Clean Code

Bölüm 1

“Bu kitabı iki sebepten ötürü okuyorsunuz: ilki yazılımcısınız, ikincisi daha iyi bir yazılımcı olmak istiyorsunuz. Güzel, çünkü daha iyi yazılımcılara ihtiyacımız var.” diye başlıyor *Clean Code*’a, usta yazılımcı Robert C. Martin, ve devam ediyor:
Bu kitap iyi programlamayı anlatıyor. Bir sürü kodlama örneği olacak. Bitirdiğimiz zaman ise iyi kod ve kötü kod arasındaki farkı anlayabileceğiz. Nasıl iyi kod yazabileceğimizi ve kötü yazılmış bir kodu iyi bir koda nasıl dönüştürebileceğimizi öğreneceğiz.

80'lerde bir şirket müthiş ve çok popüler bir uygulama yazdı. Ancak bir zaman sonra yeni sürüm çıkma (release) dönemleri uzamaya başladı. Bir sonraki sürümde hatalar çözülmemiş oluyordu. Yüklenme süresi uzuyor ve çökmeler artıyordu. Büyük bir hüsran ile uygulamayı kaldırdığım günü hatırlıyorum. Zaten bir zaman sonra da şirket tamamen piyasadan çekildi.
20 yıl sonra şirketin ilk çalışanlarından biri ile karşılaştım ve ona ne olduğunu sordum. Cevabı korkularımı doğruladı. Ürünü markete erkenden sürebilmek için çok acele etmiş ve kodda çok büyük bir kargaşaya sebep olmuşlardı. Daha fazla özellik ekledikçe, kod daha da kötü bir hal almış ve o kadar kötü hale gelmişti ki, artık kodu yönetemiyorlardı. Böylece kötü kod şirketin kapanmasına sebep olmuştu.
Hepimiz zaman zaman geri dönüp kodumuzu temize çekeceğimizi söylemişizdir. Ancak o zamanlar LeBlanc’ın şu kuralını bilmiyorduk: “Sonra asla demektir (Later equals never).”

Kod karmaşıklığı arttıkça takımların verimliliği düşer ve sıfıra yaklaşır. Verimlilik düştükçe de yöneticiler yapabildikleri tek şeyi yaparlar; verimliliği artırması umudu ile projeye daha çok insan kaynağı eklerler. *(İnsan kaynak mıdır yoksa değer mi?)* Takımdaki herkes verimliliği artırmak için büyük baskı altındadır. Öyle ki verimliliği sıfıra daha da yaklaştıracak şekilde kod karmaşası yaratmaya devam ederler.

Peki Koda Ne Oldu?

Ne oldu da iyi kod bu denli bir hızla kötü koda dönüştü? Bunun için bir çok sebep sıralayabiliriz. Gereksinimlerin (requirements) çok fazla değiştiğinden şikayet edebiliriz. Teslim tarihlerinin (deadline) çok sıkı olduğundan da yakınabiliriz. Beceriksiz yöneticilere ya da hoşgörüsüz müşterilere de püskürebiliriz. Ancak hata tamamen bizde. Bizler profesyonel değiliz!

Kabul etmesi zor, hata nasıl bizde olabilir? Diğerleri, onların hiç suçu yok mu? Hayır. Yöneticiler taahhüt vermek için bizden birşeyler duymayı beklerler. Beklemedikleri zaman bile onlara ne düşündüğümüzü söylemekten kaçınmamalıyız. Proje yöneticileri de zamanlama için bizden birşeyler duymayı beklerler. Bu yüzden, proje planlaması ve başarısızlıklar konusunda epey suçluyuz!

"Eğer yapamazsam kovulurum!" diyorsun fakat büyük ihtimalle kovulmayacaksın. Çoğu yönetici istiyormuş gibi yapmasa bile gerçeği ister. Çoğu yönetici iyi kod ister, zamanlama konusunda takıntılı olduklarında bile. Zamanı ve gereksinimleri tutkuyla savunabilirler, ancak bu onların işidir. Senin işin ise aynı tutku ile kodunu savunmaktır.

Temiz kod yazabilmek, temizlik (cleanliness) duygusuyla uygulanmış sayısız küçük teknik yöntemlerin disiplinli bir şekilde kullanımını gerektirir. Bazılarımız bu duyguya doğuştan sahiptirler. Bazılarımız ise elde edebilmek için savaşmak zorundadırlar. Bu duygu sadece iyi ya da kötü kodu ayırt etmemizi sağlamaz, aynı zamanda kötü kodu temiz koda (clean code) dönüştürebileceğimiz stratejiyi de bize gösterir. Bu duygudan yoksun bir yazılımcı karmaşık bir modüle baktığında karmaşıklığı tanır ancak onunla ne yapacağı hakkında en ufak bir fikri yoktur. Bu duyguya sahip bir yazılımcı ise bu karmaşık koda bakar ve seçenekleri görür.

Temiz Kod Nedir?

Yapılmış birçok tanımı var. Ben de çok iyi bilinen bazı yazılımcılara ne düşündüklerini sordum:

Bjarne Stroustrup (C++'ın mucidi): *Kodumun şık ve temiz olmasını seviyorum. Kodda mantık, hataların saklanmasını zorlayacak kadar düz; bağımlılıklar (dependency) bakımı kolaylaştıracak kadar minimal olmalı. Tüm istisnai durumlar (exceptions) ele alınmalı, performans optimale yakın olmalı.*

