean.getBoolean(getProperty("compression")) );
  }



    public static MemCachedClient getClient()
    {
          return client;
    }



    public void set(String key, Object obj)
    {
          getClient().set(key, obj);
    }
```

```
	public void set(String key, Object obj, Date date)
	{
		getClient().set(key, obj, date);
	}


	public Object get(String key)
	{
		return getClient().get(key);
	}


	public Map getStats()
	{
		return getClient().stats();
	}

	public void delete(String key)
	{
		delete(key,null);
	}

	public void delete(String key, Date expiry)
	{
		getClient().delete(key,expiry);
	}



	public Map getStats(String[] server)
	{
		return getClient().stats(server);
	}

}
```

Yukarda yer alan MemcachedImpl sınıfı ile MemCached serverlerini kullanmaya başlayabiliriz. Sistem için gerekli ayarları (örneğin hangi MemCached serverlerin kullanıldığı) memcached.properties dosyasında tutuyoruz. Bu dosyanın içeriği aşağıda yer almaktadır.

```
#PROD

# memcached.server=server1:port,server2:port....
memcached.servers=192.168.2.198:11211,192.168.2.199:11211,192.168.2.211:11211
memcached.servers.weights=1,2,1
memcached.activated=true


initcon                 = 100
mincon                  = 100
maxcon                  = 512
maintsleep              = 1000
nagle                   = false
alivecheck              = true
failover                = false
failback                = false
```

```
compression             = false

domain                  = prod

impl                    = smart.core.cache.MemcachedImpl
```

memcached.servers anahtarı, kullanılan MemCached serverlerin IP adresleri ve port numaralarını ihtiva etmektedir. Kullanılan MemCached serverler virgül ile birbirlerinden ayrılmıştır. Buna göre 192.168.2.198 - 192.168.2.199 - 192.168.2.211 IP numaralı serverler MemCached serverleridir ve 11211 numaralı port üzerinden bu servise ulaşılabilir.

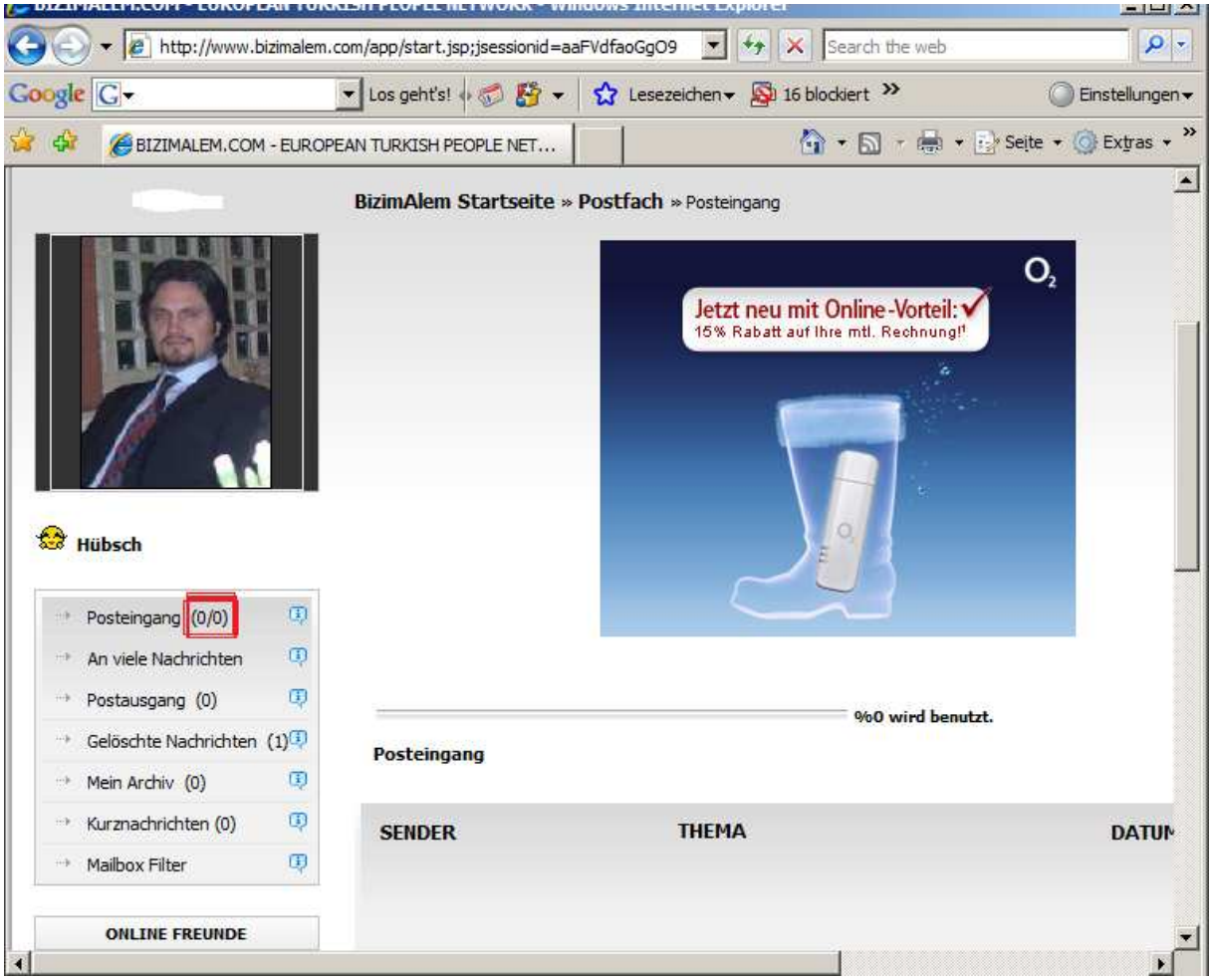
Kullanılan bazı diğer parametreler:

- **initcon = 100** – Client API her MemCached server için başlangıçta 100 adet TCP bağlantısı kurar. Böylece bir bağlantı (connection) havuzu (pool) oluşturularak, caching işlemleri hızlandırılmış olur.
- **mincon = 100** – Bağlantı havuzunda en az 100 bağlantı olacak şekilde ayarlamalar yapılır.
- **maxcon = 512** – Bağlantı havuzunda en fazla 100 bağlantı olacak şekilde ayarlamalar yapılır. Client API otomatik olarak bağlantı havuzunun hacmini ayarlar.

Java için kullandığımız Client API implementasyonunda MemCached serverlere olan bağlantı devamlı kontrol edilir. Eğer MemCached serverlerinden birisi görevini yerine getirmiyorsa, otomatik olarak mevcut server listesinden alınır. Bu şekilde çalışmayan komponentlerin devre dışı kalması sağlanır.

Bir sonraki kod bölümünde MemCached caching serverlerinin BizimAlem kodu bünyesinde nasıl kullanıldığı yer almaktadır.

