

# Temiz Kod (Clean Code)

***1. Bölüm - Temiz Kodun Temelleri***



Eğitmen:

**Akın Kaldıroğlu**

Çevik Yazılım Geliştirme ve Java Uzmanı

# Konular



- **Temiz Kod Nedir?**
- **Temiz Kod Çerçevesi**
- **Basitlik**
  - Basit Kod
  - Anlaşılır Kod
  - Mimariye Uyum
  - İsimlendirme
- Standartlar
- Dilin Kullanımı
- Dokümantasyon
- **Odaklılık**
- **Tamlık**
- **Doğruluk**

# Temiz Kod Nedir?



- Hanlon's razor der ki:

**Never attribute to malice that which is adequately explained by stupidity.**

**Aptallıkla açıklanabilecek şeyi kötü niyete yormayın.**

- Johann Wolfgang von Goethe de şöyle der:

**..misunderstandings and neglect create more confusion in this world than trickery and malice. At any rate, the last two are certainly much less frequent.**

**.. yanlış anlamalar ve ihmal, kurnazlık ve kötü niyetten daha fazla karışıklık çıkarır. Her halde, son ikisi kesinlikle çok daha seyrektiler.**

# Clean Code - Temiz Kod Nedir?



- Clean Code ya da Temiz Kod nedir?
- Nasıl tanımlarsınız?
- Ya da kodun hangi özellikleri onu temiz yapar?
- Ya da kodun hangi özellikleri onu kirli/kötü yapar?
- Siz kişisel olarak kod yazarken temiz olması adına nelere dikkat ediyorsunuz?
- Kurumunuzda temiz kod yazma yönünde ne gibi yönlendirmeler ya da önlemler var?

# Clean Code Nedir? - I



- **Bjarne Stroustrup:** Şık ve etkin (elegant and efficient)
- **Grady Booch:** Basit ve doğrudan (simple and direct)
- **Michael Feathers:** Dikkatli, önem veren birisi tarafından yazılmış görünür (it looks like it was written by someone who cares)
- **Ron Jeffries:** Küçük, ifade gücü yüksek, basit ve tekrarsız (small, expressive, simple, and no duplication)
- Bu ifadeler açıklanmaya muhtaçtır,örneğin “basit ve doğrudan” o kadar da basit ve doğrudan değildir, ya da “şık kod” nedir?.

# Clean Code Nedir? - II



**Temiz kod, orijinal yazarından başka developer tarafından okunabilir ve geliştirilebilir.** Birim ve kabul testlerine sahiptir. Anlamlı isimleri vardır. Bir şeyi yapmanın pek çok yolundan ziyade tek bir yolunu sağlar. Açık-seçik olarak tanımlanmış minimal bağımlılıklara sahiptir ve **temiz ve minimal bir API** sunar. Kod okunabilir olmalıdır, çünkü sadece programlama diline bağlı kalınırsa, her türlü gerekli bilgi kodda açık bir şekilde ifade edilemez.

**Dave Thomas**

# Bence “Temiz Kod” - I



- Kodlama (programlama) aslen karmaşık ve zordur ve aslolan geliştirmek değil değiştirmektir (bakım), dolayısıyla değişimde kod geliştirmek daha da zordur.

**Temiz Kod (Clean Code), zaten zor olan programlama sürecini, insanı zaafiyet ve kaprislerle daha da zorlaştırmamaktır, kolaylaştırır.**

**Temiz Kod (Clean Code), rahat anlaşılır ve kolay değişimdir koddur. Temiz kod, odaklıdır dolayısıyla da kısaltır ve bağımlılıkları azdır.**

# Bence “Temiz Kod” - II



- **Clean code, teknolojiden, sektörden bağımsız olarak, temiz, anlaşılır, yalın, kaliteli koddur.**
- Kod temiz değilse kötüdür (**dirty kod**), karmakarışıkır (**Spaghetti Code**) **Code Smell, Kötü Koku,**'ya sahiptir.
- Kötü kokuları öğrenmek, temiz koda ulaşmak için önemlidir.
- **Clean/Dirty Code, bir teknoloji problemi değildir.**
- **Clean/Dirty Code, bir ahlak ve bilgi problemidir.**
- **Clean/Dirty Code, işini iyi yapmakla ilgilidir.**

# Temiz Kod - Teknik Borç - I

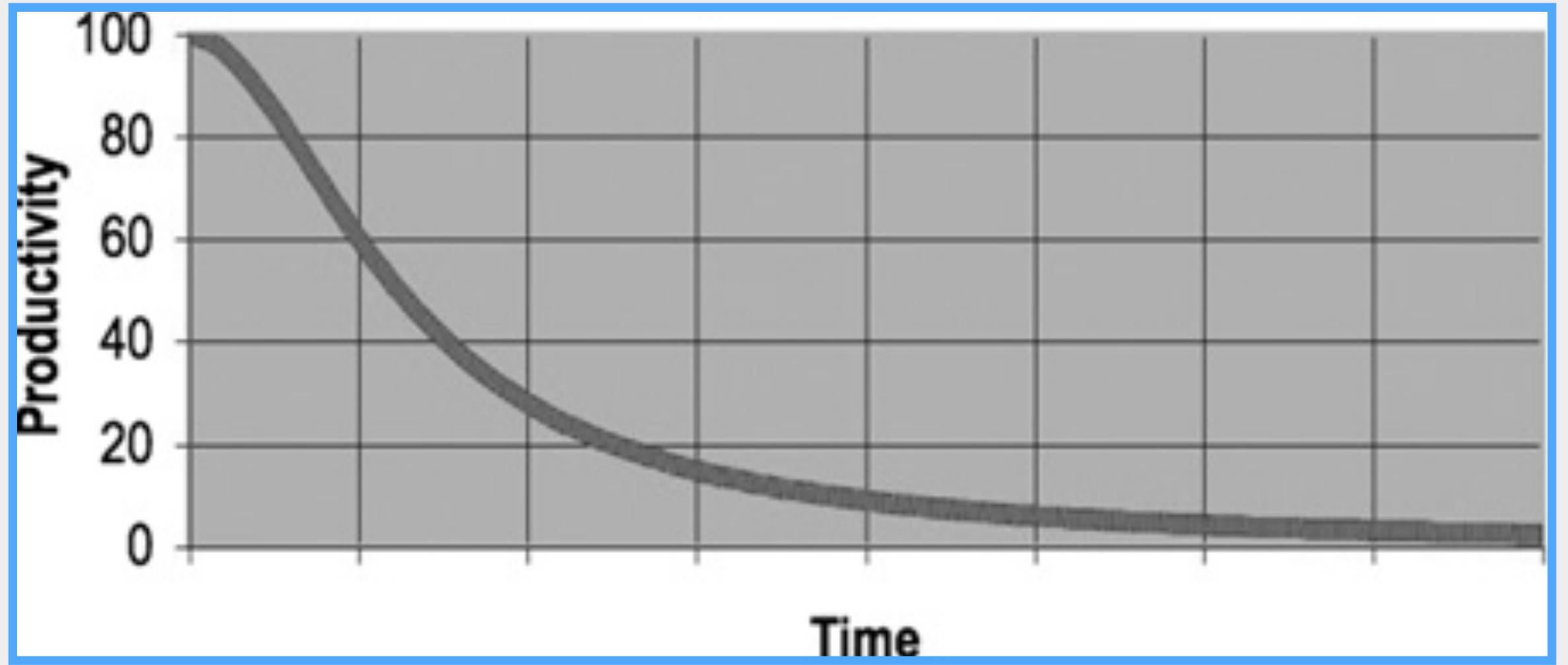


- Temiz yazılmayan kod, karmakarışıkır (mess).
- Temiz yazılmayan kod, teknik borç (technical debt) biriktirir.
- Teknik borcu yüksek kodun bakımı zordur (low maintainability).
- Teknik borcu yüksek kod, zor anlaşılır dolayısıyla doğruluğunu ya da yanlışlığını belirlemek de zordur, değiştirilmesi de maliyetlidir.
- Temiz yazılmayan kodun oluşturduğu karmakarışıklık çığ gibi büyür.

# Temiz Kod - Teknik Borç - II



- Yazılımda üretim hızını etkileyen pek çok faktör vardır:
  - İş ve teknik mimari,
  - Yetkin yazılımcılar, vs.
  - Temiz olmayan kod da, yeni kodun yazılmamasında güçlük çıkartarak, üretim hızını düşüren önemli bir etkendir.



# Temiz-Yalın-İyi-Kaliteli Kod



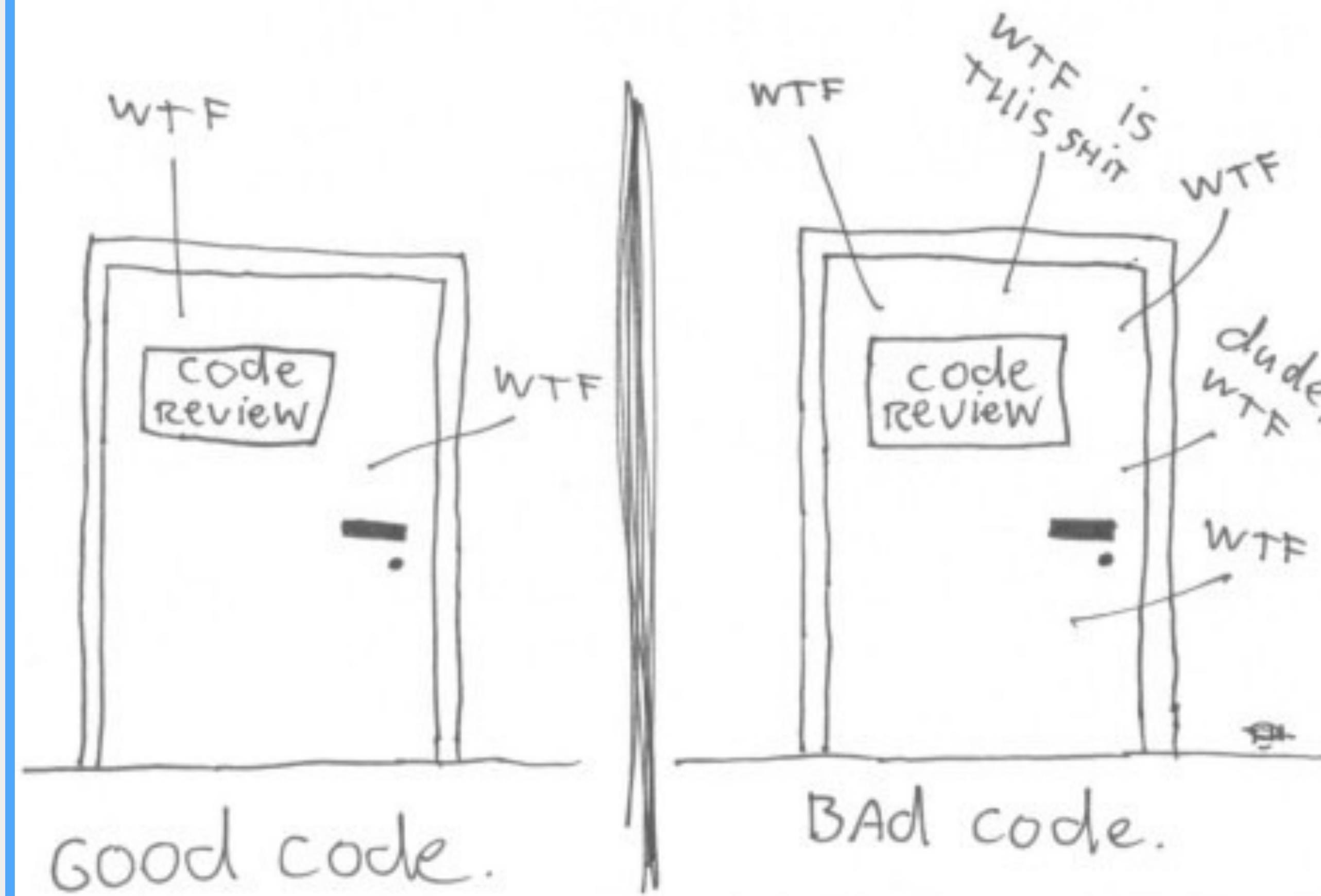
- Clean Code'u dilimizde temiz kod yanında
  - Yalın kod
  - İyi kod
  - Kaliteli kod
- olarak da çevrilebilir.