Grady Booch (Object Oriented Analysis and Design with Applications kitabının yazarı): *Temiz kod basit ve açıktır. Temiz kod, iyi yazılmış bir düzyazı gibidir. Temiz kod, asla tasarımcının niyetini gizlemez, daha çok berrak soyutlamalarla ve düz kontrol satırlarıyla doludur.*
Dave Thomas (OTI Labs'ın kurucusu): *Temiz kod, onu geliştiren yazılımcı dışında başka geliştiriciler tarafından da okunabilir ve iyileştirilebilir. Birim ve kabul testleri vardır. Anlamlı isimlendirmeleri vardır. Bir şeyin yapılması için tek bir yol vardır. Çok az bağlılığı vardır ve temiz bir API sağlar.*
Michael Feathers (Working Effectively with Legacy Code kitabının yazarı): *Temiz kod için bildiğim birçok özelliği sıralayabilirim; ancak bir tanesi diğer tüm özellikleri kapsıyor. Temiz kod her zaman ona değer veren biri tarafından yazılmış gibi görünür.*
Peki Robert C. Martin olarak ben ne düşünüyorum? *Bu kitap size tam da bunu anlatacak; bir değişken, sınıf ya da metod adının temiz olabilmesi için ne düşündüğümü yazacağım. Elbette bu kitaptaki önermelerin çoğu tartışmaya açık. Büyük ihtimalle bazılarına katılmayacaksınız, bazılarına şiddetle karşı çıkacaksınız. Sorun değil. Sadece şunu bilmelisiniz ki, bu yöntemleri onlarca yıllık tecrübeler sonucunda, birçok deneme ve yanılmamalarla öğrendim.*

Bölüm 2

Her şeye bir isim veririz; değişkenlerimize, fonksiyonlarımıza, argümanlarımıza, sınıflarımıza ve paketlerimize. O kadar çok isimlendirme yaparız ki, bunu iyi yapsak iyi olur aslında. Yazılımcılar kod yazarken problemleri kendileri yaratırlar. Örneğin aynı ismi aynı kapsamdaki (scope) 2 farklı objeye veremediğimizden, birini değiştirme yoluna gideriz. Sayı eklemek yeterli olmadığında, bunu birinin bir harfini eksilterek yaparız. Ancak, isimlerin farklı olması gerekiyorsa anlamları da farklı olmalıdır.
a1, ***a2***, ***a3***, ... gibi isimlendirmeler kesinlikle anlamlı değildir. Bu tür isimler yazarın amacı hakkında en ufak bir ipucu bile vermezler. Burada ***a1*** ve ***a2*** yerine ***source*** (kaynak) ve ***destination*** (hedef) kullanılması çok daha anlamlıdır:

Telaffuz Edilebilir İsimler Kullanın

Eğer bir değişkenin adını telafuz edemiyorsanız onu kullanmayın. Yakın zamanda karşılaştığım bir örnek: "yeterli" anlamında, ***adequate*** kelimesinin yerine, daha yaygın kullanılan ve dolayısıyla telaffuzu daha iyi bilinen ***enough*** kelimesi tercih edilmelidir.

Aranabilir İsimler Kullanın

Tek harfli isimler ya da sayı sabitleri arama dostu değildir. Örneğin ***MAX_CLASSES_PER_STUDENT*** sabitini çok rahatça bulabiliriz. Ancak *7*'yi bulmak oldukça zahmetlidir. Aynı şekilde ***e*** de çok kötü bir seçimdir. İngilizce'deki en yaygın harftir ve neredeyse her yerde karşımıza çıkacaktır.

Kişisel tercihim, tek harfli isimleri yalnızca kısa metotlarda lokal değişken olarak kullanmaktır. Bir ismin uzunluğu, o değişkenin kapsamının genişliği ile eşdeğer olmalıdır. Eğer bir değişken ya da sabit bir kod bloğunda birden fazla kullanılacaksa, arama dostu bir isim vermek en iyisidir.

Arayüzler ve Gerçekleştirimler (Interfaces — Implementations)

Bir arayüz ve onu gerçekleştirecek somut (concrete) bir sınıf yazacağınızı düşünün. Bu iki sınıfa ne isim verirdiniz? ***IShapeFactory*** ve ***ShapeFactory*** mi? Ben arayüz sınıflarını sade bırakmayı tercih ediyorum. Başa ***I*** harfi eklemek yaygın bir pratik, ancak dikkat dağıtmakta bir numara ve bilgi vermek konusunda da sonuncu. Eğer arayüz ya da gerçekleştirim sınıflarından birini belirtmem gerekiyorsa, gerçekleştirim sınıfını seçiyorum; ***ShapeFactoryImpl*** şeklinde. Arayüz sınıfını belirtmekten çok daha iyi.

Sınıf isimleri ***Customer***, ***WikiPage***, ***Account*** ya da ***AddressParser*** gibi isimlerden/isim tamlamalarından oluşmalıdır. Sınıf isimleri asla bir fiil olmamalıdır.

Kelime Oyunu Yapmaktan Kaçının

İki farklı amaç için aynı kelimeyi kullanmaktan ya da aynı amaçlar için farklı kelimeleri kullanmaktan kaçının.

Örneğin ***Controller***, ***Manager*** ya da ***Driver*** kelimelerini aynı

kapsamda farklı sınıflar için kullanmak iyi bir kullanım örneği değildir. Birini seçin ve onunla devam edin.
Örneğin birileri sizden önce ***add*** metodu yazmış olsun ve bu metot da iki değeri birbirine birleştiriyor (concat) olsun. Bizim de bir listeye değer ekleyen bir metota ihtiyacımız olsun. Bu metoda ***add*** mi demeliyiz? Hayır. Bu durumda yeni metodumuza ***insert*** ya da ***append*** demeliyiz. Yeni bir ***add*** metodu yazmak, kelime oyunu yapmaktır.

Gereksiz Bağlamlardan Kaçının

firstName, ***lastName***, ***street***, ***houseNumber***, ***city***, ***state*** ve ***zipcode*** isimli değişkenlerimiz olduğunu düşünelim. Birlikte alınca bir adresin detayları olduğunu çok çabuk anlayabiliyoruz. Ancak sadece ***state*** değişkenini görürsek, gene de adrese ait olduğunu düşünebilir miyiz?
Önekler (prefix) kullanarak bağlam (context) sağlayabilirsiniz; ***addrsFirstName***, ***addrLastName***, ***addrState*** vb. En azından okuyucular bu değişkenlerin daha büyük bir yapının parçası olduğunu anlayabileceklerdir. Elbette daha iyi bir çözüm ***Address*** isimli bir sınıf yaratmaktır.