```
public String getIncomingMailCounter() throws Exception
{
      logger.debug("getIncomingMailCounter()");

      String key = KeyAttributes.getKey(
                  KeyAttributes.KEY_MAILBOX_COUNTER , getUsername());
      String vo =null;

      try
      {

            vo = (String)CacheManager.instance().get(key);

            if(vo == null)
            {
                  vo = getIncomingMailCounter();
                        CacheManager.instance().set(key, vo,
                                    Expire.EXPIRE_IN_MINUTE_5);
            }
      }
      catch(Exception e)
      {
            reportError(e);
      }
      return vo;
}
```



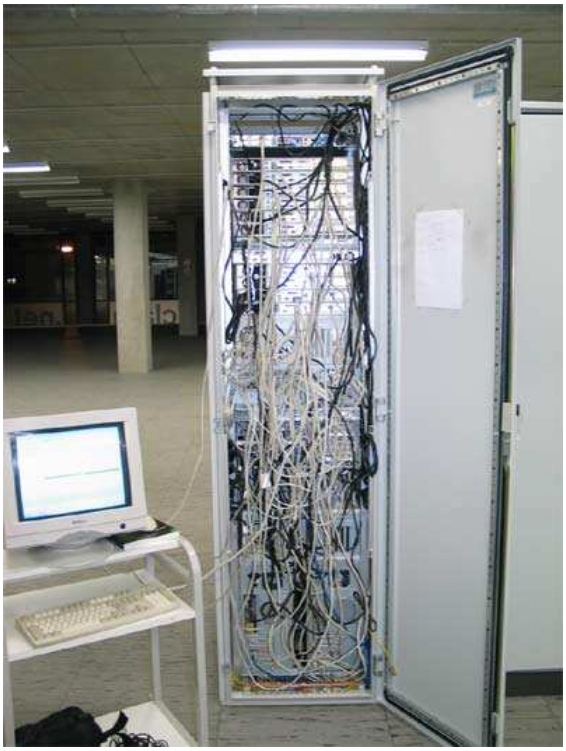
Resim 26 Üye posta kutusu

getIncomingMailCounter() metodu ile bir önceki resimde görüldüğü gibi posta kutusunda olan emaillerin adedi hesaplanmaktadır. Metot içinde vo isminde bir String tanımlanmaktadır. vo ismindeki değişken posta kutusundaki mektup adedini ihtiva eder. Bu veriyi bilgibankasından edinmeden önce, ClientManager sınıfı yardımıyla caching sisteminde arıyoruz. Bu işlem için bir anahtar oluşturmamız gerekiyor. ***KeyAttributes.getKey()*** metodu yardımıyla kullanıcıya has anahtar oluşturulmaktadır. Login yapmış bir üyeye ait bilgileri caching sisteminde tutabilmek için, anahtarın içinde üyenin ismi yer almaktadır. Üyenin isminin **test1** seklinde olacağını düşünürsek, oluşturulan anahtar **mailcounter:test1 ş**eklinde olacaktır. Bu şekilde değişik isimdeki üyeler için üyeye özel verileri caching sisteminde tutmak mümkündür.

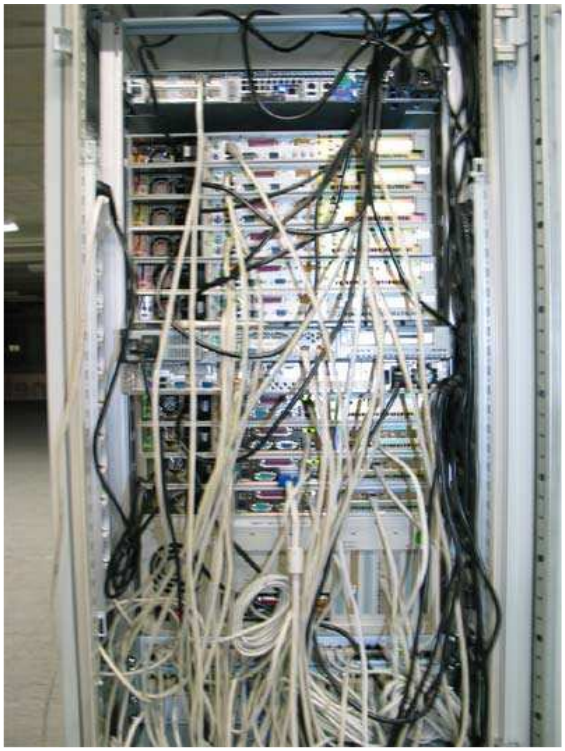
BizimAlem için oluşturduğum caching sistemi tüm performans sorunlarının kökünü kazıdı. Daha önce neden yapmamıştım acaba? ;-)

