**CLEAN CODE**

Yazımıza Clean Code A Handbook of Agile Software Craftsmanship kitabının girişinden clean codingin önemini anlatan bir yazı ile başlayalım .

“ 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) .””

“Yazılım ne kadar iyi tasarlanmış ve sade oluşturulursa, değişiklik yapmak o kadar kolay ve ucuz olur.”

Clean Code , kod yazarken takip edilen bir felsefedir. Kodunuzun tam ve mümkün olduğunca kusursuz olması gerektiği fikridir . İhtiyaç olandan daha fazla kodunuzun olmamasıdır . Gereksiz hiçbir satır olmamalı . Kodlama yaparken yalnızca sorunu çözmek yerine , mümkün olduğunca verimli , okunabilir ve bakımı yapılabilir kodlama tarzıdır . Clean Codu 4 ana başlıkta inceleyeceğiz .

1. İSİMLENDİRME :

İsimlendirme konusu kulağa çok kolay gibi gözükse de bu konu yazılım geliştirmenin en önemli noktalarındandır . Anlamsız veya kısaltma şeklinde verilen isimler bir süre sonra kodun okuna bilirliğini azaltarak kodu anlayacak kişinin zamanını çalar . Aynı şekilde anlamlı ve doğru verilen isimler yazılımcının işini inanılmaz derecede kolaylaştırabilir . Normal şartlarda , bir fonksiyonu ismiyle değişkenleriyle yapılan işlemleri ile yukardan aşağıya bir kitap okur gibi okuyup anlamlandırabiliyor olmamız gerekir . Bu tip kodlar yorum satırlarına da nadir olarak ihtiyaç duyarlar ve self-documented kod olarak isimlendirilirler .

Örnek olarak aşağıdaki koda inceleyelim :

* int u; // gün cinsinden geçen süre

**u** burada hiçbir şeyi açıklamıyor, günlerle ya da zaman ile alakalı hiçbir şey uyandırmıyor . Daha anlamlı isimler seçmeliyiz , şunlar gibi :

* int gunCinsindenGecenSure;
* int gecmisGunSuresi;
* int gecenSureninGunDegeri;

1. KOŞUL KULLANIMI :

Bir değeri koşul içerisinde tekrar true ya da false’a eşitlemek , gerek olmayan bir kalabalık yaratacaktır . Koşul cümlelerinde ters koşul kullanmak da kodu okuyanlar için kafa karışıklığı yaratmaktadır . Bu yüzden elimizden geldiğince pozitif anlamlı koşullar üzerinden ilerlemeliyiz . Bir grup seçenek arasından seçim yapacaksak bunu static olarak vermektense enumlara bağlayıp kullanmak daha yerinde olacaktır . Böylece kodda ilgili bir opsiyonu değiştirmek istediğimizde , o opsiyonun kullanıldığı yerleri tek tek bulup değiştirmek yerine , ilgili enum opsiyonunu değiştirmemiz yeterli olacaktır . Kodun içerisinde hiçbir anlam ifade etmeyen sayı , string gibi ifadeleri de direkt olarak kullanmamamız gerekir . Değişken çok yerde kullanılıp belli aralıklarla değiştirilmeyecek ise database’de tutmak yerine aşağıdaki şekilde bir tanımlama koda anlam katacaktır . Kompleks koşullar okunurken kafa karışıklığı yaratır . Her ifade ve bunu içerisinde barındıran her koşul aslında cevabı true ya da false olan bir soruya cevap verdiğidir ve bu sorunun cevabına göre koşul değerlendirilir . Bunu göz önünde bulundurarak elimizdeki kompleks koşulların hangi soruya cevap verdiğine istinaden bir değişkene atamamız veya fonksiyonlaştırmamız işimizi oldukça kolaylaştıracaktır . Yazılımcılar olarak kodları çok sevsek de bazen her şeyin cevabı kod olmayabiliyor . Elimizde , belli değerlere karşı belli sonuç döndüren bir yapı olduğunu varsayalım . Bunu hard-coded olarak kod içerisinde tutmaktansa , database’de bir tabloda json formatında veya key value şeklinde tutup ilgili key’e ilgili value’yi dönen bir fonksiyon üzerinden yürüyebiliriz . Validasyonları iç içe if kullanarak yapmak çoğu yazılımcının düştüğü bir yazım yanlışıdır . Bu şekilde bir validasyon zincirinin okunması ve aynı zamanda hangi if’in hangi else’e denk geldiğinin anlaşılması zordur . Bunun yerine if’leri alt alta sıralayıp ilk yakalanan validasyondan itibaren kodu return etmek daha doğrudur .