Bölüm 3

Bir Metot Ne Kadar Kısa Olmalı?

Fonksiyonların ilk kuralı küçük, kısa olmaları gerektiğidir. İkinci kural ise daha da kısa olmaları gerektiğidir.

Switch İfadeleri

Kısa bir ***switch*** ifadesi yazmak zordur. Bir şey yapan ***switch*** de yazmak zordur. Doğaları gereği ***switch*** ifadeleri *N* tane şey yaparlar. Ve ne yazık ki, ***switch*** ifadelerinden her zaman kaçamıyoruz, ancak her ***switch*** ifadesinin alt seviye bir sınıfa gömüldüğünden ve tekrarlanmadığından emin olabiliriz. Elbette bunu polimorfizmle (çok biçimlilik) yaparız.

Açıklayıcı İsimler Kullanın

Yukarıdaki örneğimizde ***testableHtml*** fonksiyonumuzun ismini ***SetupTeardownIncluder.render*** olarak değiştirdim. Bu isim çok daha iyi, çünkü fonksiyonun yaptığı işi daha iyi tanımlıyor. ***private*** metotlarımızın isimlerine de aynı denklikte

tanımlayıcı isimler verdim; ***isTestable*** ya da ***includeSetupAndTeardownPages*** gibi...
Ward'ın prensibini hatırlayın: "Programın her parçası beklediğiniz gibi çıktığında, temiz kod üzerinde çalıştığınızı anlarsınız." Bu prensibi uygulamanın ilk adımı, tek bir şey yapan küçük fonksiyonlar için iyi isimler seçmektir. Küçük ve daha odaklı bir fonksiyon için açıklayıcı bir isim seçmek daha kolaydır.

Uzun isimler kullanmaktan kaçınmayın. Uzun tanımlayıcı isimler, kısa bilmece gibi isimlerden veya uzun bir yorumdan daha iyidir. İsim seçmek için zaman harcamaktan korkmayın. Bir sürü farklı isim deneyin ve her biri ile kodu tekrar tekrar okuyun. Açıklayıcı isimler seçmek zihninizdeki tasarımı ortaya çıkaracak ve onu geliştirmenize yardım edecektir.
İsimlerinizde tutarlı olun. Fonksiyon isimlerinizde aynı isimleri, kelime gruplarını, sıfatları ve fiilleri kullanın.

Argümanlar

Bir fonksiyon için ideal argüman sayısı sıfırdır. ***StringBuffer*** örneğimizi düşünelim. Onu bir örnek değişken (instance variable) olarak yazmak yerine argüman olarak da geçebilirdik. Ancak okuyucularımız değişkeni her gördüğünde tekrar tekrar yorumlamak zorunda kalacaklardı.
Argüman fonksiyon adından daha farklı bir soyutlama seviyesidir ve o noktada çok önemli olmayan bazı detayları bilmeye zorlar. Argümanlar test açısından daha da zorlar. Hiç argüman yoksa önemsizdir. Bir tane varsa çok zor değildir. İki argüman biraz daha zorlayıcıdır ancak iki argüman sonrası yıldırıcı olabilir.

Bir fonksiyona tek bir argüman geçmenin iki yaygın sebebi vardır. Bir argüman ile bir soru soruyor olabilirsiniz; ***boolean fileExists("My File")*** gibi... Ya da belki bu argümanı işleme sokuyor, onu bir şeylere dönüştürüyor ve ***return*** ediyorsunuzdur. Bu iki kullanım şekli kullanıcıların bir fonksiyonda görmek isteyebilecekleri kullanım şekilleridir.
Tek argümanın biraz daha az kullanım şekli ise ***event***'tir. Bu kullanım şeklinde girdi var ancak çıktı yoktur. Tüm program bu

fonksiyonun çağrısını bir ***event*** (olay) olarak yorumlar ve argümanı sistemin durumunu değiştirmek için kullanır. Örneğin; ***void passwordAttemptFailedNtimes(int attempts)***. Bu kullanımda oldukça dikkatli olunmalıdır ve okuyucuya bunun bir ***event*** olduğu açıkça belli edilmelidir. İsimler ve bağlamlar dikkatli seçilmelidir.

Boolean Argümanlar

Boolean argümanlar çirkindir. Fonksiyonlara parametre olarak geçmek ise korkunç bir pratiktir. Metodun imzasını karmaşıklaştırır ve "Bu fonksiyon birden fazla şey yapıyor." diye bağırır.
Argüman ***true*** ise bir şey yap,***false*** ise başka bir şey yap.

Nesne Argümanlar

Eğer bir fonksiyon 2 ya da 3 argümandan fazlasına ihtiyaç duyuyorsa, bu argümanlardan birkaçı bir sınıf ile sarmalanmalıdır.

Fiiller ve İsimler

Bir fonksiyon için iyi isim seçmek, fonksiyonun ve değişkenlerin niyetlerini açıklamak konusunda güzel bir başlangıç olabilir. Tek argümanlı (Monad) durumunda, fonksiyon ve argüman çok iyi bir fiil ve isim ikilisi olacak şekilde seçilmelidir.
Örneğin ***write(name)*** imzasına sahip bir fonksiyon oldukça açıklayıcıdır. Hatta ***writeField(name)*** bize ***name***'in bir alan (field) olduğunu ve bir yerlere yazılacağını söyler.
İkinci bir örnek
olarak ***assertEquals*** metodu, ***assertExpectedEqualsActual (expected, actual)*** olarak yazılsaydı çok daha iyi olabilirdi. Bu yazım şekli, argümanların sırasını hatırlamanın zorunluluğunu ortadan kaldırır.