BizimAlem Server Altyapısı

BizimAlem için aktüel sürümünde iki dolap dolusu sunucu hizmet vermektedir. Kullanılan altyapı ve sunucuların fotografları aşağıda yer almaktadır.



Resim 27 BizimAlem server sistemleri

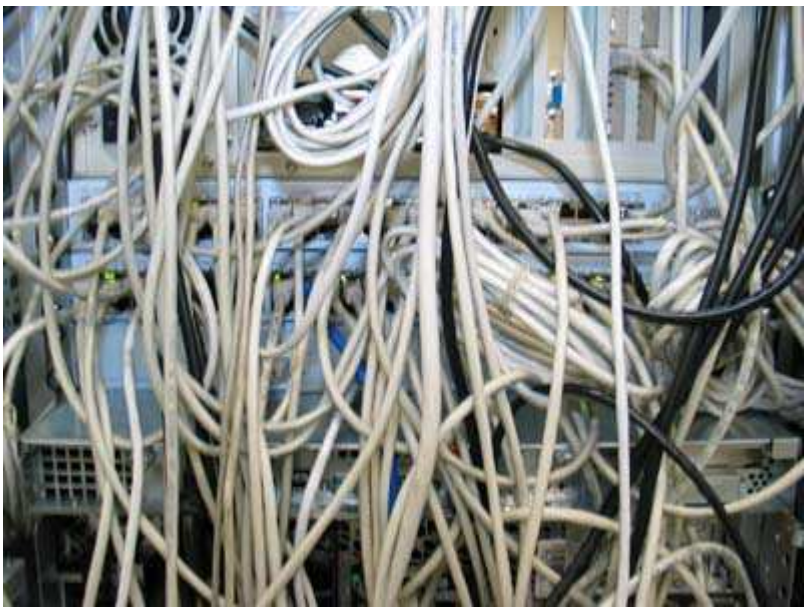


Resim 28 BizimAlem server sistemleri

Resim 29 BizimAlem server sistemleri

Resim 30 BizimAlem server sistemleri

Resim 31 BizimAlem server sistemleri

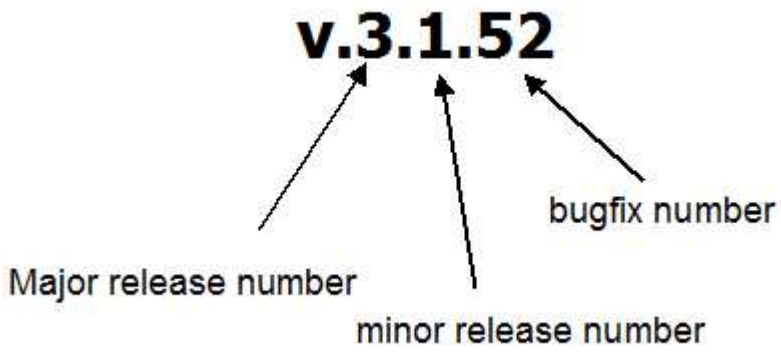


Resim 32 BizimAlem server sistemleri

Resim 33 BizimAlem server sistemleri

BizimAlem Sürüm Numarası

Bugün itibariyle (14.1.2009) BizimAlem'in aktüel sürümü v.3.1.52 dir.



Resim 34 BizimAlem sürüm numarasi

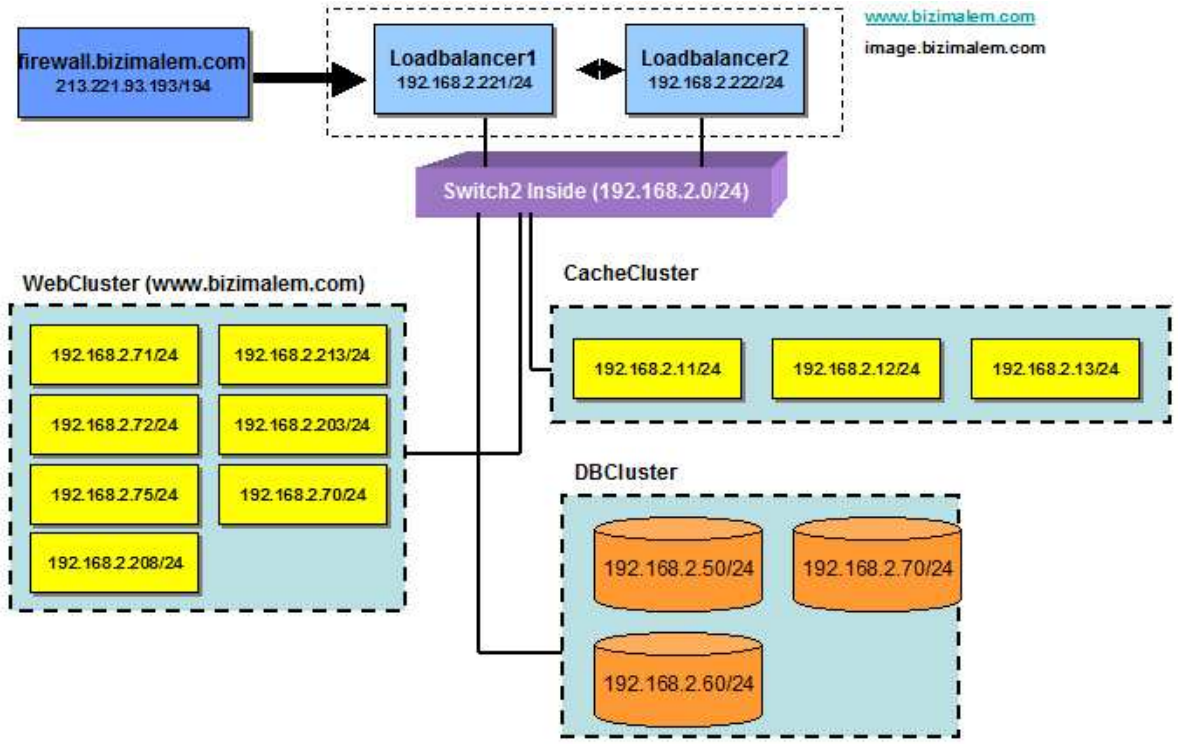
Sürüm numarası değişik bileşenlerden oluşmaktadır. Bunlar:

- **Major release**: 3 rakamı ana sürüm numarasını gösterir. BizimAlem bu hali ile 3. versiyonda. Sadece sistemde büyük çapta değişiklikler yapıldığında major release numarası bir artırılır. Büyük bir ihtimalle BizimAlem v.4 olmayacak, çünkü proje için 6 seneyi aşkın bir zamandır yapılması gereken herşey yapıldı.
- **Minor release**: 1 rakamı alt sürüm numarasını gösterir. Sisteme eklenen her yeni modül ile bu rakam bir artırılır. Örneğin su an planladığım bir chat modülü var. Bu modül sisteme eklendikten sonra v.3.2.0 sürümü oluşacak.
- **Bugfix number**: Oluşan her hatanın giderilmesiyle bu rakam bir artırılır. Ne yazık ki programalama esnasında kullanıcının yapabileceği tüm işlemler göz

önünde bulundurulamadığı için yeni hatalar oluşmakta ve bu hatalar her yeni sürümle giderilmektedir.

Son Teknik Altyapı Şekli