Örnek olarak aşağıdaki koda inceleyelim :

Kodun ilk hali aşağıdaki gibidir . Müşteriden yeni bir istek gelir , SUCCESS durumunda yeni bir kontrol eklenmelidir . Yazılımcı da en kolay nereye eklerim diye düşünüp SUCCESS içine yeni kodu ekleyecektir .

* İf (user\_result == SUCCESS) {

Reply.WriteErrors(“”);

} else {

Reply.WriteErrors(user\_result);

}

Reply.Done();

Derin bir şekilde içi içe geçmiş kodun anlaşılması zordur. Biz çok da zor olmayan bir örnek verelim.

* if (user\_result == SUCCESS) {

**if (permission\_result != SUCCESS) {**

**reply.WriteErrors("error reading");**

**reply.Done();**

**return;**

**}**

reply.WriteErrors ("");

} else {

reply.WriteErrors(user\_result);

}

reply.Done();

Koyu kısım sonradan koda eklenmiş gibi duruyor . Kodu takip ederken, user\_result ve permission\_result gibi iç içe yer alan değişken değerlerini takip etmek gerekiyor . Şimdi kodu fonksiyondan erken dönme ile iyileştirelim . Bunu da , “hatalı durumlar” ile ilk başa çıkarak yapabiliriz .

* if (user\_result != SUCCESS) {

reply.WriteErrors(user\_result);

reply.Done();

return;

}

if (permission\_result != SUCCES) {

reply.WriteErrors(permission\_result);

reply.Done();

return;

}

reply.WriteErrors ("");

reply.Done();

Yeni kodumuz sadece bir seviye içerde ve anlaşılması da çok daha kolay.

1. DÖNGÜ KULLANIMI :

Fonksiyonların ilk kuralı küçük, kısa olmaları gerektiğidir. İkinci kural ise daha da kısa olmaları gerektiğidir. Bir fonksiyon 100’lerce satırdan oluşmamalıdır en fazla 20 satırda fonksiyonu tamamlarsanız kodu okuyacak kişi için çok iyi olur . Dry (dont repeat yourself) çoğu yazılımcının duyduğu bir kod prensibidir. Bir fonksiyonelliği kodun birden fazla kısmında tekrar eden bir şekilde yazdıysanız ve daha da kullanmaya ihtiyaç duyabileceğiniz potansiyelde bir kod ise bu kodu metotlaştırmanız gerekmektedir demek . Eğer fonksiyonunuz satır sayısı gitgide artıyorsa ve siz de hala yazmaya devam ediyorsanız , orada durup bu fonksiyonu daha küçük parçalara nasıl ayırırım diye düşünmeniz gerekir . Her fonksiyon yalnızca bir işi gerçekleştirmeli ve bunu da en iyi şekilde gerçekleştirmelidir . Fonksiyonu yazan yazılımcı da onu ileride çağıracak olan yazılımcı da o fonksiyonu tek ve spesifik bir amaç için çağırmalıdır . Sonuçta önceden yazılmış bir fonksiyonu kullanırken tekrar tekrar kodlarını kontrol edemeyiz ve kimse adından yola çıkarak çağırdığı bir fonksiyonun arkada başka bir şeyler yapmasını istemez . Koşul ifadelerinin içerisindeki koşulları programatik olarak çoğu zaman anlarız ancak neden o şekilde yazıldığını tam olarak anlayamayız . Clean kod yazarken kullandığımız dilin bize verdiği nimetlerden de yararlanmalıyız . Bir fonksiyonda çok fazla parametre olması sıkıntılı bir durumun belirtisi olmakla beraber fonksiyonun yapması gerekenden fazla iş yaptığının da habercisi olabilir . Fonksiyon yazarken parametre sayısını mümkün oldukça 3’ten az tutmaya gayret etmeliyiz . “Eğer bir fonksiyon sadece isminde belirtilen adımları yapıyorsa, o fonksiyon bir şey yapıyordur.”