# Kod'un Farklı Niteleyicileri



- “Kod” kelimesinin başına farklı sıfatlar getirilerek nitelemeler yapılabilir:
  - High-performance code,
  - Secure-code, vs.
- Eğer bir kod parçası rahat anlaşılır, hızlıca değiştirilebilir, varsa hatası kolayca bulunup düzeltilebilir değilse ne kadar performanslı ya da güvenli olduğunun pek önemi olmayabilir.
- **Kaldı ki, kodun temiz olması ile yüksek performanslı ya da güvenli olması örneğin, çelişen şeyler de değildir.**

# The ONLY VALID MEASUREMENT OF CODE QUALITY: WTFs/MINUTE

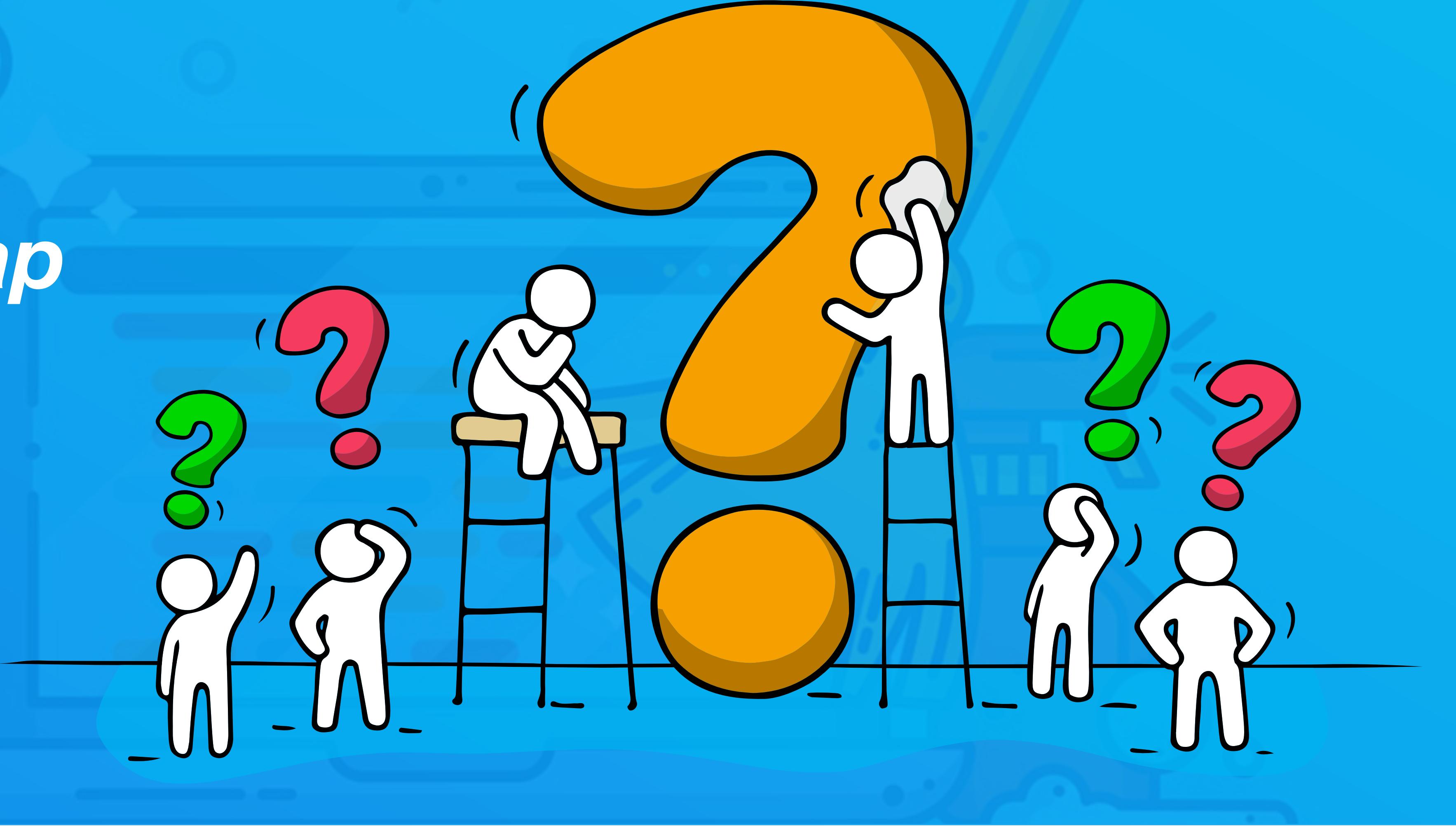


# Temiz Kod Nasıl Elde Edilir?



- Öncelikle kurumsal ve bireysel farkındalık ve bilgi,
  - Kurumsal süreç ve bireysel iş yapış şekilleri,
- Azıcık düşünme yani tasarım,
- Bol iletişim, paylaşma ve tartışma,
- Refactoring için cesaret ve zaman,
- PMD, SonarCube gibi kod kalitesi araçları,
- Kod review, formal ya da peer review süreci.

# *Soru ve Cevap Zamani!*



# Temiz Kod Çerçeveşi

# Temiz Kod Çerçeve - I



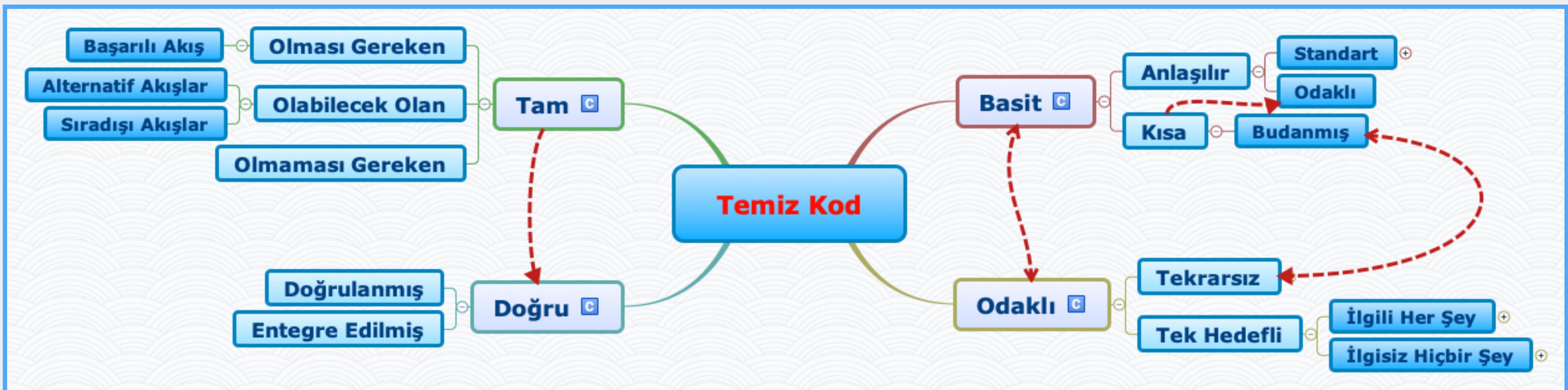
- Temiz kodu bileşenlerine ayırarak daha rahat anlaşılmasını ve uygulanmasını sağlamak amacıyla geliştirilmiş bir çerçevedir (framework).
- Çerçeve de ana ve alt hedefler ile onları gerçekleştirmede kullanılacak metodlar ve teknikler verilmiştir.

# Temiz Kod Çerçevesi - II

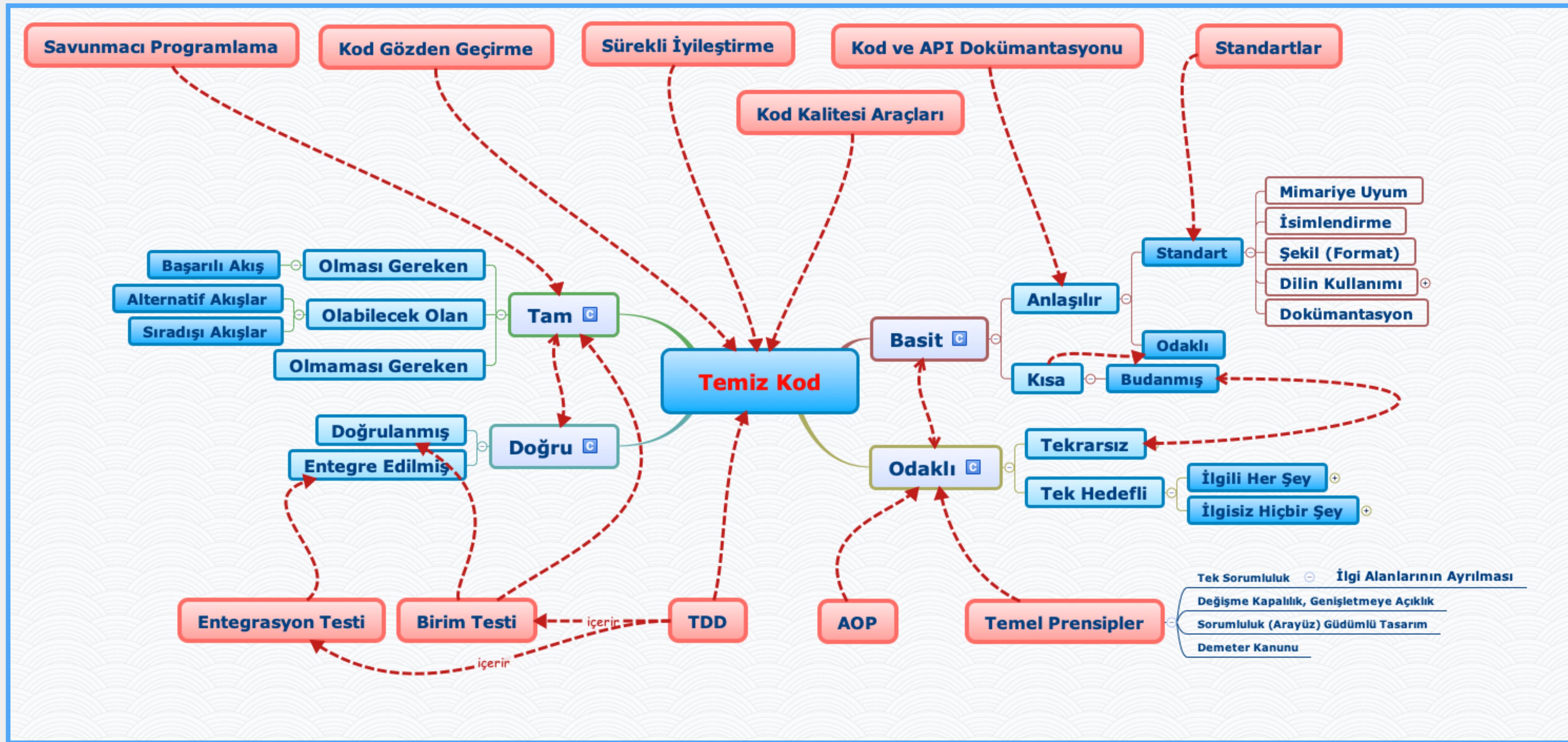


- Çerçeveye göre Temiz Kod, temelde dört özelliğe sahiptir:
  - **Basit**
  - **Odaklı**
  - **Doğru**
  - **Tam**

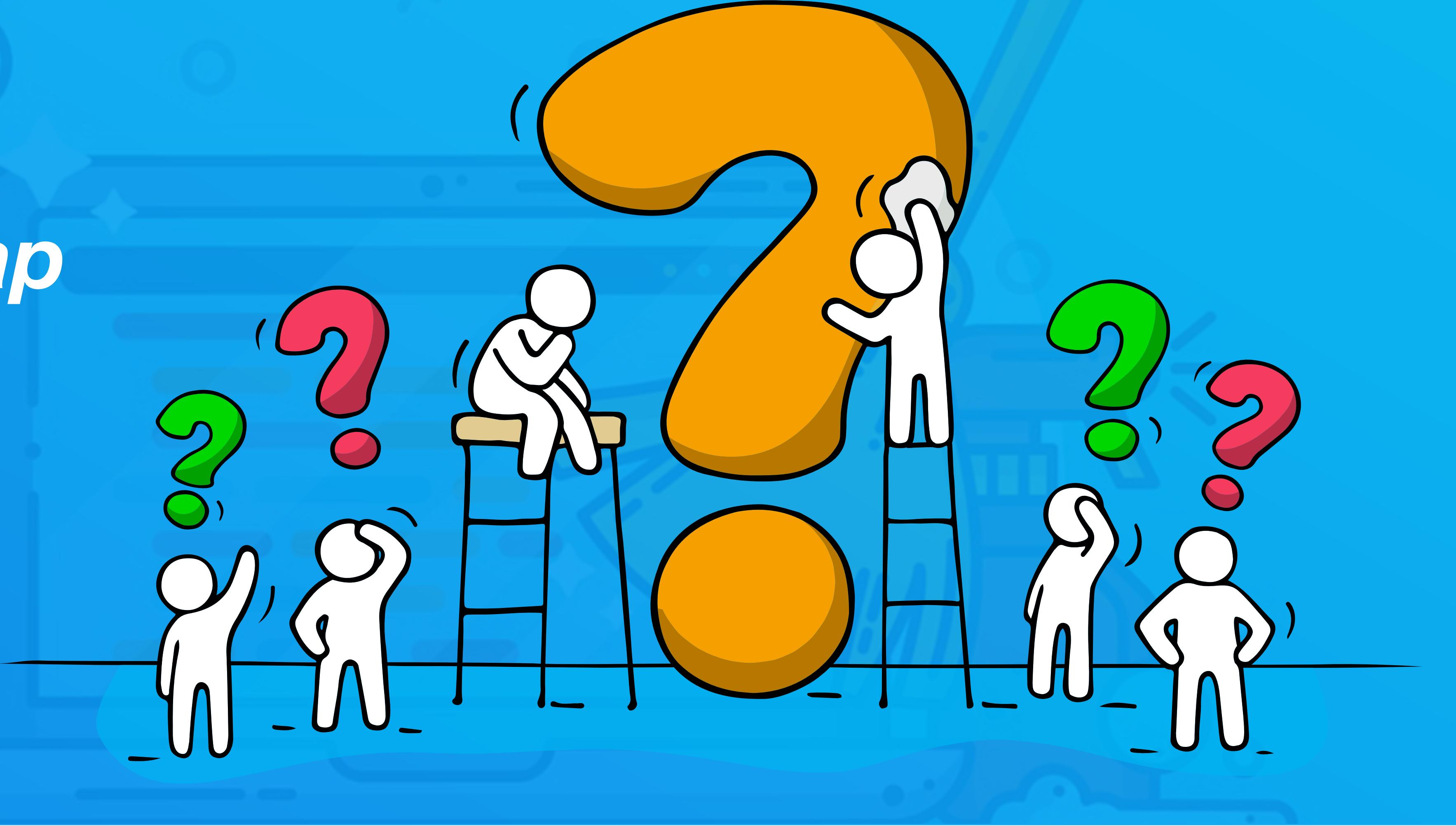
# Temiz Kod Çerçevesi - III



# Temiz Kod Çerçevesi - IV



# *Soru ve Cevap Zamani!*



# Basitlik



**selsoft**  
build better, deliver faster



- Çoğu zaman dilde basiti (simple), bayağı, gelişigüzel anlamında kullanırız.
  - Bu “galat-ı meşhur”dur.
- Basitlik (simplicity) meziyettir, karmaşıklık (complexity) gelişigüzelliktir, sıradanlıktır.
- Anlaşılabilirlik, takip edilebilirlik, makuliyet (rationality), bir düzeni (order) işaret eder.
- Anlaşılma zorluğu, takip edilemezlik, akıl-dışılık (irrationality) ise düzensizliğin göstergesidir.