v.3.1.52 ile oluşan teknik altyapı bir sonraki resimde yer almaktadır.



Resim 35 v.3.1.52 sürümü teknik altyapı

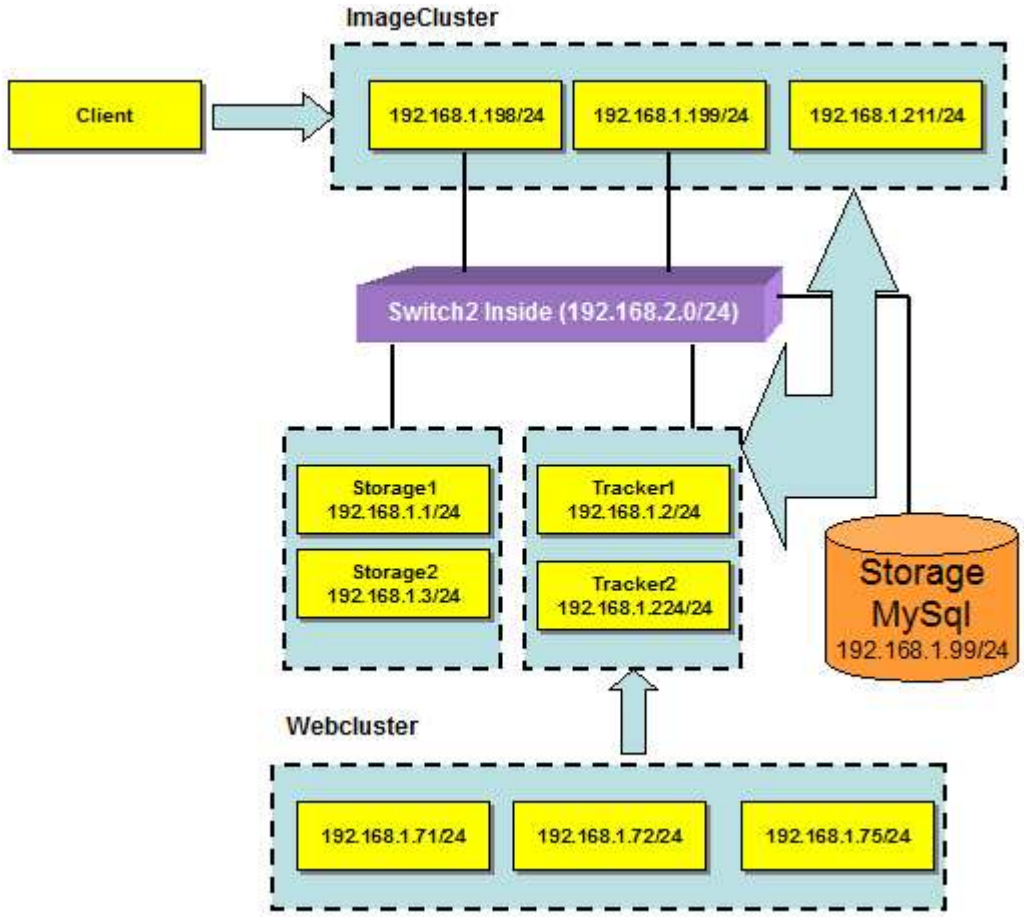
BizimAlem Storage Subsystem

BizimAlem üyeleri kendi resimlerinin yer aldığı albümler oluşturabilirler. Her albüme limit olmadan resim yüklenebilir. Albümlerin adedi de limitli değildir. Yarım milyondan fazla üyenin olduğu bir platformda, kaç milyon resmin bu şekilde sisteme ekleneceğini düşünebilirsiniz. Bu kadar büyük bir resim kütlesini sistemde barındırabilmek için merkezi bir storage (dosya deposı) sistemi oluşturulması gerekmektedir. Bu storage sisteminin kapasitesi, sistemin diğer bölümlerini etkilemeden artırılabilmelidir, yani yukarıya doğru sınırı olmayan bir storage sistemi gerekli!

Uzun bir süre 2 TB kapasitesi olan bir server ile bu sorunları çözebileceğimi düşündüm. Lakin sunucu bir zaman sonra 1.9 TB sınırına dayandığında, sadece bir sunucu ile bu sorunu çözemeyeceğimi anladım. Bunun yanısıra bu sunucunun teknik sorunlardan dolayı bazen devre dışı kalması, tüm dosyaların sistem tarafından erişilemez hale gelmesini beraberinde getiriyordu. Uzun araştırmalar sonunda açık kaynaklı olan ve benim beklentilerime cevap verebilecek bir storage sistemi buldum. MogileFS!

MogileFS ile Merkezi Storage Sistemi

MogileFS [9] Danga Interactive tarafından geliştirilmiş, dağınık çalışabilen (distributed) bir dizin sistemi (file system). Storage kapasitesini istediğiniz kadar, yeni sunucular ekleyerek yükseltebiliyorsunuz. MogileFS ile kurduğum storage sisteminin teknik mimarisi bir sonraki resimde yer almaktadır.



Resim 36 BizimAlem Storage mimarisi

Storage sistemi değişik komponentlerden oluşuyor. Bunlar:

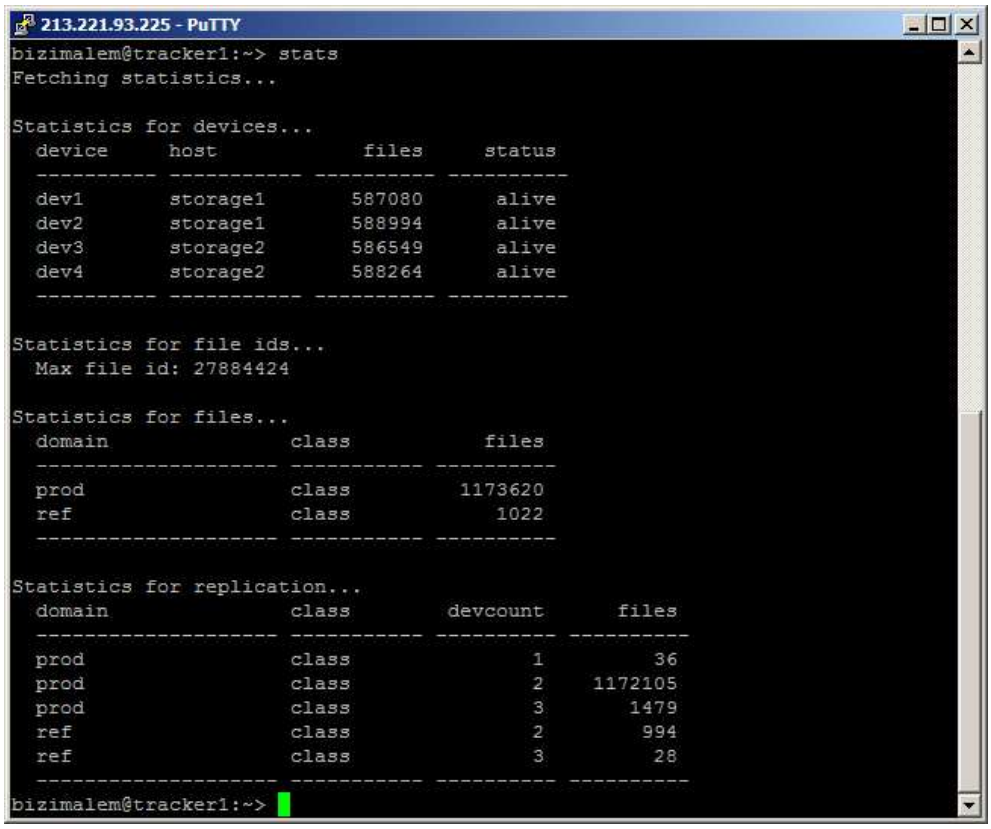
- **ImageCluster**: Storage sisteminde bulunan dosyalara ulaşmak için kullanılan sunucular. Storage sisteminde resimler, pdf dosyaları, word dokümanları vs olabilir. Bu dosyalara erişim ImageCluster üzerinden gerçekleşir.
- **Storage1:** Dosyaların depolandığı sunuculardan bir tanesi. 1 TB kapasiteli bir sunucu.
- **Tracker1:** Storage sisteminde bulunan her dosyanın unique (tek) bir numarası bulunmaktadır. Sisteme yeni bir dosya eklemek için webcluster içinde bulunan sunucular Tracker1/Tracker2 sunucuları ile bağlantı kurarak, dosyanın Storage1 ya da Storage2 de depolanmasını talep ederler. Bu sunucular dosyanın hangi Storage node üzerinde kayıtlı olduğunu tekrar Tracker1 ya da Tracker 2 üzerinden öğrenebilirler. Aynı şekilde bir kullanıcı bir dosyayı edinmek istediğinde

[9] Bakınız: http://www.danga.com/mogilefs/

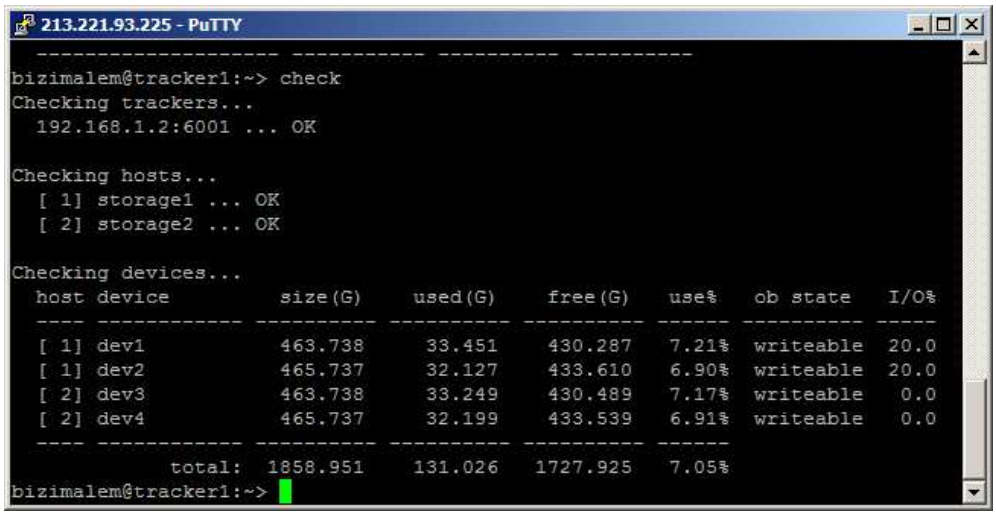
ImageCluster içinde bulunan bir sunucu Tracker1/Tracker2 ile bağlantı kurarak, dosyanın numarası ile Tracker üzerinden dosyayı edinir. Tracker hangi dosyanın hangi Storage node üzerinde olduğunu bilen komponenttir. Bu bilgiyi edinmek için MySQL bilgibankasını kullanır.

- **MySQL:** Hangi dosyanın hangi Storage node üzerinde olduğu ve hangi numaraya sahip oldugu MySQL bilgibankasında tutulur. Tracker MySQL bilgibankasına bağlantı kurarak, dosyanın hangi Storage node üzerinde olduğunu tespit eder. Tracker herhangi bir dosyayı Storage nodelardan birisine gönderdikten sonra, gerekli bilgileri MySQL bilgibankasına ekler. Storage sisteminin yönetimi Tracker ve MySQL komponentleri ile gerçekleşmektedir.