Yan Etkiler Oluşturma (Side Effects)

Yan etkiler bizi kandırır. Fonksiyonunuz bir şeyi yapmaya söz verir ama yanında başka gizli şeyler de yapar. Bazen kendi sınıfının değişkenlerinde beklenmedik değişiklikler yapar. Bazen bunu fonksiyonlara geçilen parametreler ile ya da globaller ile yapar.

Bölüm 4

Kötü koda yorum yazmayın onu tekrardan yazın.
Yorumları yazmamızın amacı, kodda kendimizi iyi ifade edemediğimiz noktaları telafi etmektir. Yorumlar her zaman

koddaki kusurlarımızdır, çünkü kendimizi onlarsız nasıl ifade edebileceğimizi bilmiyoruz.
Bu yüzden eğer kendimizi yorum yazarken buluyorsak, durup tekrar düşünelim. Kendimizi açıklayabileceğimiz başka yollar olup olmadığına bakalım. Yorumlara neden bu kadar kötü bakıyorum? Çünkü yorumlar yalan söylerler. Her zaman ve bilerek değil, ancak çoğunlukla söylerler.

Yorumlarla Kötü Kodu Süslemeyin

Yorum yazmanın en yaygın motivasyonlarından biri de kötü koddur. Bir modül yazar ve onun kafa karıştırıcı, dağınık olduğunu biliriz. Bu yüzden kendimize *"Yorum yazsam iyi olacak."* deriz. Aslında, temiz yazsak iyi olacak.

Bir kaç yorumu olan temiz ve açıklayıcı kod, bir sürü yorumu olan karmaşık ve darmadağın bir koddan daha iyidir. Yarattığın karmaşayı açıklamak için yazacağın yoruma harcadığın enerjiyi, bu karmaşayı temizlemek için harcamalısın.

Bölüm 5

Kodunuzun düzgün formatta (biçimde) olmasına dikkat etmelisiniz. Kodunuzun formatını belirleyen bir takım kurallar benimsemeli ve o kurallara her zaman uymalısınız. Eğer bir takımda çalışıyorsanız, takım olarak belli kuralları kabul etmeli ve tüm üyeler bu konuda hem fikir olmalıdır.

Formatlamanın Amacı

Açık olalım; formatlama yani kodun biçimsel düzeni önemlidir. Kodu formatlamak iletişimle alakalıdır ve iletişim profesyonel bir geliştiricinin işinin ilk adımıdır. Belki de profesyonel bir geliştiricinin en önemli işinin "kodun çalışması" olduğunu düşündünüz. Umuyorum bu kitap, bu düşüncenizi değiştirmiştir. Bugün eklediğiniz yeni bir fonksiyonalite, muhtemelen bir sonraki sürümde tekrar değişecektir ve kodunuzun okunabilirliği, yapacağınız tüm değişiklikler için köklü bir etkiye sahip olacaktır. Kodlama tarzı ve okunabilirliği, orijinal kod değiştikten çok çok sonra da bakımını etkileyecektir. Disiplininiz ve kodlama tarzınız

orada durmaya devam edecektir. Peki, iletişim kurmamıza yardım eden en iyi formatlama konuları neler?

Dikey Formatlama

Gazete Metaforu

Örnek Değişkenler (Instance Variables)

Sınıfın en tepesinde tanımlı olmalıdır, ancak gene de bu durum örnek değişkenlerin dikey uzaklığını artırmamalıdır. Çünkü iyi tasarlanmış bir sınıfta örnek değişkenler sınıfın metotları tarafından defalarca kullanılırlar.

Bağımlı (Dependent) Fonksiyonlar

Eğer bir fonksiyon başka bir fonksiyonu çağırıyorsa, bu fonksiyonlar birbirine yakın olmalı ve eğer mümkünse çağıran çağrılanın üstünde olmalıdır. Bu durum programa doğal bir akış sağlar. Aşağıdaki koda bakalım; en üstteki fonksiyonun aşağıdaki fonksiyonu nasıl çağırdığına ve onun da diğerlerini nasıl çağırdığına bakın. Bu akış, çağırılan fonksiyonu bulmayı ve tüm modülün okunuşunu kolaylaştırıyor.

Bir satır ne kadar geniş olmalı?

Programcılar genelde kısa satırları tercih ederler. Benim kuralım ise bir satırda asla sağa kaydırma (scroll) yapma gereksinimi duymamak. Ancak bugünlerde monitörlerimiz çok geniş ve genç programcılarımız yazı tipini öyle küçük tutuyor ki, ekrana 200 karakter sığdırabiliyorlar. Bu iyi değil. Bir satır 120 karakterden uzun olmamalıdır.

Yatay Açıklık ve Yoğunluk

Yatay boşluğu, güçlü ilişki içerisinde olanları birleştirmek ve daha zayıf olanları ayırt etmek için kullanırız.

Takım Kuralları

Her programcının kendi formatlama kuralları vardır. Ancak bir takımda çalışıyorsa, orada takımın kuralları geçerli olur. Geliştiricilerden oluşan bir takım kendi basit formatlama tarzı üzerinde anlaşmalıdır ve takımın her bir üyesi bu tarzı kullanmalıdır. Yazılımımızın anlaşmazlık yaşamış bir grup insan tarafından yazılmış gibi değil de, tutarlı görünmesini isteriz.

2002'de bir projeme başladığımda, takımla beraber kodlama tarzımız üzerinde çalışmak için bir araya geldik. Ayraçları nereye koyacağımıza, sınıf, değişken, metot isimlerimizin nasıl olması gerektiğine karar verdik. Daha sonra bu kuralları geliştirme aracımızın (IDE) ***Code Formatter***'ına (geliştirme araçlarının sunduğu otomatik kod biçimlendirme araçları) tanımladık ve hep ona bağlı kaldık. Sadece benim tercihim değil, takımın da kararını verdiği kurallardı bunlar. Takımın bir üyesi olarak, ben de hep bu kuralları takip ettim.
İyi bir yazılımın, iyi okunabilen dokümanlardan oluştuğunu unutmayın. Tutarlı ve akıcı tarzları olmalı. Okuyucu bir dosyada gördüğü formatlama tarzının, diğer tüm dosyalarda da olduğunu düşünebilmeli.