- Basitlik, erişilmesi, sanılanın aksine, karmaşıklıktan daha zor olandır.
- Bir şeyi ilk defa yaparken muhtemelen karmaşık yaparız.
- Basit olan rahat anlaşılır, tutarlı, tek düzeye umulandır; şaşırtmayandır.
- Karmaşık olan ise anlaşılması zor, tutarsız ve çeşitlidir.
- **KISS:** Keep it simple, stupid - keep it simple and short!

# **Basitlik Nihai Karmaşıklıktır**



- Leonardo Da Vinci'ye göre

**Simplicity is the ultimate sophistication.**

**Basitlik nihai karmaşıklıktır.**

# Basitlik Nasıl Elde Edilir? - I



- Basit, bir seferde elde edilen bir durum değildir.
- Basitleştirmek, bir şeye tümüyle, her yönüyle hakim olup, asıl ile teferruatı ayırmaktır, gelişigüzel çıkarma, indirgeme değildir.
- Bu da çoğu zaman bir süreç gerektirir.
- Kodun basitleştirilmesine **iyileştirme (refactoring)** denir.
- İyileştirme temelde, çıkarmak ve bölüp-parçalamaktır.

# Basitlik Nasıl Elde Edilir? - II



# Basitlik Nasıl Elde Edilir? - III



- Mükemmellik nasıl elde edilir? Basitlikle mi yoksa karmaşıklıkla mı
- Fransız yazar **Antoine de Saint-Exupery**'a göre:

**Perfection is achieved, not when there is nothing more to add, but  
when there is nothing left to take away.**

**Mükemmellik, eklenerek bir şey olmadığında değil, çıkarılacak bir şey  
olmadığında başarılır.**

# Basitlik Nasıl Elde Edilir? - IV



- Einstein'e atfedilen

**Everything should be made as simple as possible, but no simpler.**

**Her şey olabildiğince basit olmalı, ama daha da basit değil.**

ya da

**Make things as simple as possible, but not simpler.**

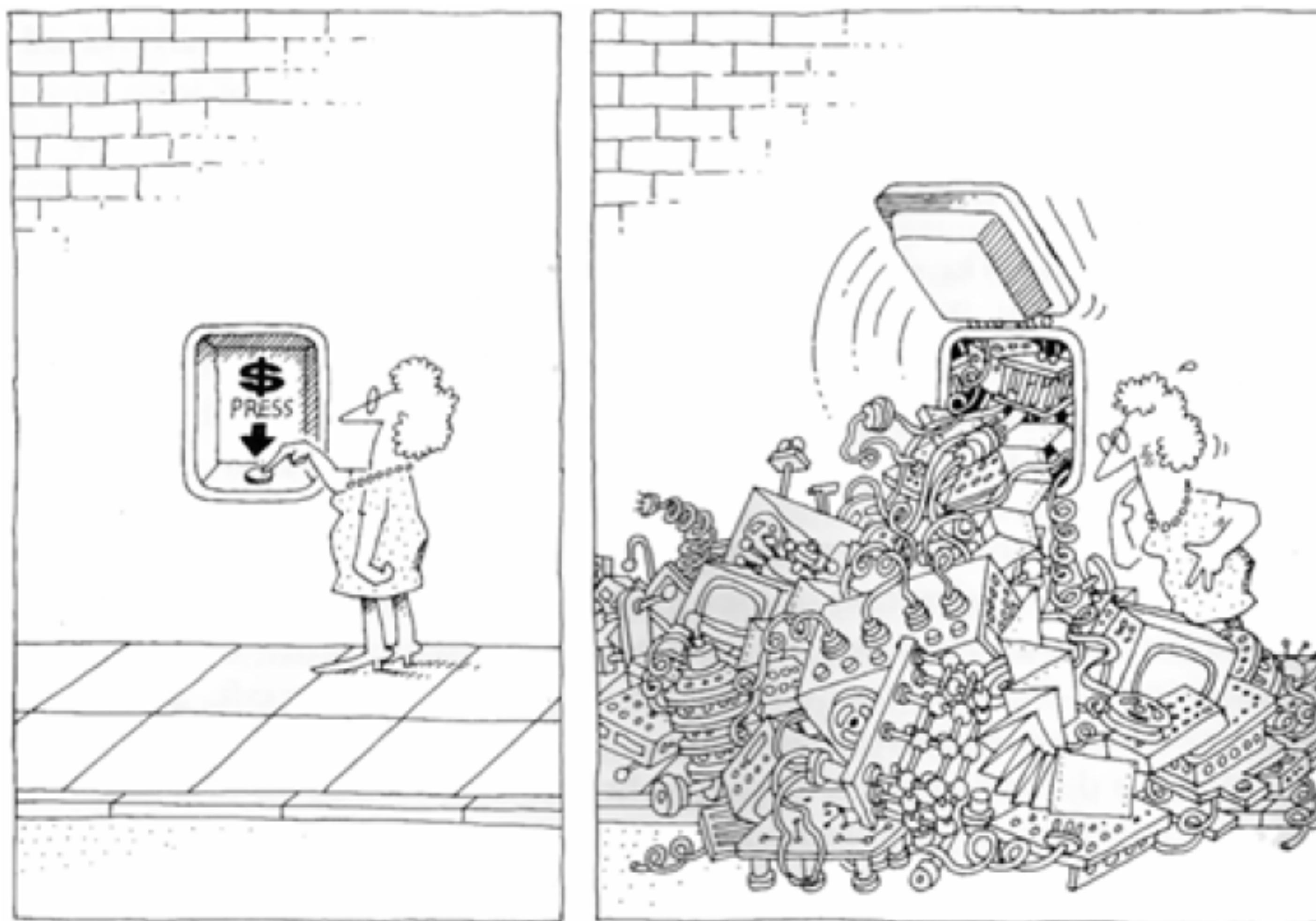
**Her şeyi olabildiğince basitleştir, ama daha da basit yapma.**

cümleleri bunu ifade eder.

# Basitlik İçin Tasarım



- Fiziksel mühendislik disiplinleri, ürünlerinde çok başarılıdırlar.
  - “3 yaşındaki torunum akıllı telefonu bizden iyi kullanıyor!”
- Bu başarıda en temel faktör, basitliğe yönelik tasarımındır.
  - Intuitive ya da sezgisel, beklentiği gibi.
- Ama bu başarı bir seferde olmadı, on yıllar hatta yüz yıllar süren bir süreçte meydana geldi.



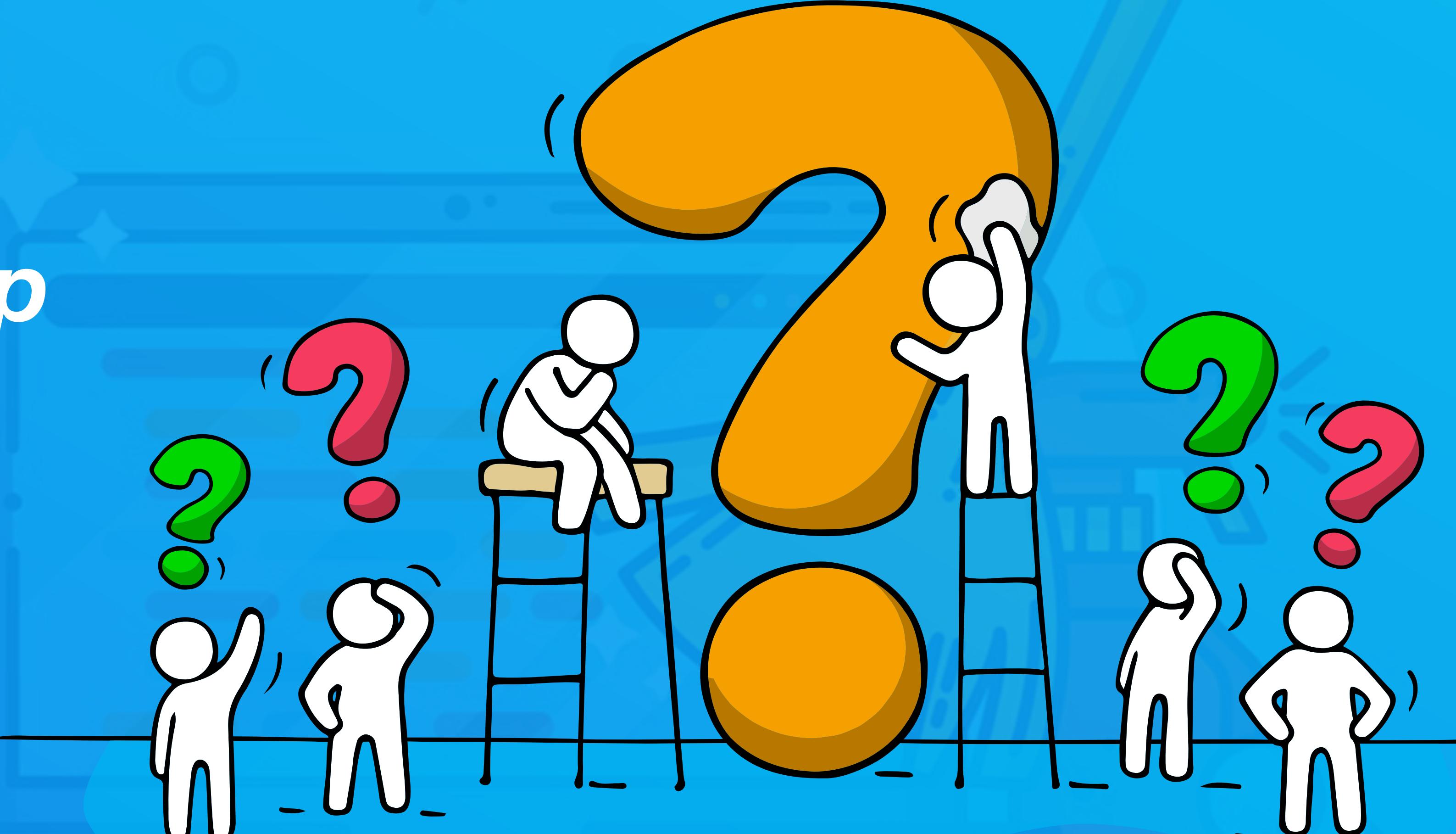
The task of the software development team  
is to engineer the illusion of simplicity.

# Basitlik İçin İletişim



- Öte yandan basitlik için güçlü iletişim şarttır.
- Üyeleri arasında iyi bir iletişimimin ve uyumun bulunmadığı ortamlarda, bulunana göre,
  - Basit olanı elde etmek çok daha zordur,
  - Herkesin üzerinde hemfikir olduğu yapılar oluşturulamaz,
  - Yeni üyelerin sisteme uyum sağlamaşı daha uzun zaman alır.