Mevcut storage kapasitesini artırmak için yeni StorageX sunucularının sisteme eklenmesi yeterli olmaktadır. Bu şekilde 1000 TB bile depolamak mümkün oluyor!



Resim 37 Storage istatistikleri



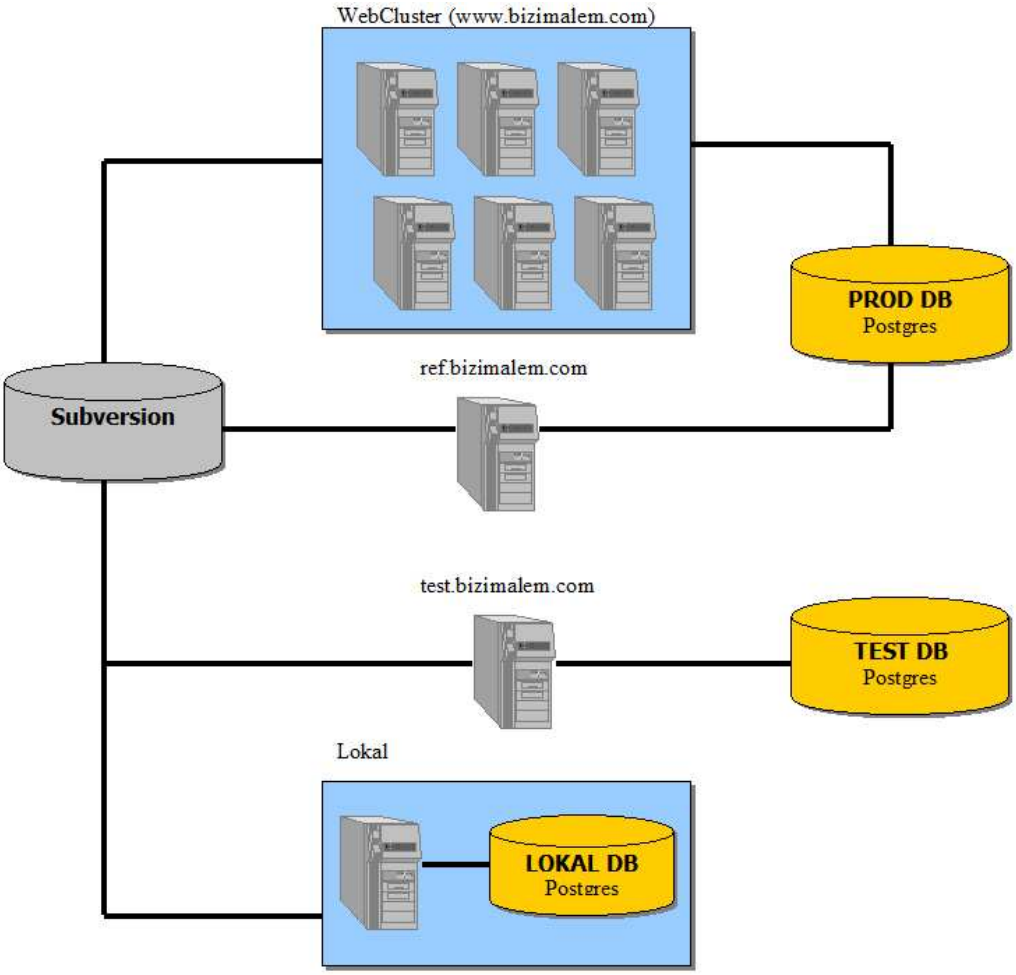
Resim 38 Storage istatistikleri

Resim 37 aktüel storage istatistiklerini göstermektedir. Storage1 sunucu bünyesinde dev1 ve dev2 isminde iki harddisk bulunmaktadır. Bu harddisklerin kapasitesi 0.5 TB dir. Storage2 de aynı konfigürasyona sahiptir. Bu konfigürasyonda storage sisteminde iki Storage node sunucu ve 4 hardisk bulunmaktadır. Resim 37 de her harddisk bünyesinde 600.000 a yakın dosyanın bulundugu görülmektedir. Sistem tarafından oluşturulan her dosyanın iki kopyası Storage1 ve Storage2 üzerinde paylaştırılır. Bu durumda Storage nodelardan birisi devre dışı kalsa bile dosyanın ikinci kopyasına diğer Storage node üzerinden ulaşmak mümkündür.

Resim 38 de Storage node kullanım istatikstikleri yer almaktadır. dev1 ve diğer harddiskler yaklaşık 500 GB kapasiteye sahiptir. Kullanılan storage alanı 34 GB civarındadır. dev1 ve dev2 bu snapshot yapıldığında %20 I/O (Input / Output) trafiğine sahiptir.

BizimAlem Development Staging Server

Yazılım süreci değişik evrelerden oluşur. Programcılar oluşturdukları kodu yazılım için kullandıkları bilgisayarlarında lokal test ederler. Bunun yanısıra kodun entegrasyonu için merkezi bir sunucu kullanılır. Kod merkezi versiyon kontrol sistemine eklendikten sonra, belirli aralıklarla yeni test sürümleri oluşturularak, bir test serveri üzerinde çalıştırılır. Bu test ortamının kendine ait bir bilgibankası vardır. Bu testler olumlu sonuç verdikten sonra oluşturulan sürüm referans olarak isimlendirilen server üzerinde çalıştırılır. Referans serveri merkezi bilgibankası sistemini kullandığı için, oluşan kod gerçek şartlarda test edilmiş olur.