Bölüm 6

Değişkenlerimizi ***private*** yapmamızın bir sebebi var: kimsenin onlara bağımlı (dependent) olmasını istemiyoruz. Gerçekleştirimlerini ya da tiplerini değiştirme özgürlüğünü elimizde tutmak istiyoruz. Peki, programcılar neden ***getter*** ve ***setter*** metotlarını otomatik olarak ekler ve değişkenlerini sanki ***public*** değişkenlermiş gibi açığa çıkarır?

Demeter Kuralı (Law of Demeter)

Demeter Kuralı der ki: Bir modül, değiştirdiği bir nesnenin içini bilmemelidir. Nesneler, verilerini saklar ve işlemlerini açarlar. Yani, bir nesne iç yapısını erişimciler aracılığıyla açmamalıdır.

Fonksiyon, izin verilen fonksiyonlardan herhangi biri tarafından dönen nesneler üzerindeki metotları çağırmamalıdır. Diğer bir deyişle: "Arkadaşlarınla konuş, yabancılarla değil (*talk to friends, not to strangers*)."

Melez Yapılar (Hybrids)

Yarısı nesne yarısı veri yapısı olan melez yapılar bazen karışıklığa sebep olur. Önemli işler yapan fonksiyonları, ***private*** değişkenleri ***public*** yapan erişimcileri (getters/setters) ve ***public*** değişkenleri vardır. Bu tür karmaşık

yapılar yeni fonksiyonlar eklemeyi zorlaştırır. Ve hatta yeni veri yapıları eklemeyi de zorlaştırır. Melez yapılar oluşturmaktan kaçınmalıyız. Bu tür yapılar, başka tiplerden ya da fonksiyonlardan korunma ihtiyacı olup olmadığından emin olunamayan sistemlerin göstergesidir.

Aktif Kayıt (Active Record)

Aktif kayıtlar ***DTO***'ların özel formlarıdır. ***Public*** değişkenleri olan veri yapılarıdır; ancak ***save*** ve ***find*** gibi yönlendirici metotları vardır. Bu aktif kayıtlar genellikle veritabanı tablolarından ya da diğer veri kaynaklarından doğrudan çeviridir. Fakat geliştiricilerin aktif kayıtları belli iş kuralları içerisine koyarak, onlara nesnelermiş gibi davranmaya çalıştıklarına sık sık denk geliyoruz. Çözüm ise elbette bir aktif kayda veri yapısıymış gibi davranmak ve iş kurallarını içeren ayrı nesneler yaratarak iç yapıyı saklamaktır.

Bölüm 7

Hata işleme (Error handling), kod yazarken yapmak zorunda olduğumuz şeylerden sadece biri. Bir şeyler yanlış gidebilir ve programımız patlayabilir. Biz programcılar olarak kodumuzu böyle durumlara karşı hazırlamaktan sorumluyuz.

İlk Önce Try-Catch-Finally Bloklarını Yazın

try-catch yazmak programınıza bir kapsam sağlar ve uygulamanın ***try*** bloğunda bir yerlerde hata alabileceğini ve ***catch***'de duracağını belirtirsiniz. ***catch*** blokları programınızı tutarlı bir durumda bırakmak zorundadırlar. Bu sebeple istisnalar fırlatabileceğiniz bir koda, ***try-catch-finally*** bloklarını yazmakla başlayın. Kullanıcının koddan ne beklemesi gerektiğini anlamasına yardım edecektir.

İstisnalarla Bağlam Sağlayın

Fırlattığınız her istisna, kaynağa ve hatanın yerine ait yeterli bilgi sağlamalıdır. ***Java***'da herhangi bir istisnadan bir iz bulabilirsiniz ya da bazen bu izler bize hiçbir şey söylemez. Bu nedenle fırlattığınız istisnalarda bilgilendirici hata mesajları verin. Başarısız olan işlemden, hatanın tipinden bahsedin. Eğer uygulamanızda loglama yapıyorsanız, ***catch*** bloğunuza bu hatayı loglayabilecek yeterli bilgiyi geçin.

Çağıranın İhtiyaçlarına Uygun İstisna Sınıfları Tanımlayın
Hataları sınıflandırabilmenin bir sürü yolu vardır; kaynaklarına göre, türlerine göre sınıflandırabiliriz.

Bölüm 8

Sistemlerimizdeki yazılımları nadiren kontrol ederiz. Bazen üçüncü taraf paketler satın alırız veya açık kaynak kullanırız. Diğer zamanlarda da şirketimizin diğer ekiplerinin bizler için yazdığı bileşenlere bağımlıyızdır. Ve bu yabancı kodları kodumuzla temiz bir şekilde birleştirmek zorunda kalırız.

Üçüncü Taraf Yazılım (Kod) Kullanmak

Bir arayüz kullanıcısı ile arayüz sağlayıcısı arasında doğal bir gerginlik vardır. Üçüncü taraf paketlerin ve çatıların (framework) sağlayıcıları, yazılımlarının geniş kitlelerce kullanılabilir olmaları ve bir çok ortamda çalışabilir olmaları için çabalarlar. Diğer bir taraftan kullanıcılar ise belirli ihtiyaçlara odaklanmış arayüzlerle çalışmak isterler. İşte gerginlik burada başlar.