# *Soru ve Cevap Zamani!*





**s** selsoft

build better, deliver faster

**Basit Kod**



- C. A. R. Hoare der ki

**There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies.**

**Bir yazılım tasarıımı geliştirmenin iki yolu vardır. Birisi onu o kadar basit yapmaktır ki hiç bir hata olmadığı açık olsun. Diğerini o kadar karmaşık yapmaktır ki açıkta görünen hiç bir hata olmasın.**

# Basit Kod Nasıldır? I



- Basit kodu tanımlamak zordur ama en azından kötü örneklerle bakarak kalitenin ne olduğuyla alakalı bir şeyler söyleyebiliriz.
- En azından böyle değildir!

```
cardInfo.setIdNo(cardInfo.getTcCitizen() ?  
    (formUtils.isNullOrEmptyString(verifyOtpWSResponse.getTckn()) ?  
        registrationForm.getTckn() : verifyOtpWSResponse.getTckn()):  
    (formUtils.isNullOrEmptyString(verifyOtpWSResponse.getCustno())) ?  
        registrationForm.getTckn(): verifyOtpWSResponse.getCustno());
```

# Basit Kod Nasıldır? II



Bu çok daha basittir.

```
// Burada CardInfo'nun Idsi atanır. Id olarak eğer musteri TC vatandası ise  
// ve ya WS'den ya da fromdan gelen tckn atanır. Eğer müşteri TC vatandaşı  
// değilse Id olarak customerNo alınır.  
// Burada da yine ya WS'den ya da formdan gelen customer no atanır.  
String idNo = null;  
boolean isTcCitizen = cardInfo.getTcCitizen();  
String tcknFromForm = registrationForm.getTckn();  
String tcknFromWS = verifyOtpWSResponse.getTckn();  
boolean nullOrEmptyTcknFromWS = formUtils.isNullOrEmptyString(tcknFromWS);  
String custNoFromWS = verifyOtpWSResponse.getCustNo();  
boolean nullOrEmptyCustNoFromWS = formUtils.isNullOrEmptyString(custNoFromWS);  
  
if(isTcCitizen){  
    if(nullOrEmptyTcknFromWS)  
        idNo = tcknFromForm;  
    else  
        idNo = tcknFromWS;  
}  
else{  
    if(nullOrEmptyCustNoFromWS)  
        idNo = tcknFromForm;  
    else  
        idNo = custNoFromWS  
}  
cardInfo.setId(idNo);
```



```
cardInfo.setIdNo(cardInfo.getTcCitizen() ?  
    (formUtils.isNullOrEmptyString(verifyOtpWSResponse.getTckn()) ?  
        registrationForm.getTckn() : verifyOtpWSResponse.getTckn()):  
    (formUtils.isNullOrEmptyString(verifyOtpWSResponse.getCustno())) ?  
        registrationForm.getTckn(): verifyOtpWSResponse.getCustno());
```

```
// Burada CardInfo'nun Idsi atanır. Id olarak eğer musteri TC vatandası ise  
// ve ya WS'den ya da fromdan gelen tckn atanır. Eğer müşteri TC vatandaşı  
// değilse Id olarak customerNo alınır.  
// Burada da yine ya WS'den ya da formdan gelen customer no atanır.  
String idNo = null;  
boolean isTcCitizen = cardInfo.getTcCitizen();  
String tcknFromForm = registrationForm.getTckn();  
String tcknFromWS = verifyOtpWSResponse.getTckn();  
boolean nullOrEmptyTcknFromWS = formUtils.isNullOrEmptyString(tcknFromWS);  
String custNoFromWS = verifyOtpWSResponse.getCustNo();  
boolean nullOrEmptyCustNoFromWS = formUtils.isNullOrEmptyString(custNoFromWS);  
  
if(isTcCitizen){  
    if(nullOrEmptyTcknFromWS)  
        idNo = tcknFromForm;  
    else  
        idNo = tcknFromWS;  
}  
else{  
    if(nullOrEmptyCustNoFromWS)  
        idNo = tcknFromForm;  
    else  
        idNo = custNoFromWS  
}  
cardInfo.setId(idNo);
```

# Farklar - I



- Bu iki kod örneği arasında en temel fark anlaşılırlıktır.
- Birden fazla iş, gereksiz bir kısaltmayla tek bir işe indirgenmiş.
  - Karmaşıklığı arttırdığı gibi tekrar kullanımı da önlüyor.
- Mekanı rahat kullanmıyor, görüntü açısından sıkıntılı.
- Açıklanmış, dokümant edilmiş değil.
- Çok sık görülmeyen, aşina olunmayan bir “? :” kullanımı var.

# Farklar - II



- Tüm bunlara yol açan esas problem, soyutlama seviyesindeki hatadır.
- Bir metotta yapılabilecek bir işi bir cümlede yapıyor.
- Neden böyle bir iş bir metotta yapılacak olsun ki?
- Burada açıklanması gereken şey, “soyutlama seviyesine uygun tek iş” ile neyin kastedildiğidir.
- Basit kod en temelde doğru soyutlama seviyesinde olan koddur.

# Basit Kod - I



- Basit olan kodun temelde iki özelliği vardır:
  - Basit kod rahat anlaşılır,
  - Basit kod olabildiğince kısadır.
- Rahatça anlaşılabilir kod hem standartlara uygun olarak yazılmıştır hem de odaklıdır, yani bir yerde sadece bir şeyi yapmayı hedefler.

# Basit Kod - II



- Basit kod kısadır ve küçüktür, çünkü hem odaklıdır hem de budanmıştır.
- Hiç bir kod kendi başına kısa olsun diye yazılmaz, kısalık ancak odaklı olarak ile başarılabilir.
- Eğer kod öncelikle sadece gerekliliğini yapar, gereksiz olana dokunmaz ve sonrasında da iyileştirilirse, bir süreç içerisinde, zamanla olabilecek en kısa hale gelir.

# Basit Kod Anlaşılırdır



- Max Kanat-Alexander Code Simplicity kitabına şöyle giriş yapar:

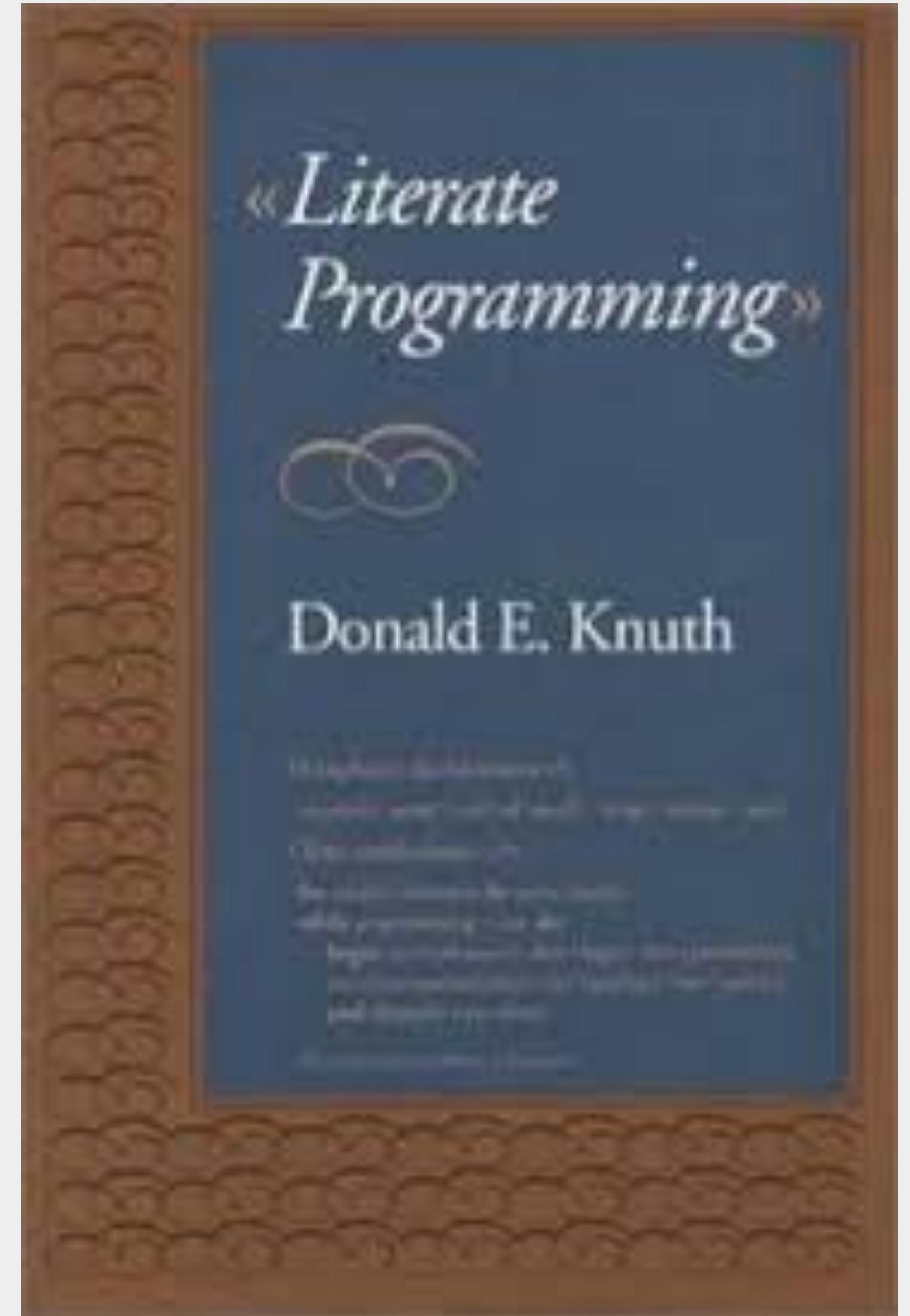
**The difference between a bad programmer and a good programmer is understanding . That is, bad programmers don't understand what they are doing, and good programmers do. Believe it or not, it really is that simple.**

**İyi programciyla kötü programci arasındaki fark anlamadır. Yani kötü programcılar ne yaptıklarını anlamazlar, iyi programcılar ise anlarlar. İster inanın, ister inanmayın ama bu gerçekten bu kadar basittir.**