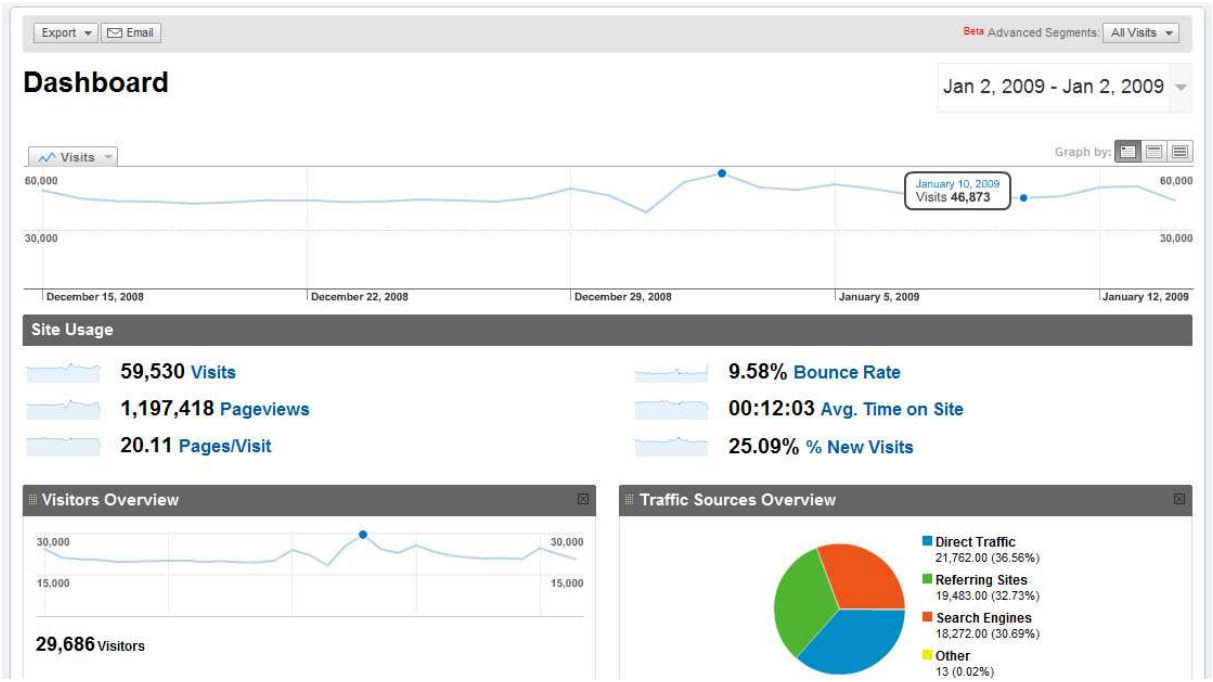
Resim 39 BizimAlem staging server yapısı

Kodun değişik ortamlarda değişik şartlarda test edilmesi için kullanılan server sistemlerine staging server ismi verilir. BizimAlem için kullanılan staging server yapısı resim 39 da yer almaktadır. Bunlar:

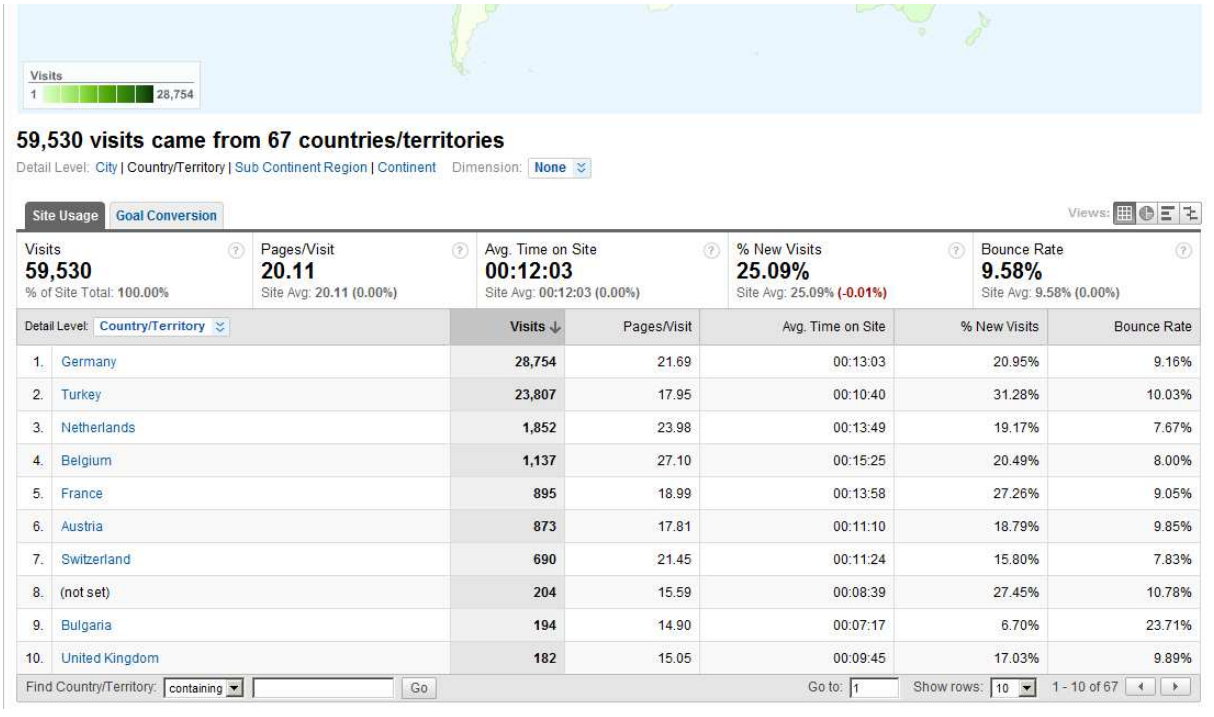
- **Lokal**: Yazılımı kendi bilgisayarimda yapıp, lokal çalışan bir bilgibankası ile test ettikten sonra oluşan kodu versiyon kontrol sistemine eklerim.
- **test.bizimalem.com**: Entegrasyon testleri test.bizimalem.com serverinde gercekleştirilir. Yeni bir test sürümü oluşturarak, bu sürümü test.bizimalem.com isimli sunucu da deploy (aplikasyonun çalışır hale getirilmesi) ederim. test.bizimalem.com kendine ait bir bilgibankasına sahiptir.
- **ref.bizimalem.com**: Entegrasyon testi tamamlanmış sürüm ref.bizimalem.com serverine deploy edilir. ref referans kelimesinin kısaltılmış halidir. Ref sistemi çalışan ana produktif sistem ile aynı bilgibankasını paylaştığı için (PROD DB), oluşturulan yeni kodun gerçek şartlarda test edilmesi sağlanır. Çoğu ref sistemlerinde PROD DB den bir kopya alınarak REF DB oluşturulur. Ben sürüm oluşturma işlemini kısaltmak için oluşturduğum ref sisteminin PROD DB yi kullanması sağladım.
- **www.bizimalem.com**: Gerçek sistem.

Kod ref sisteminde başarılı bir şekilde test edildikten sonra v.3.2.0 şeklinde yeni bir sürüm oluşturularak, bu yeni sürüm WebCluster icinde bulunan serverlere deploy edilir.

Google Analytics İstatistikleri



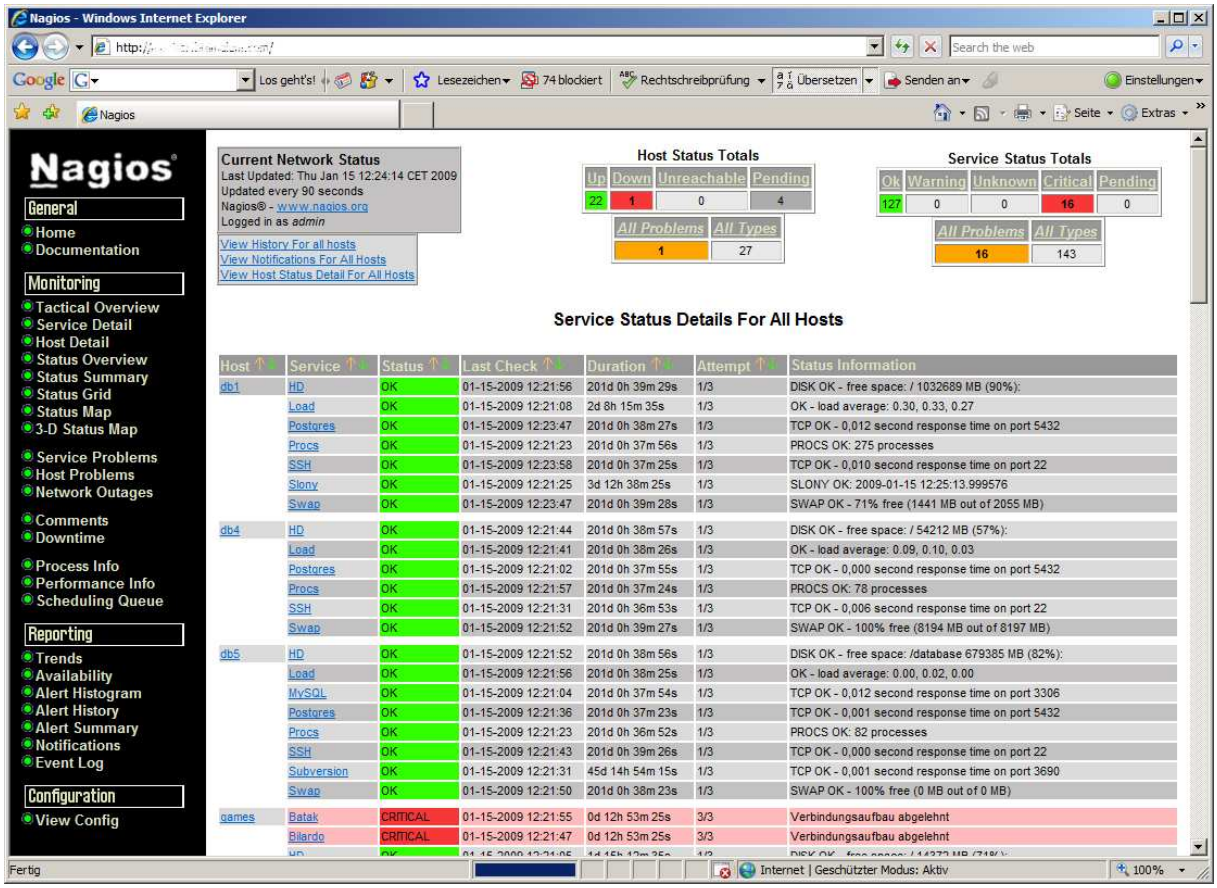