Sınırları Keşfetmek ve Öğrenmek

Üçüncü taraf yazılımlar/çatılar daha kısa sürede daha fazla işlevsellik elde etmemize yardımcı olur. Ancak bu paketleri kullanmak istediğimizde test yazmak bizim için en iyi yol olabilir. Üçüncü taraf kütüphanemizi nasıl kullanacağımızın net olmadığını varsayalım. Doküman okumak ve onu nasıl kullanacağımıza karar vermek için bir veya iki gün (veya daha fazla) harcayabiliriz. Ardından, kodumuzu üçüncü taraf kodu kullanacak şekilde yazabilir ve düşündüğümüz şekilde çalışıp çalışmayacağını görebiliriz. Hatta çoğu zaman kendimizi, aldığımız hataların kodumuzda mı yoksa onların hataları mı olup olmadığını anlamaya çalışırken buluruz. Üçüncü taraf kodu öğrenmek ve entegre etmek zordur. İkisini birlikte yapmak daha da zordur. Üretim (production) kodumuzda yeni şeyler denemek yerine, üçüncü taraf kodu keşfetmek için bazı testler yazabiliriz. ***Jim Newkirk*** bu testleri, *öğrenme testleri* (learning tests) olarak nitelendiriyor.

*Öğrenme testleri*nde, üçüncü taraf API'yi uygulamamızda kullanmayı umduğumuz şekilde çağırırız. Aslında, bu API hakkındaki anlayışımızı kontrol eden kontrollü deneyler yapıyoruz. Testler, API'den ne istediğimiz üzerine odaklanır.

Bölüm 9

Mesleğimiz son yıllarda uzun bir yol kat etti. 1997'de hiç kimse ***Test Driven Development*** (TDD) kavramını bilmiyordu. Büyük çoğunluğumuzun birim testleri, programlarımızın çalışıp çalışmadığından emin olmak için yazdığı kodlardı. Özenle sınıflarımızı ve metotlarımızı yazar ve sonra bunları test etmek için bazı özel kodlar yazardık.

Bugünlerde, yazdığım her satır kodun beklediğim gibi çalıştığından emin olmak için test yazıyorum. Geçen testlerimi yazdıktan sonra ise, yazdığım kodla sonradan çalışacak herkesin de çalıştırmasına uygun olacak şekilde olduklarından emin oluyorum.

Agile ve TDD hareketleri birçok programcıyı otomatik birim testleri yazmaya teşvik etti ve her geçen gün daha fazla programcı bu hareketlere katılıyorlar. Ancak daha gidilecek çok yol var. Kodlama disiplinimize test yazmayı eklemekte acele edince, birçok programcı iyi testler yazmanın incelikli ve önemli noktalarından bazılarını kaçırdılar.

TDD'nin 3 Kuralı

Herkes ***TDD***'nin üretim kodundan önce birim testleri yazmamızı istediğini bilir. Ancak bu kural buzdağının sadece görünen kısmıdır:

- Geçmeyen bir birim testi yazmadan, üretim (uygulama) kodu yazmamalısın.
- Aynı anda birden fazla geçmeyen birim testi yazmamalısın. Derleme hatası da geçmeyen test demektir.
- O andaki geçmeyen testi geçirecek üretim kodundan başka üretim kodu yazmamalısın.

Bu üç yasa sizi otuz saniye uzunluğunda bir döngüye sokar. Testler ve üretim kodu birlikte yazılmışlardır. Sadece, testler üretim kodundan bir kaç saniye öncedir.

Bu şekilde çalışırsak, her gün düzinelerce, her ay yüzlerce ve her yıl binlerce test yazmış oluruz. Ve testlerimiz hemen hemen tüm üretim kodumuzu kapsar. Üretim kodunun boyutuna rakip olabilecek test kodu, yıldırıcı bir yönetim problemi oluşturabilir.

Testlerimizi Temiz Tutmak

Birkaç yıl önce, test koduna üretim koduyla aynı kalite ve standartlarda bakım yapılması gerekmediğine karar veren bir ekibe

koçluk yapmam istendi. Değişkenlerin iyi adlandırılması, test metotlarının kısa ve açıklayıcı olması gerekmiyordu. Test kodlarının iyi tasarlanmış ve üzerinde düşünülmüş bir şekilde bölünmesine de ihtiyaç yoktu. Testler çalıştığı ve üretim kodunu kapsadığı sürece sorun yoktu.
Hiç test olmamasındansa kirli testlerin olmasının daha iyi olduğunu söyleyebilirsiniz. Ancak bu ekibin farkında olmadığı şey, kirli testlerin bulunmasının -daha kötü olmasa da- hiç test olmaması ile eşdeğer olmasıydı. Sorun şuydu ki; üretim kodları geliştikçe testler de değişmeliydi. Ve testler ne kadar kirliyse, değiştirmesi de o kadar zor oluyordu. Üretim kodunu değiştirirken eski testler geçmemeye başlar, ve test kodundaki karışıklık bu testlerin tekrar geçirilmesini zorlaştırır. Böylece testler gittikçe artan bir yükümlülük gibi görülür.
Sürüm çıkardıkça testleri yazma maliyeti arttı ve nihayetinde geliştiricilerin tek şikayet ettiği şey haline geldi. Yöneticiler tahminlerin (estimations) neden bu kadar arttığını sorduğunda, testleri suçluyorlardı. Sonunda testleri tamamen çıkarmaya zorlandılar. Ancak testler olmadan, sistemlerinin bir kısmındaki değişikliklerin diğer kısımları bozmadığından emin olamazlardı. Bu nedenle hata (defect/bug) oranı artmaya başladı. Hata sayısı arttıkça da, üretim kodunda değişiklikler yapmaya çekinmeye başladılar. Çünkü iyilikten çok zararı olmasından korkuyorlardı. Üretim kodları çürümeye başladı. Sonunda elimize hiç testi olmayan, karışık ve hatalarla dolu üretim kodu, sinirli müşteriler ve test çabalarının başarısız olduğuna dair hisler kaldı.
Bir bakıma haklılardı. Onların test yazma çabaları başarısız olmuştu. Ancak bu başarısızlığın sebebi, testlerin dağınık olmasına izin verme kararlarıydı. Testlerini temiz tutmuş olsaydılar, test yazma çabaları başarısız olmazdı.