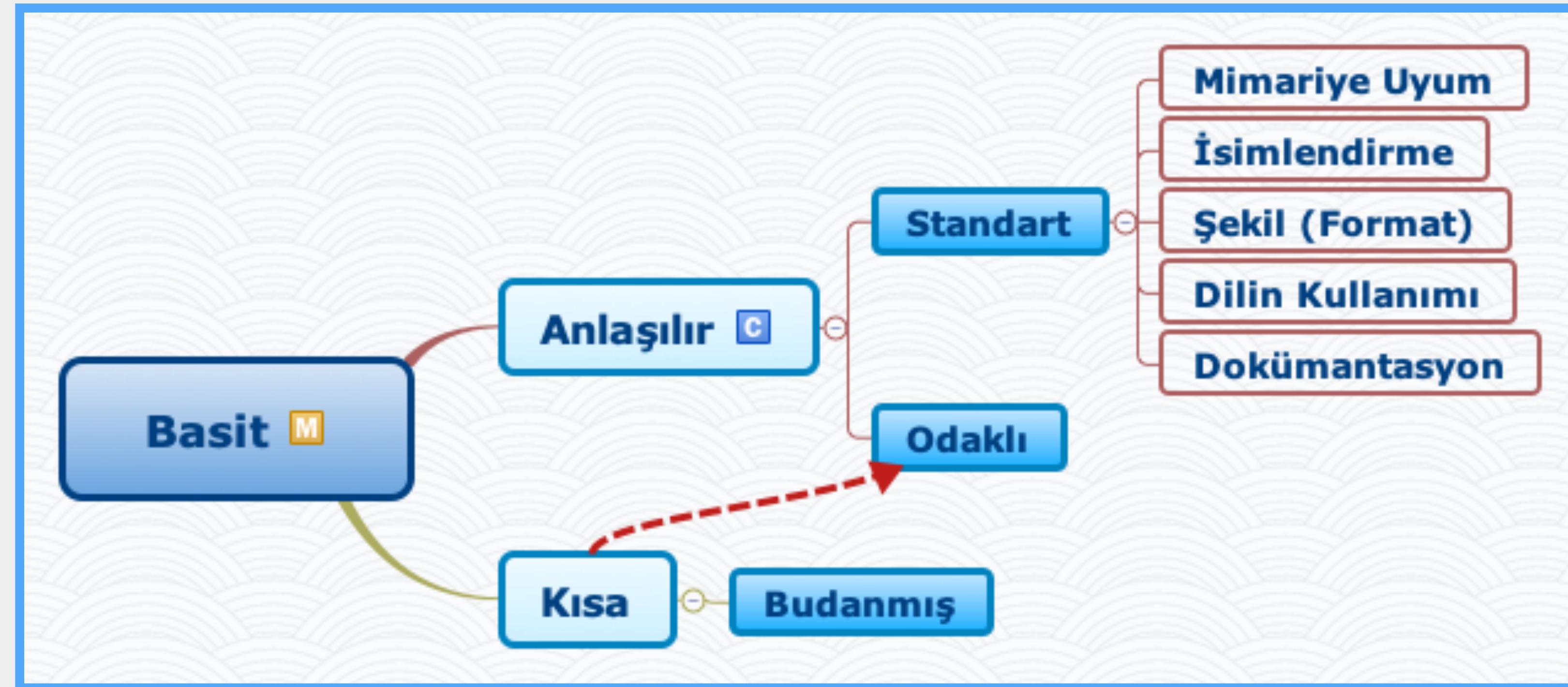
# Okunabilen Kod



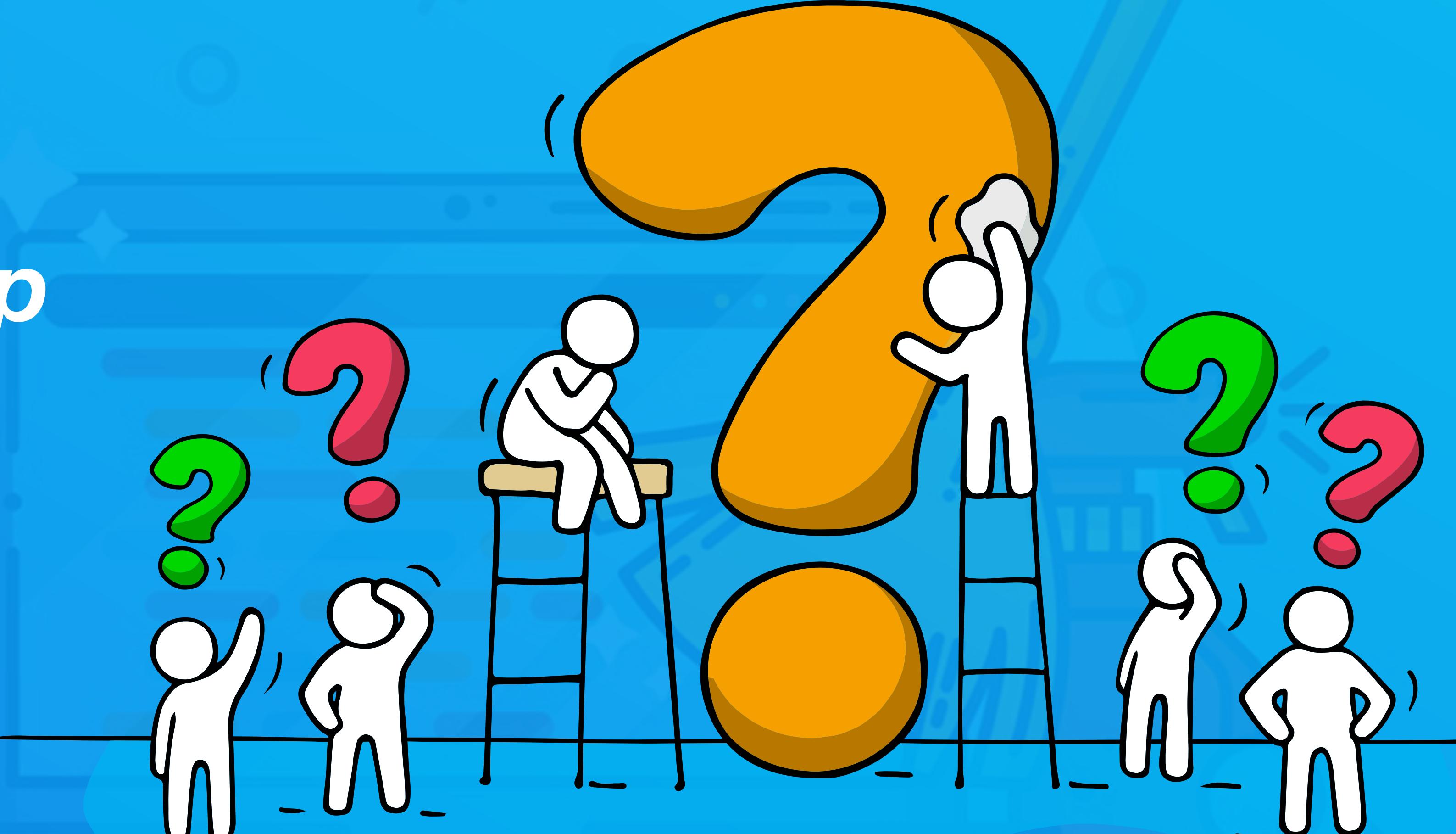
- Donald Knuth, Literate Programming isimli yaklaşımı önerirken, yazılan programların bir edebi eser olarak görülmesi gerektiğini ifade eder.
- Programlar edebi eserler gibidir, okunabilir (literate) olmalıdır,
- Okunamaz (illeterate) edebi eser yazmak nasıl tuhaf ise okunamaz program yazmak da anlamsızdır.



# Basit Kod - III



# *Soru ve Cevap Zamani!*





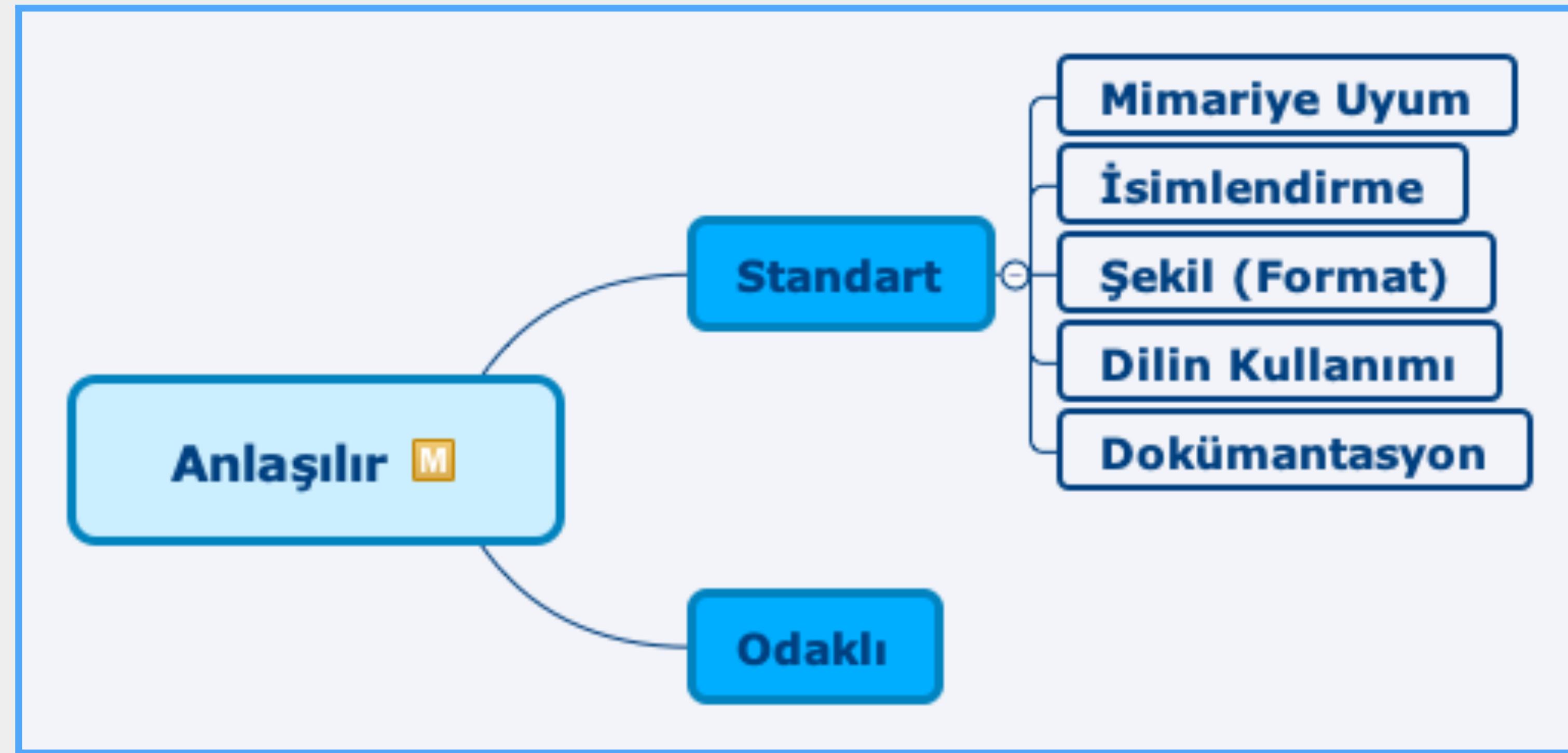
# Anlaşıılır Kod

# Anlaşıllır Kod - I



- Anlaşıllır kodun iki şartı vardır:
  - Şekil şartı, standart olması,
  - İçerik şartı, odaklı olmasıdır.
- Anlaşıllır kod standard bir şekilde yazılmıştır, mümkün olan her yerde aynı yaklaşımı, aynı çözümü, şekli, aynı ismi vs. kullanır.
- Anlaşıllır kod odaklıdır, bir yerde sadece ve sadece bir şeyi yapar ama tam yapar.

# Anlaşıllır Kod - II



# Standart Kod - I



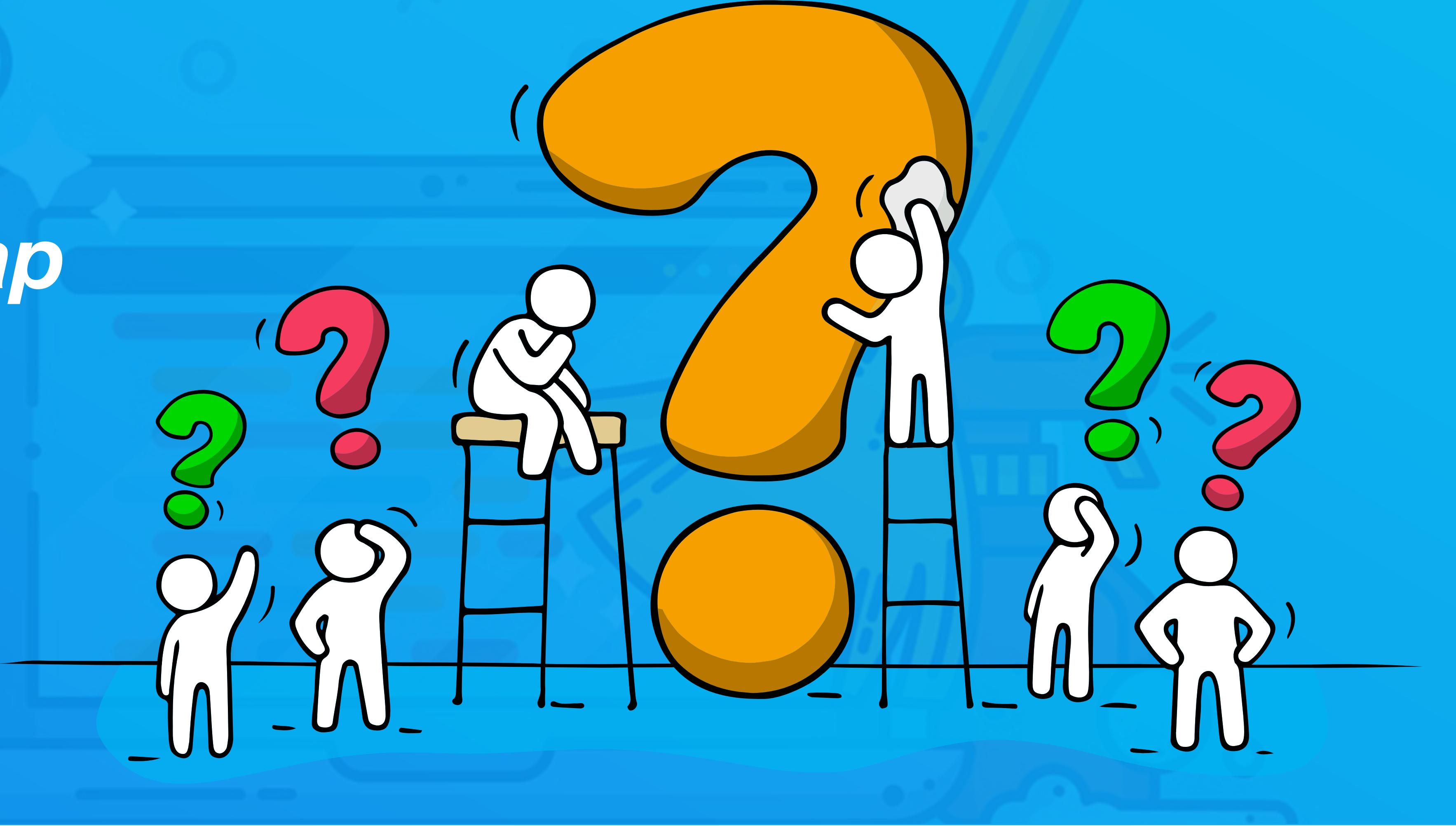
- Anlaşılır kod öncelikle standarttır, yani:
  - **Mimariye Uyum:** Belirlenen mimariye uygun geliştirilir,
  - **İsim:** Anlaşılır isimler içerir,
  - **Şekil:** Ferahtır, mekanı etkin kullanır dolayısıyla rahat okunur,
  - **İsim, Şekil ve Dilin Kullanımı:** Tutarlıdır yani şaşırtmaz, beklenildiği gibidir, aşina olunandır
  - **Dokümantasyon:** Gerektiği kadar, anlaşılmaya yardım edecek şekilde dokümantedir.

# Standart Kod - II



- Herhangi bir sebepten standartlara uyulmadıysa, muhakkak bu durum sebebiyle birlikte belirtilir ve diğer takım üyeleriyle paylaşılır.
- Ve bu sapma birden fazla yerde olacak ise, standardın parçası haline getirilerek sapma olmaktan çıkarılır.

# *Soru ve Cevap Zamani!*





**s** selsoft

build better, deliver faster

**Mimariye  
Uyum**



- Mimari, üst düzey tasarımıdır.
- Mimari, bileşenleri, en temel özelliklerini ve aralarındaki ilişkileri betimler.
- Mimari, temelde riskler ve fonksiyonel olmayan ihtiyaçlar göz önüne alınarak tasarlanır.



- Mimari tasarım, projenin en soyut ama en katma değerli kalıbı/  
şablonu ve yol göstericisidir.
  - Anlaşılıklık, mimari tasarımla başlar.
  - Mimari tasarım, takım üyelerince benimsenir, nasıl uygulanacağı  
öğrenilir ve projede uygulanır.
  - Yeterli miktarda mimari tasarım, projenin başlarında belirlenir ve  
ihtiyaç oldukça yolda tamamlanır.

# Mimariye Uyum - I



- Anlaşılır kod, en temelde takımca benimsenmiş mimariye uyduğu için anlaşılırdır.
- Mimari uyum, yeknesak, tekdüze, şaşırtmayan bir kod yapısı ortaya koyar.
- Anlaşılır kod, mimaride belirlenmiş yapıların dışına çıkmaz.
- Gerekirse mimari güncellenir ama daima mimariye sadık kalınır.