Resim 40 Google BizimAlem.com istatistikleri



Resim 41 Google BizimAlem.com istatistikleri

Ağ ve Servis Yönetimi

Kırka yakın server ve ağ komponentinin oluşturduğu bir sistemi yönetmek ve kontrol altında tutabilmek için bir network monitoring sistemine ihtiyacınız var. Her sunucu üzerinde otomatik olarak kontrol (monitoring) edilmesi gereken birden fazla servis olacaktır. BizimAlem'in ağ ve sunduğu servislerin kontrolünü Nagios[10] isimli network monitoring sistemi ile yapıyorum.



Resim 42 Nagios

Resim 42 de yer aldığı gibi her sunucu (server) için kullanılan servisler Nagios tarafından belirli aralıklarla kontrol edilmektedir. Çalışır durumda olan servisler için Status alanında OK yer almaktadır. Çalışır durumda olmayan, yani devre dışı kalmış servisler için Status alanında CRITICAL bilgisi yer alır. Devre dışı kalan her sunucu ve servis için Nagios ağ yöneticilerine otomatik olarak email göndererek, durumu bildirir.

Son

BizimAlem.com projesi yedi yaşını doldurdu ve Avrupa'da geniş bir kitleye hitap eden bir web platformu haline geldi. Bu makalemde böyle bir projenin hangi evrelerden geçtigini ve artan talep sonucunda nasıl büyümesi gerektiğini anlatmaya çalıştım. Daha önce de belirttiğim gibi BizimAlem.com benim için bugün bile büyük bir yazılım laboratuarı. Birçok yeni

10 Bakınız: http://www.nagios.org

teknolojiyi bir laboratuarda deneme ve öğrenme imkanı buluyorum. Oluşan sorunları çözmek için değişik alanlarda ihtisas yapmam gerekiyor. Bu severek yaptığım birşey, çünkü yazılımcı olmamın yanısıra başka disiplinlerde bilgi edinme şansı buluyorum. Umarım sizinde başınıza böyle güzel birşey gelir. Büyük düşünmekten ve oynamaktan çekinmeyin!