F.I.R.S.T Kuralı

Temiz testler, F.I.R.S.T kısaltmasını oluşturan şu beş kuralı takip eder:

F

ast: Testler hızlı olmalıdır. Testler yavaş çalıştıklarında, sık sık çalıştırmak istemezsiniz. Testleri sık sık çalıştırmazsanız, problemleri kolayca giderecek kadar erkenden fark edemezsiniz.

I

ndependent: Testler birbirine bağlı olmamalıdır. Bir test, bir sonraki testin koşullarını belirlememelidir. Her bir testi bağımsız olarak ve istediğiniz herhangi bir sırada çalıştırabilmelisiniz. Testler birbirine bağımlı olduğunda, hata alan ilki, hiyerarşide aşağılara doğru hatalara sebep olarak ilk hata alınan yerin tespitini zorlaştıracaktır.

R

epeatable: Testler herhangi bir ortamda çalışabilir olmalıdır. Birim testlerini üretim ortamında, QA ortamında ve dizüstü bilgisayarınızda veya trende evinize gidiyorken çalıştırabilmelisiniz. Testleriniz herhangi bir ortamda tekrarlanabilir değilse, başarısız olmalarına hep bir mazeretiniz olacaktır.

S

elf-Validating: Testler ya geçerler ya da başarısız olurlar. Testlerin geçip geçmediğini anlamak için bir ***log*** dosyasına bakmamıza gerek olmamalı veya elle iki farklı metin dosyasını karşılaştırmanız gerekmemelidir.

T

imely: Testlerin zamanında yazılması gerekir. Birim testleri, yazıldığı üretim kodundan hemen önce yazılmalıdır. Testlerinizi kodunuzu yazdıktan sonra yazarsanız, üretim kodununun test edilmesi zor olabilir. Kodunuzu test edilebilecek şekilde tasarlayamayabilirsiniz.

Bölüm 10

Kapsülleme (Encapsulation)

Değişkenlerimizi ve ***util*** fonksiyonlarımızı gizli tutmak isteriz, ancak bazen bir değişkeni veya ***util*** metodu testlerden erişilebilmesi için ***protected*** yapmamız gerekebilir. (Uncle Bob, bir metodun testinin olmasının o metodun kapsüllenmesinden daha önemli olduğuna vurgu yapıyor.)

Sınıflar Küçük Olmalıdır

Sınıf tasarımı konusundaki ilk kural sınıfların küçük olmaları gerektiğidir. Buradaki sorumuz, ne kadar küçük?

Fonksiyonların boyutuna, satır sayılarına bakarak karar verdik; ancak sınıfların boyut ölçümünün daha farklı bir birimi vardır: sorumluluklarının sayısı.

Birbirine Bağlılık (Cohesion)

Sınıflar az sayıda örnek değişkene (instance variable) sahip olmalıdır. Bir sınıfın her bir metodu, bu değişkenlerden bir veya daha fazlasını değiştirmelidir. Her bir değişkenin her bir metot tarafından kullanıldığı bir sınıf, maksimum düzeyde birbirine bağlılığa sahiptir.

Değişimden İzole Etme

İhtiyaçlar değişecek, bu nedenle kod da değişecektir. Gerçekleştirimleri içeren somut (concrete) sınıflarımız, gerçekleştirimleri sadece sunan soyut (abstract) sınıflarımız olduğunu biliyoruz. Somut sınıflarımıza bağımlı istemcilerimiz varsa, bu detaylar değiştiğinde bu istemciler de risk altındadır. Bu etkiyi yalıtmak için arayüzleri (interface) ve soyut sınıfları kullanabiliriz.

Bölüm 11

Bağımlılık Enjeksiyonu

Kontrolü tersine çevirme (*Inversion of Control — IoC*) yönteminin bağımlılıkların yönetimi (dependency management) için bir uygulaması olan Bağımlılık Enjeksiyonu (*Dependency Injection — DI*), nesnelerin oluşumunu kullanımdan ayırmada güçlü bir mekanizmadır. ***IoC*** ikincil sorumlulukları bir nesneden -o işe adanmış- başka nesnelere taşır, ve böylece ***Single Responsibility*** prensibi desteklenmiş olur. Bağımlılıkların yönetimi konusu özelinde düşünecek olursak, bir nesne kendi bağımlılıklarının örneklerini oluşturma sorumluluğunu almamalıdır. Bunun yerine, bu sorumluluğu başka bir yetkili mekanizmaya, ***IoC*** / ***DI*** mekanizmasına bırakmalıdır.
Sistemler de temiz olmalıdır. Kötü bir mimari iş mantığını (business logic) belirsizleştirir ve çevikliği (agility) olumsuz etkiler. İş mantığı gizlendiğinde, hataların bulunması ve yeni özelliklerin eklenmesi zorlaşır. Çeviklik azalırsa, üretkenlik azalır ve ***TDD***'nin faydaları kaybolur.

Soyutlamanın her seviyesinde niyet açık olmalıdır. Sistemler veya bağımsız modüller tasarlarken, çalışabilecek en basit çözümü uygulamayı unutmayın.

Bölüm 12

İyi tasarıma sahip sistemler oluşturabilmek için takip edebileceğiniz 4 kural olduğunu söylesem? Bu kuralları izleyerek, kodunuzun yapısını ve tasarımını anlayarak, ***SRP*** ve ***DIP*** gibi ilkelerin uygulanmasını kolaylaştıracak olsanız?