- Robert C. Martin'e göre mimari tasarımın ana gayesi, yazılım sistemini geliştirmek ve bakımını yapmak için gerekli olan insan kaynağını en az seviyede tutmaktır.
- Tasarım kalitesi ise, önce sistemi geliştirmek ve sonra bakımını yapmak için gerekli olan iş ve enerji ile orantılıdır.
- İyi tasarımlar, işi ve enerjiyi en az seviyede tutarlar, kötü tasarımlar ise daha çok iş ve daha yüksek enerjiye ihtiyaç duyarlar.

# Mimariye Uyumsuzluk - I



- Projelerde standarttan ilk sapma genelde mimariye uyumda yaşanır.
  - Mimarının dolambaçlı (!) yolları yerine, kısa ve pratik yollar keşfedilir.
  - Bu durumda isimlendirme de farklılaşmaya başlar.
  - Ve sıra dışı çözümler üretilir, bu da temiz olmayan kod demektir.

# Mimariye Uyumsuzluk - II

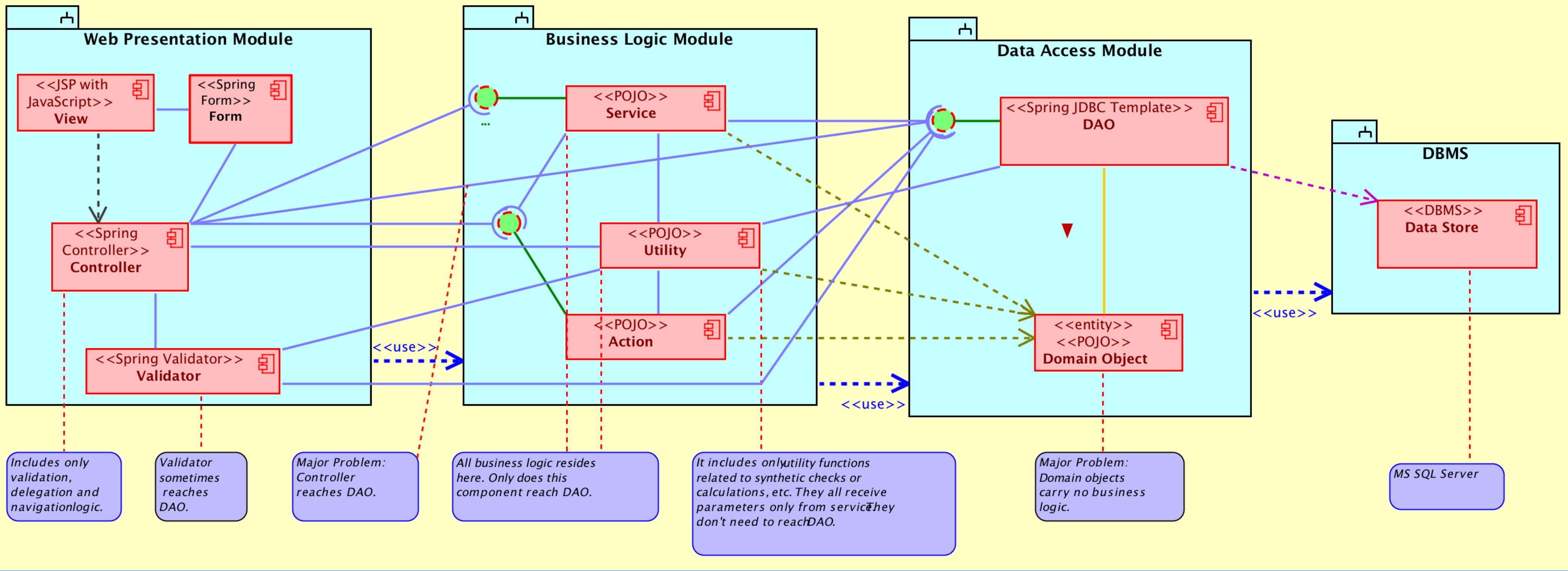


- Kötü tasarlanmış mimari ile çalışmada çözüm, kısa yollar bularak mimariden sapma değildir, mimariyi değiştirmektir.
- Çünkü aslolan, iyi ya da kötü, anlaşılmış olana uymaktır,
- Anlaşılma yönünden kötü bir çözüm, birden çok iyi çözümden daha iyidir.

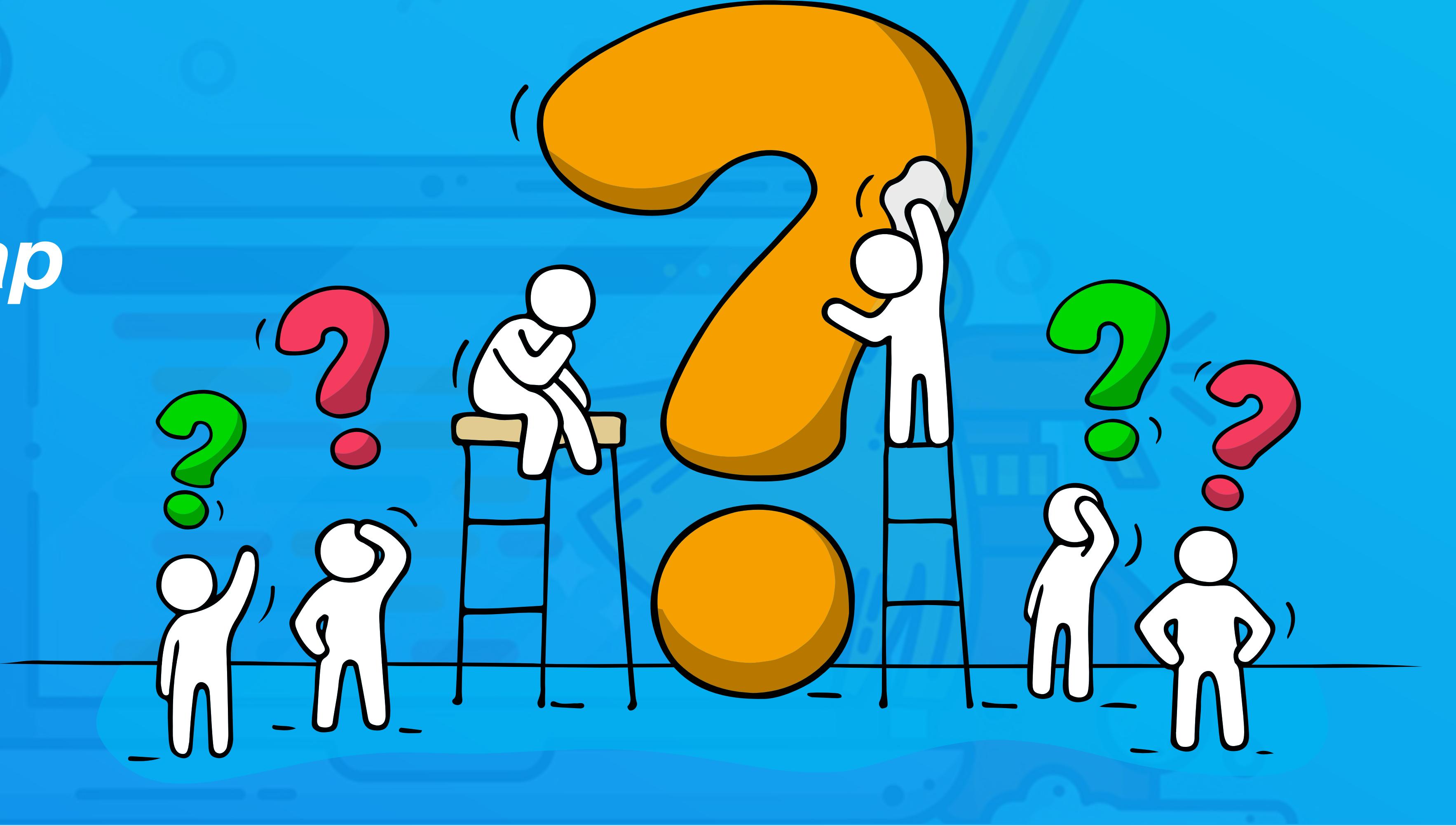
# Mimariye Uyumsuzluk - III



Component Diagram – As is



# *Soru ve Cevap Zamani!*





**s** selsoft

build better, deliver faster

**isimlendirme**

# Dilimiz ve Dünyamız



- L. Wittgenstein Tractatus Logico-Philosophicus (5.6 nolu not) isimli kitabında şöyle der:

**The limits of my language mean the limits of my world.**

**Dilimin sınırları, dünyamın sınırlarını demektir.**

**Dilimin sınırları, dünyamın sınırlarını ımler.**





**There are only two hard things in Computer Science: cache invalidation and naming things.**

**Bilgisayar Bilimlerinde sadece iki tane zor şey vardır: ön belleği temizleme ve şeyleri isimlendirme.**

**Phil Karlton**

# İsimlendirme ve Temiz Kod



- Temiz kodda, isimler de temizdir.
- Düzgün isimlere sahip olmayan kodun, temiz olması mümkün değildir.
- Düzgün isimlere sahip olmayan kodun, yüksek ihtimalle odaklı ve kısa olması da mümkün değildir.
- Çünkü isimler, düşünce dünyamızı ele verirler, ilkel ve karışık düşüncelerde, en temelde isimler belirsizdir.

# İngilizce İsimlendirme - I



- Öncelikle, İngilizce isimler kullanın.
- İngilizce isimlendirme ile, ileride yabancılarla çalışma, yazılımın dolayısıyla da kodun satılması, açık kaynak yapılması vb. durumlarda problem olmaz.
- İngilizce isimlendirme, söz dizimi ve yazım açısından da programlamaya daha uygundur.

# İngilizce İsimlendirme - II



- Ama İngilizce hecelemeye (spelling) dikkat edin:
  - Doğrusu hangisi “receive” mi “recieve” mi?
  - "**I before E, except after C**": *believe, die, friend* ama *receive!*

```
public Order receiveOrder(){ }  
public Order recieveOrder(){ }
```

# İsimlendirme - I



- Anlaşılır isimlendirmenin en temel engeli kısaltmadır.
- Her kısaltma bir kodlamadır ve kısaltanın bir zihinsel durumunu yansıtır.
- Zihinsel kodlamalardan kaçının, açıkça anlaşılır olun.

```
private int nofInst;
```

```
private int nofInst;
```

- “*I*” mı “*L*” mi “*1*” mi? “*L*” ise “*fI*” “*float*” mı demek?
- “*number of*” mu “*no*” mu?
- Ya da “*Institution*” mı, “*Installation*” mı yoksa “*Installments*” mı?

# Kabul Edilebilir Kısalmalar - I



- Çok bilindik, evrenselleşmiş değil ise kısaltmalardan kaçının:
  - Kapsamı son derece kısa, geçici (temporary) nitelikli değişkenler.
  - Örneğin **for** vb. döngülerde **i**, **j**, **k**, gibi değişkenlerde, genelde bitiş şartı değişkene anlam verir ve bu türden değişkenlerin hayatları çok kısadır.
  - Bağlamda anlamı hemen belirenler: **in**, **out**, **r** (**Circle**'da yarıçap, radius) vb. değişken isimleri

# Kabul Edilebilir Kısalmalar - II



- Evrensel kısaltmalar: `Http`, `Tcp`, `Pi`
- `init()` vb. fonksiyon/metot isimleri

# Kısa - Uzun İsim



- Kısa, bulmaca gibi isimler yerine uzun, okuması zaman alan ama anlamlı isimler verin.

```
int amnt  
int tmpAmnt;
```

```
Product pr
```

```
File file
```

```
int temporaryAmount
```

```
Product productToDelete
```

```
Product productToShipToSecondaryAddress
```