1. YORUM KULLANIMI :

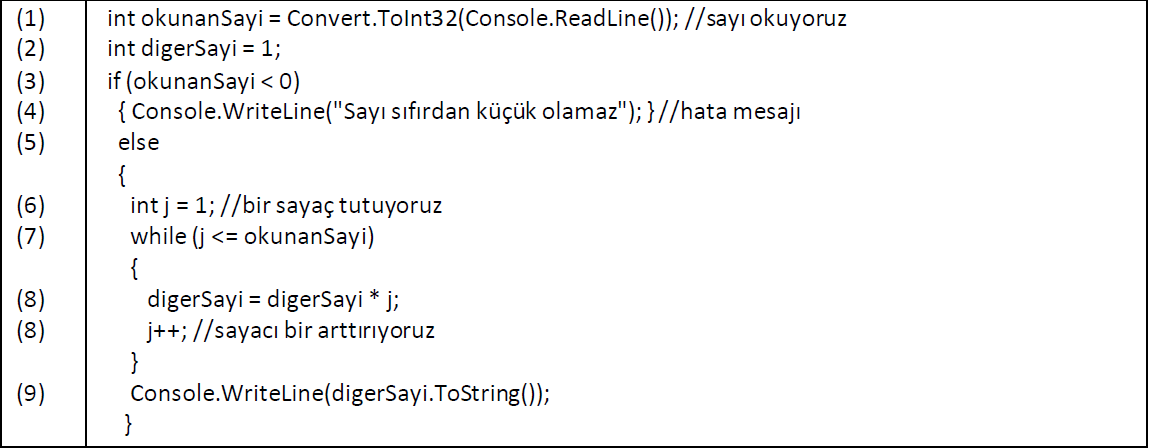
İyi yazılmış ve güzel yerleştirilmiş bir yorum satırı kodu okuyacak kişi için çok faydalı olabilecek iken kötü ve düşünülmeden yazılmış yorum satırları bütün kodu bir çöplük haline getirebilir . Çünkü yorum satırlarının %90’ının dayandığı sebep bellidir , oda kötü kodu gizlemektir . Yorum satırı yazmaya , kötü ve ne yaptığı belli olmayan fonksiyon ve kod parçacıklarını açıklamak için yöneliriz . Bu da kod için oldukça tehlikeli bir durumdur . Neden yorum satırları kod açısından bu kadar sıkıntılıdır derseniz , yorum satırları genelde yalan söyler . Kodlar değişir ve gelişir ancak yorum satırları korunmaz . Yorum satırı yazmaya harcadığımız zamanı kodu temizlemeye harcamalıyız . Yorum satırını da artık kodun yetersiz kaldığını düşündüğümüz durumlarda kullanmalıyız . Zombie kod , kullanılmayan kod parçacıklarının yorum satırı haline alınıp kod içesinde tutulmasıdır ki diğer developerlar tarafından fazlaca kafa karışıklığına yol açar . Developerin kafasında yorum satırındaki kod neden vardı ? , yanlışlıkla mı yoruma alındı da unutuldu ? , kim yaptı ? , niye yaptı ? , ileride bu koda ihtiyaç olucak mı yoksa artık tamamen işlevsiz mi ? gibi sorular oluşturarak kafa karışıklığı yaratır ve okunurluğu azaltır .

Örnek olarak aşağıdaki kodu inceleyelim :

* int i=1; //sets i 1

Buradaki yorum satırına gerek yoktur mesela .

Şimdi gelin Clean Codu kavramını bir örnekle pekiştirelim .



Olumsuz koşullu else’e yazmak daha iyi olurdu (Else ile if’in yerlerini değiştirmek) . Yorum satırlarının çoğu zaten kodun ne yaptığı anlaşılan yerlere yazılmış (Yazılmasına gerek yoktu çoğunun) . J yerine daha güzel bir isimlendirme yapılabilirdi . While yerine for döngüsü kullanmak daha iyi olabilirdi .

KAYNAKÇA :

* [https://github.com/JuanCrg90/Clean-Code-Notes](https://github.com/JuanCrg90/Clean-Code-Notes%20)
* <https://blog.kmk.net.tr/clean-code-nedir>
* <http://cagataykiziltan.net/tr/clean-code/>
* [https://medium.com/@busrauzun/clean-code-kitabindan-notlar-1-temiz-kod-derken-44e6f7a27eb0](https://medium.com/@busrauzun/clean-code-kitabindan-notlar-1-temiz-kod-derken-44e6f7a27eb0%20)