Kent Beck'in dört basit tasarım kuralı, iyi tasarlanmış yazılımlar yaratmada önemli bir etkendir. Ve Kent Beck'e göre eğer bir tasarım basitse, şu kuralları takip etmelidir:

- Tüm testleri çalıştırın
- Tekrarlanmış kodlar yazmayın
- Açıklayıcı olun
- Sınıf ve metot sayısını en aza indirin

Kurallar önem sırasına göredir. Şimdi sırasıyla bu kuralları inceleyelim:

Kural #1: Tüm testleri çalıştırın

Bir tasarım, amaçlandığı gibi hareket eden bir sistem üretmelidir. Bir sistem kağıt üzerinde mükemmel bir tasarıma sahip olabilir ancak sistemin amaçlandığı gibi çalıştığını doğrulamanın basit bir yolu yoksa, kağıt üstündeki tüm çabalar boşa olabilir.

Kapsamlı bir şekilde test edilen ve tüm testlerini her zaman geçiren bir sistem ***test edilebilir bir sistem***dir. Bir sistemin ***test edilebilir*** olması önemlidir. Test edilebilir olmayan sistemler doğrulanabilir de değildir. Doğrulanamayan bir sistem asla dağıtılmamalıdır (deploy).

Sistemlerimizi test edilebilir hale getirmek, sınıflarımızın küçük ve tek amacının olduğu bir tasarıma sahip olmamıza yardımcı olur. ***SRP***'ye uyan sınıfları test etmek daha kolaydır. Daha çok test yazdıkça, test etmesi daha kolay olan sistemler ortaya çıkarmaya devam ederiz. Dolayısıyla sistemimizin test edilebilir olduğundan emin olmak, daha iyi tasarımlar oluşturmamıza yardımcı olur.

Sıkı bağımlılıklar da (tight coupling) test yazmayı zorlaştırır. Daha çok test yazdıkça; ***DIP*** gibi ilkeleri ve bağımlılık enjeksiyonu (dependency injection), arayüzler ve soyutlamalar gibi araçları daha çok kullanırız. Tasarımlarımız daha da gelişir.

Bir kere testlerimizi yazdıktan sonra, kodumuzu ve sınıflarımızı temiz tutmak için kendimizi yetkin hissederiz. Bunu, kodumuzu sürekli yeniden yapılandırarak yaparız. Eklediğimiz birkaç kod satırı için durup yeni tasarım üzerine düşünürüz. Testlerimiz geçtiğindeyse hiçbir yeri bozmadığımızdan emin olabiliriz. Testlere sahip olmamız, kodu temizlerken onu bozma korkumuzu ortadan kaldırır.
Düzenlemelerimizi yaparken, temiz tasarım hakkında tüm bildiklerimizi uygularız: birbirine bağlılığı (cohesion) artırırız, bağımlılığı (coupling) azaltırız, meselelerin ayrımını (seperation of concerns) sağlarız, sistemlerimizi modülarize ederiz, fonksiyon ve sınıflarımızı küçültür ve daha iyi isimler seçeriz. Burası ayrıca basit tasarımın son üç kuralını uyguladığımız yerdir: tekrarları kaldır, kodunun açıklayıcı olduğundan emin ol, sınıf ve metot sayılarını en aza indir.

Kural #2: Tekrarlanmış kodlar yazmayın

Tekrarlanmış kodlar, iyi tasarlanmış sistemlerin birinci düşmanıdır.

Kural #3: Açıklayıcı olun

Bir yazılım projesinin maliyetinin çoğu uzun dönem bakımıdır. Kodumuzu değiştirirken hata potansiyelini en aza indirgemek için, sistemin ne yaptığını anlamamız önemlidir. Sistemler daha karmaşık hale geldikçe, yazılımcılar olarak anlamamız zorlaşır ve hata yapma riskimiz artar. Bu nedenle, kodumuz yazanın niyetini açıkça belli etmelidir. Yazar kodu daha da açık hale getirdikçe, diğerlerinin anlaması için geçen süre ve bakım süresi daha da azalır. İyi isimler seçerek kendinizi daha iyi ifade edebilirsiniz. Ayrıca küçük fonksiyon ve sınıfların isimlendirmesi, anlaşılması ve yazılması daha kolaydır.
Standart bir terminoloji kullanarak da kendinizi ifade edebilirsiniz. Örneğin tasarım desenleri (design patterns) büyük ölçüde iletişim ve ifade etme ile alakalıdır. Bu desenleri gerçekleştiren sınıfların isimlerinde bu tür standart desen isimleri kullanarak, ***Command*** veya ***Visitor*** gibi, tasarımınızı diğer geliştiricilere kısaca açıklayabilirsiniz.

İyi yazılmış birim testleri de açıklayıcıdır. Testlerin öncelikli amacı, *örnek* dokümantasyon sağlamaktır. Testlerimizi okuyan biri sınıfın neyle ilgili olduğunu hızlı bir şekilde anlayabilmelidir. Çoğu zaman kodumuzun çalışmasını sağlıyor ve bir sonraki kişinin kodu okumasını kolaylaştırmak için hiç çabalamadan bir sonraki soruna geçiyoruz.

Kural #4: Sınıf ve metot sayısını en aza indirin

Sınıflarımızı ve metotlarımızı küçültmek için çabalarken, küçük küçük bir çok sınıf ve metot yaratabiliriz. Bu kural ise bu sayıyı minimumda tutmamız gerektiğini söylüyor.
Yüksek sayıda sınıf ve metot bazen anlamsız dogmatikliğin bir sonucudur. Örneğin, her sınıf için bir arayüz oluşturmayı ısrarla vurgulayan bir kodlama standardını veya alanların (fields) ve davranışların her zaman veri sınıflarına ve davranış sınıflarına ayrılması gerektiğinde ısrarcı olan geliştiricileri düşünelim. Bu tür dogmalara karşı direnilmeli ve daha pragmatik bir yaklaşım benimsenmelidir.
Bu kuralın, 4 kuraldan en düşük öncelikli olanı olduğunu unutmayın. Bu yüzden sınıf ve fonksiyon sayısını düşük tutmak önemli olsa da, testler yazmak, tekrarları ortadan kaldırmak ve kendimizi açıkça ifade etmek daha önemlidir.