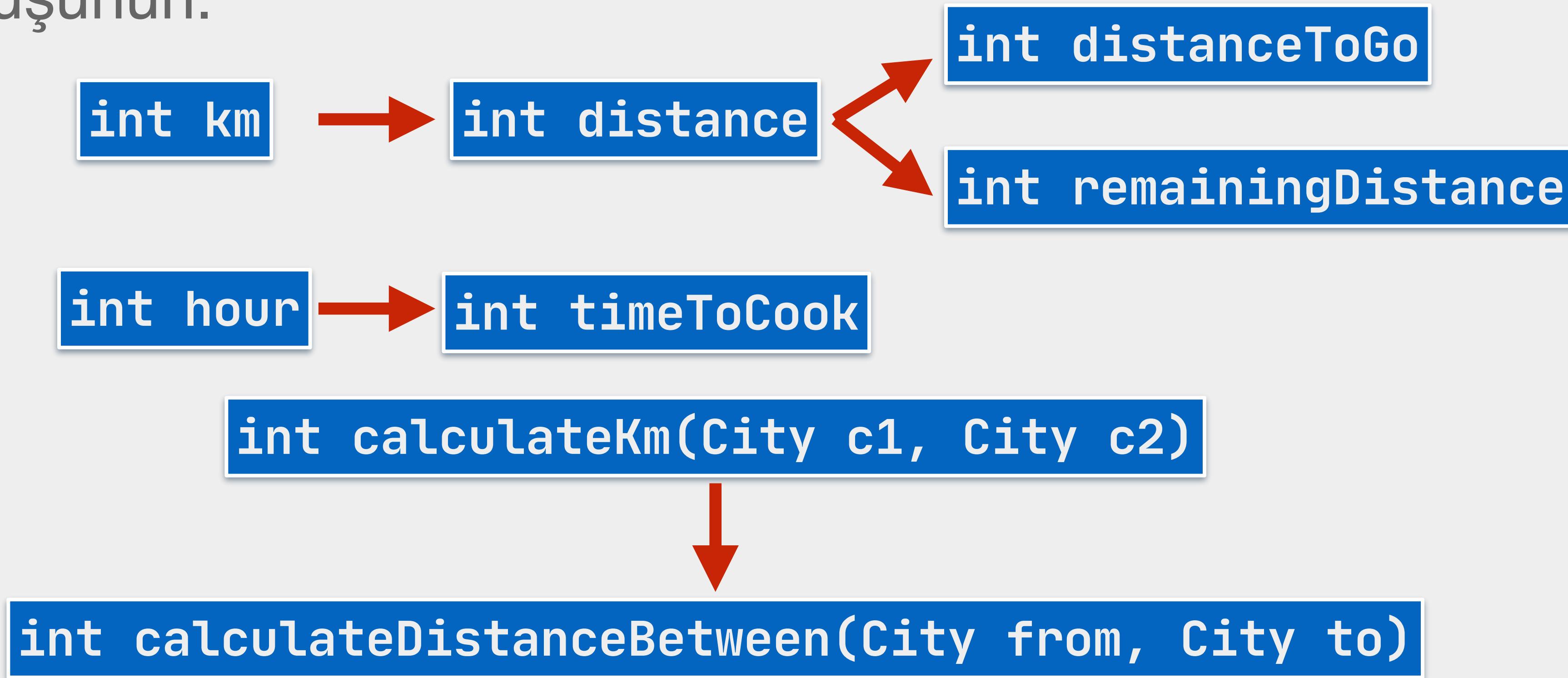
```
File fileToReadFromClientToConfigureServer
```

- İş mantığı karmaşıklaşıkça ayırt edici isimler daha önemli hale geliyor.

# Soyut, Kapsayıcı İsim



- İsimleriniz olabildiğince soyut ve kapsayıcı olsun.
- Sadece değişkenin aldığı değerlere bakarak isim vermeyin, neyi temsil ettiğini düşünün.



# isimlendirme - II



- Ne yapıyorsanız ona isim verin, isimlendirdiğinizi yapın.

```
List<RegisteredEmail> findPrimaryUserListByUid(String uid);
```

- Eğer bir yerde yaptığınız şey değişiyorsa, ismi de değiştirin.

```
List<User> findPrimaryUserListByUid(String uid);
```

# isimlendirme - III



- İsmen uzun olması anlamlı ya da bağlama uygun olduğunu göstermez.
- “**executeInner**“ ya da “**executeInner1**“ ne demek?

```
public boolean executeInner(...){...}
```

```
public boolean executeInner1(...){...}
```

- “**executeUser**“ ne demek?

```
public boolean executeUser(...){...}
```

# isimlendirme - IV



- Ne yaptığınızı bilmiyorsanız iyi isim veremezsiniz!

```
public class GenericUtil{...}
```

```
public class SystemUtil{...}
```

```
public class Util{...}
```

```
public class Helper{...}
```

- Genel isimler vermeyin, daima daha özel, daha açıklayıcı olun.

```
void process(){...}
```

```
void processDailyEarningReportRequest(){...}
```

```
class Util{...}
```

```
class DateUtil{...}
```

```
class XMLParsingUtil{...}
```

# İsimlendirme - V



- Bazen genel isimler çok belirgin bir bağlamda anlamlı olabilir:
  - Factory metodlarının `create()`
  - Command kalibindaki metodun `execute()`
- olması gibi
- Aynı kelimeleri kullanan değişik isimler oluştururken dikkatli olun.

StudentRegistration  
RegistrationStudent

# İsimlendirme - VI



- Tiplere, değişkenlere ve metotlara farklı kipte ve soyutlama seviyesinde isimler verin.
  - Tipler, kavramı temsil eden yalın kip,
  - Fonksiyon/metotlar, kavramın sorumluluğunu temsil eden emir kipi,
  - Değişkenler, kavramın bir parçasını, özelliğini temsil eden yalın kip,
- Tip isimleri metotları, metot isimleri içindeki değişkenleri vs. kapsamalı.
- Bu amaçla iş alanını iyi anlayın, teknik terimleri kavrayın.



**StudentRegistration** ⇒ Type name

**Student Registration** ⇒ Process managed in type **StudentRegistration**

**startStudentRegistration()** ⇒ Part of type **StudentRegistration**, method name

**endStudentRegistration()** ⇒ Part of type **StudentRegistration**, method name

**verifyStudentRegistration()** ⇒ Part of type **StudentRegistration**, method name

**registerStudent()** ⇒ Method name

**registerStudentForSummerTerm()** ⇒ Method name

**studentRegistered** ⇒ Variable name

**studentToBeRegistered** ⇒ Variable name

**studentWorkingForRegistrationDepartment** ⇒ Variable name

**studentRegistrationAttemp** ⇒ Variable name

# İsimlendirme - VIII



- Aynı kavramlara, işlere farklı yerlerde farklı isimler vermeyin,

```
fetchGraduateStudents()  
getStudents()  
retrieveAllStudents()  
loadAllStudents()
```

```
cost;  
total;  
grossTotal;  
price
```

- Aynı isimlerle farklı yerlerde farklı kavramlar, işler kastetmeyin.
- İsimlendirmede tüm sistemde, analist, PM, tester, developer, iş birimleri vs. tüm çalışanlarla tutarlı bir dil kurun.
- Domain-Driven Design

# isimlendirme - IX



- Ve lütfen İngilizce ve Türkçe'yi karıştırarak isimlendirmeyin!

```
public List getTotalPartageBySatan(){...}
```

ParaDate

```
public List getTotalPartageBySalesChannel(){...}
```

TransactionDate



## *Object reference not set to an instance of an object.*

**Description:** An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error and where it originated in the code.

**Exception Details:** System.NullReferenceException: Object reference not set to an instance of an object.

### Source Error:

```
Line 15:         var Query = MenulerHelper.GetAllMenuler() Where(f => f.MENUID == tool.GetQueryString("MENUID"));
Line 16:
Line 17:         rpData.DataSource = MenulerHelper.GetAllMenuler().Where(f => f.MENUID == Query.FirstOrDefault().USTID);
Line 18:         rpData.DataBind();
Line 19:
```

# isimlendirme - XI



- Metin (text, string) ya da sayısal vb. sabitelerde yanlış yazımların vs. önüne geçmek için enumeration vb. yapılar kullanın.
- Hem daha yüksek ifade gücü elde edilir hem de type-safety kazanılır.

```
String day = form.getDay();
```

```
switch(day){  
    case "Monday": ...  
    case "Tuesday": ...  
    ...  
}
```

```
Day day = form.getDay();
```

```
switch(day){  
    case Day.MONDAY: ...  
    case Day.TUESDAY: ...  
    ...  
}
```

# İsimlendirme - XII



- Ortaklaşa kararlaştırılan, herkesin benimsediği kendi isim standartlarınızı oluşturun ve herkesin buna uymasını ve aynı şekilde isim vermesini sağlayın.
- Frameworklerde çok güzel isimler vardır, örneğin **Spring Repository** sınıfındaki query metod isimleri.

```
findById(...)  
findByName(...)  
findByNameAndLastname(...)  
findByAddress(...)  
findByNameAndLastname(...)  
  
findByDateBetween(...)  
  
findBySizeSmall(...)  
  
findByOpenTrue(...)
```

# Anlaşırlık ve Kısalık



- Kısa kod iyidir ama kısa kod kısa isimlerlerle sağlanmaz,
- Anlaşırlır olma ile kısa olma “?:” örneğindeki gibi zaman zaman çelişebilir,
- Bu durumda aslolan anlaşırlıktır, kısa olmak adına anlaşırlır olmaktan fedakarlıkta bulunamayız.

```
double rs = a + ++b * c/a * b;  
//yerine  
double rs = a + (++b)* ((c / a)* b);  
  
//hatta  
b++;  
double d = c / a;  
double rs = a + b * d * b;
```

# How To Write Unmaintainable Code?



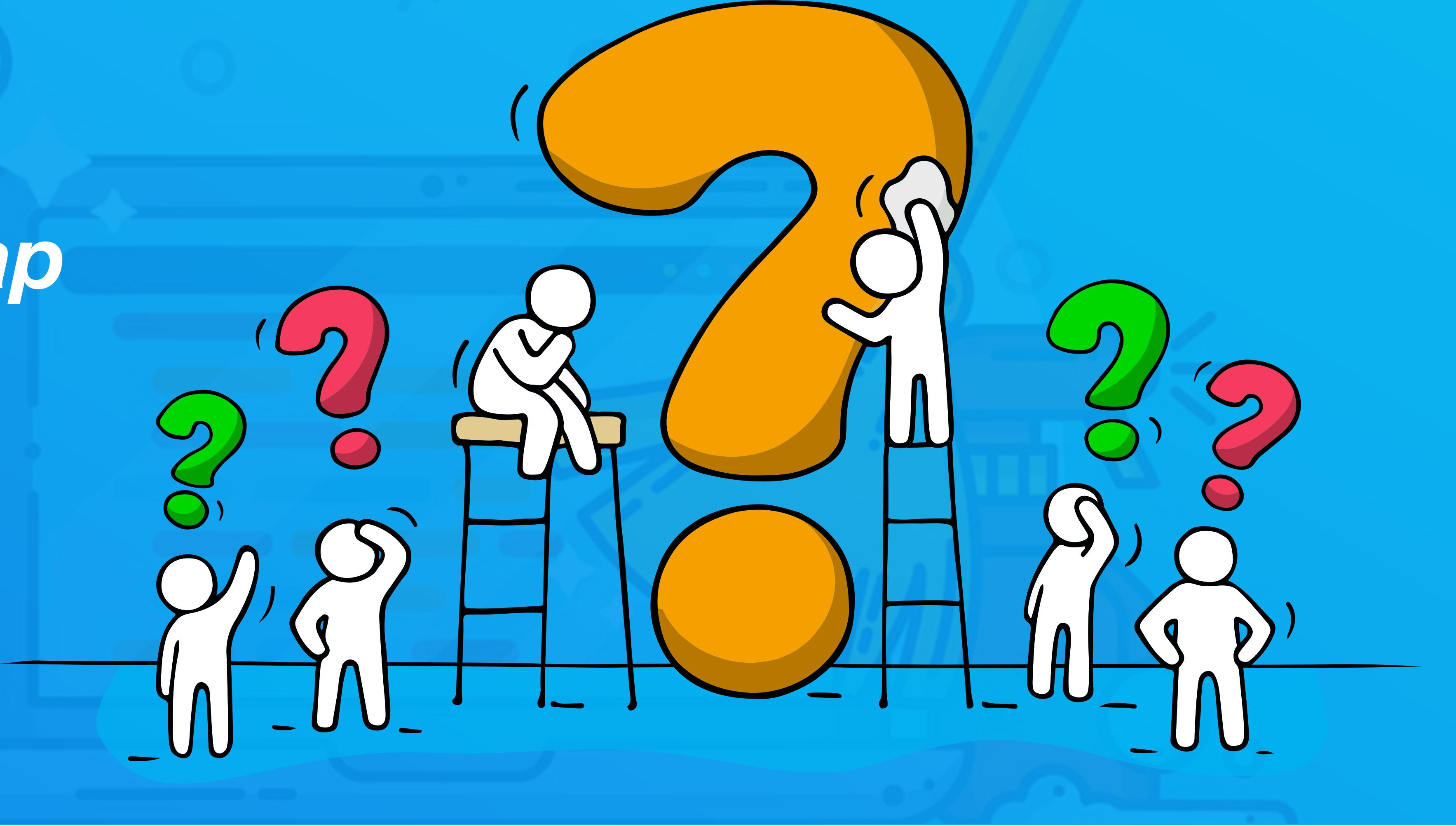
- “How To Write Unmaintainable Code?”
- [https://www.se.rit.edu/~tabeec/RIT\\_441/Resources\\_files/  
How%20To%20Write%20Unmaintainable%20Code.pdf](https://www.se.rit.edu/~tabeec/RIT_441/Resources_files/How%20To%20Write%20Unmaintainable%20Code.pdf)

# İsimlendirmede Temel Kurallar



- İngilizce isim.
- Kısaltma yok: Anlamlı olacak kadar uzun isimler!
- Şaşırıtmaca yok, standart var: Herkes aynı isimleri aynı anlamda ve aynı şekilde kullanıyor.

# *Soru ve Cevap Zamani!*





**s** selsoft

build better, deliver faster

# Standartlar

# Standartlara Uyum



- Muhakkak isimlendirme ve şekil (format) standartlarınız olsun:
  - Her dil için ayrı ayrı
  - Veri tabanı için
  - XML, properties dosyaları, web servisleri vs. için
- Ve bu standartlara uyum kabul edilebilir kodun olmazsa olmaz bir özelliği olsun.
- Kod kalite araçları, pair-programming ve gözden geçirmeler yardımıyla standartların uyulmasını kontrol edin.

# Code Conventions



- Dilde kabul görmüş kodı geleneğine (code convention) uyun:
  - **CamelCase** or **camelCase**
  - **snake\_case**
  - **kabab-case**
- Ayrıca özellikle değişken isimlerine ön ek getirerek (`iNumberOfPeople`) daha anlaşılır isimler amacıyla **Hungarian** gösterimi (notation) de vardır.

# İsimlendirme ve Şekil Standartları: Java



- Java kültürü isim ve şekil standartlarının uyuma çok önem verir:
  - Java Code Conventions September 12, 1997 (Oracle Java Code Conventions <http://www.oracle.com/technetwork/java/codeconv-138413.html>)
  - <http://www.ambysoft.com/downloads/javaCodingStandards.pdf>
  - Google Style of Java <http://google-styleguide.googlecode.com/svn/trunk/javaguide.html>



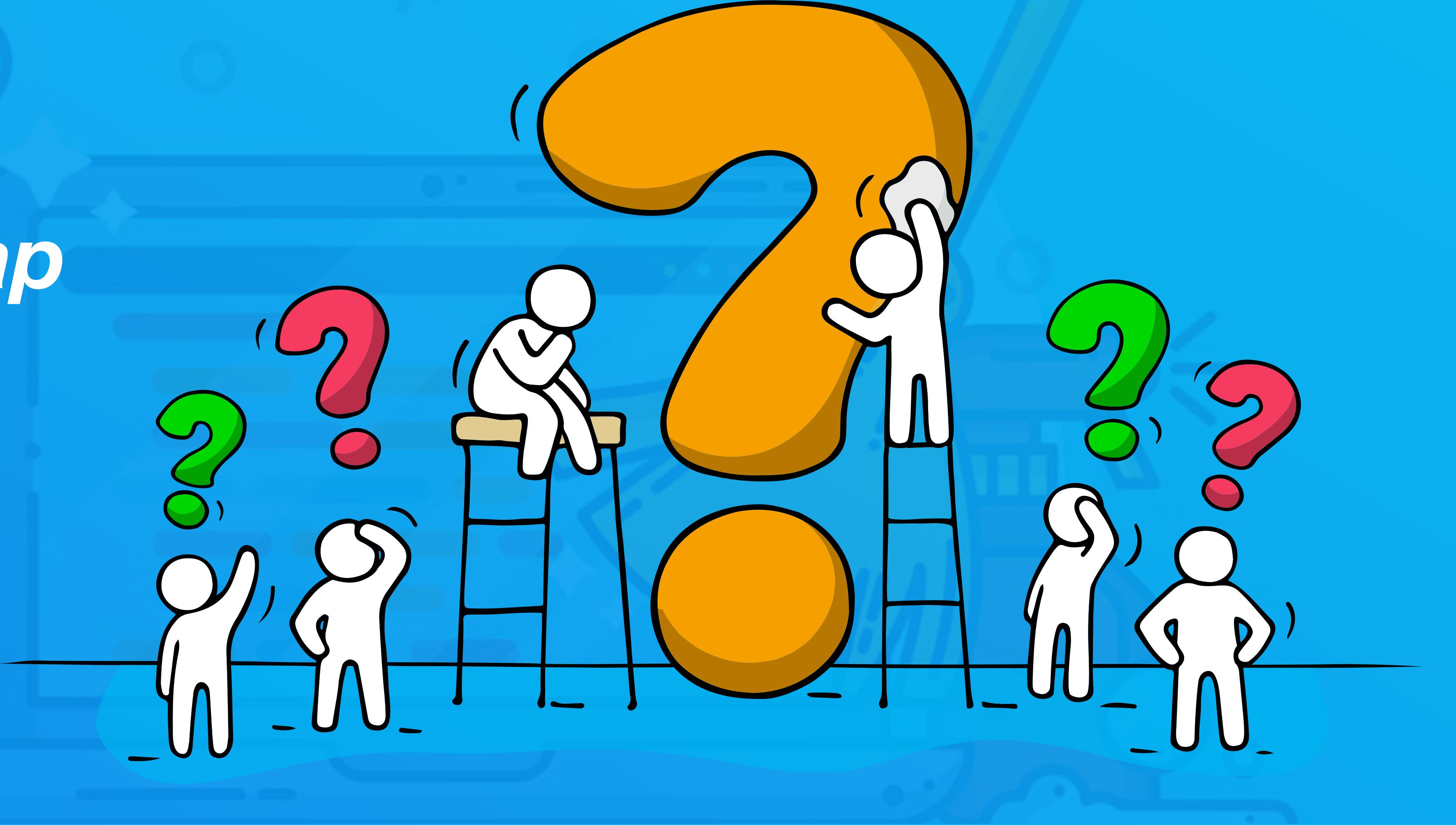
- JavaTurk'ün de Türkçe olarak oluşturulmuş bir isimlendirme ve şekil standardı vardır.
- <http://www.javaturk.org/javaturk-java-kodu-isimlendirme-ve-sekil-format-standardi/> adresinden HTML ve PDF yayınlanmıştır.
- Bu standart, Java dünyasındaki yaygın standartlara uygun olarak hazırlanmıştır.
- Serbestçe kullanabilirsiniz.

# İsimlendirme ve Şekil Standartları: C#



- C# için isimlendirme ve şekil standartları:
  - Microsoft Naming Guidelines
    - <https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/naming-guidelines>
    - <https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/general-naming-conventions>
    - [https://docs.microsoft.com/en-us/previous-versions/dotnet/netframework-1.1/xzf533w0\(v=vs.71\)](https://docs.microsoft.com/en-us/previous-versions/dotnet/netframework-1.1/xzf533w0(v=vs.71))
  - C# Coding Standards and Naming Conventions <https://github.com/ktaranov/naming-convention/blob/master/C%23%20Coding%20Standards%20and%20Naming%20Conventions.md>

# *Soru ve Cevap Zamani!*





**S** selsoft  
build better, deliver faster

**Dilin  
Kullanımı**

# Standart Dil Kullanımı

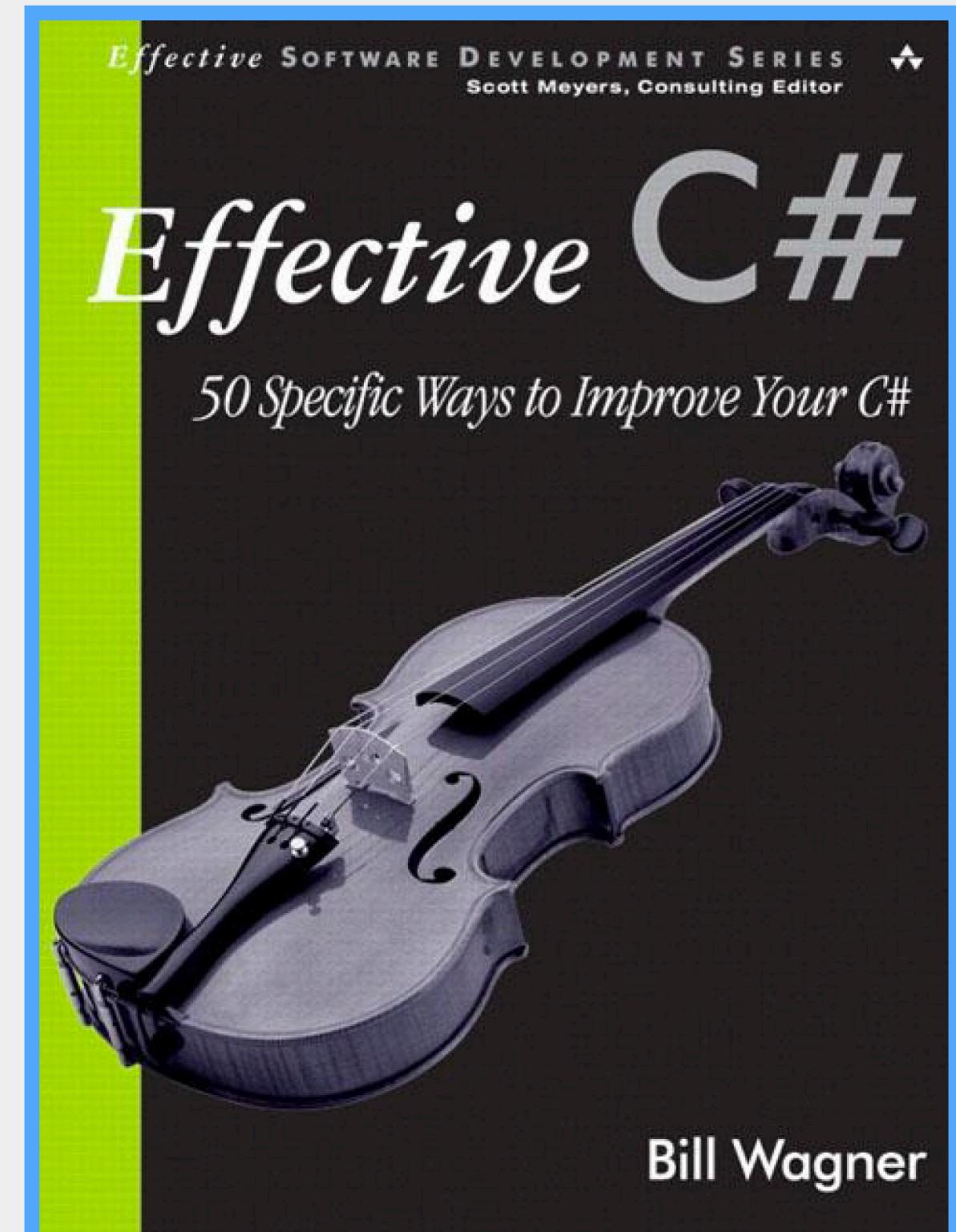
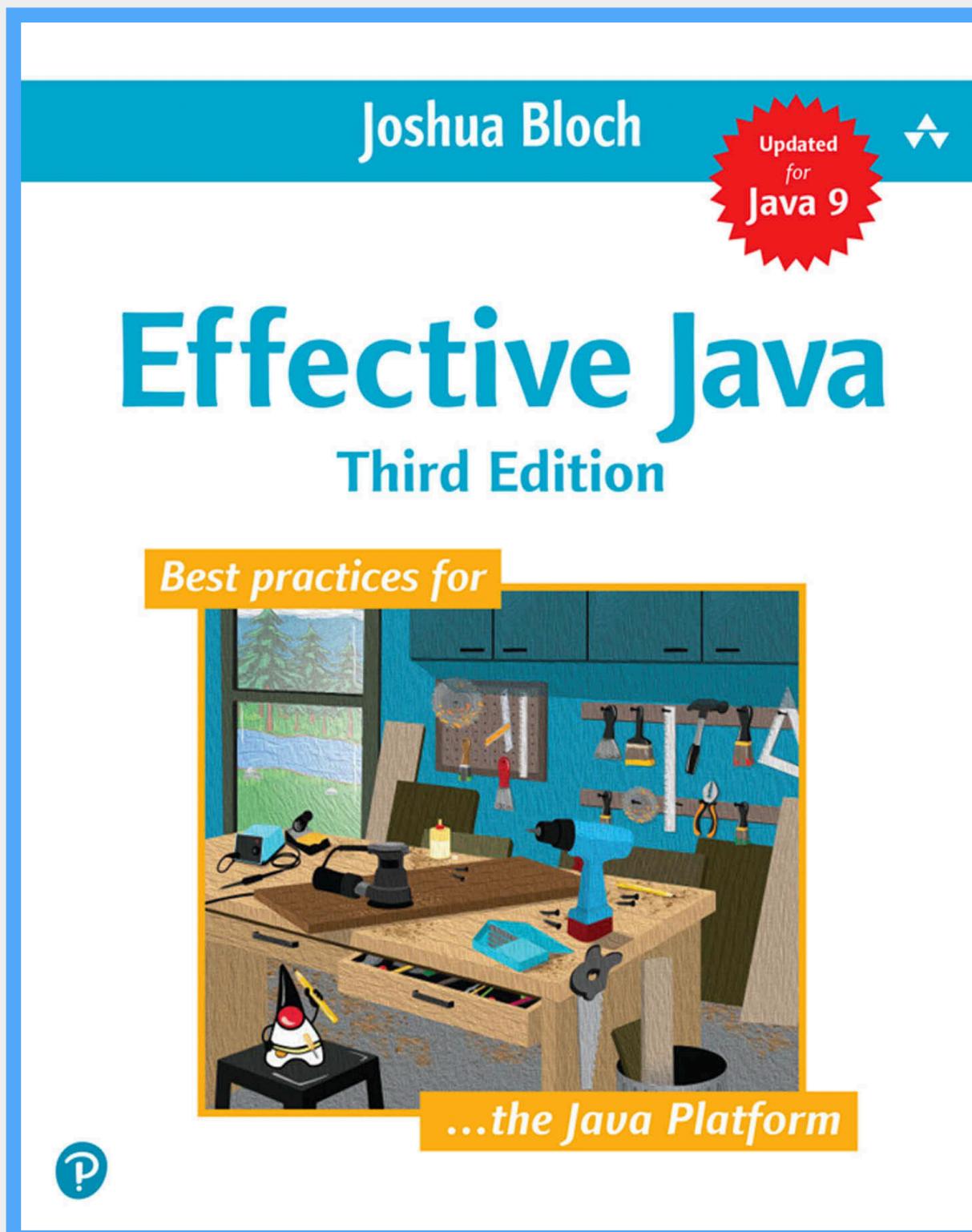
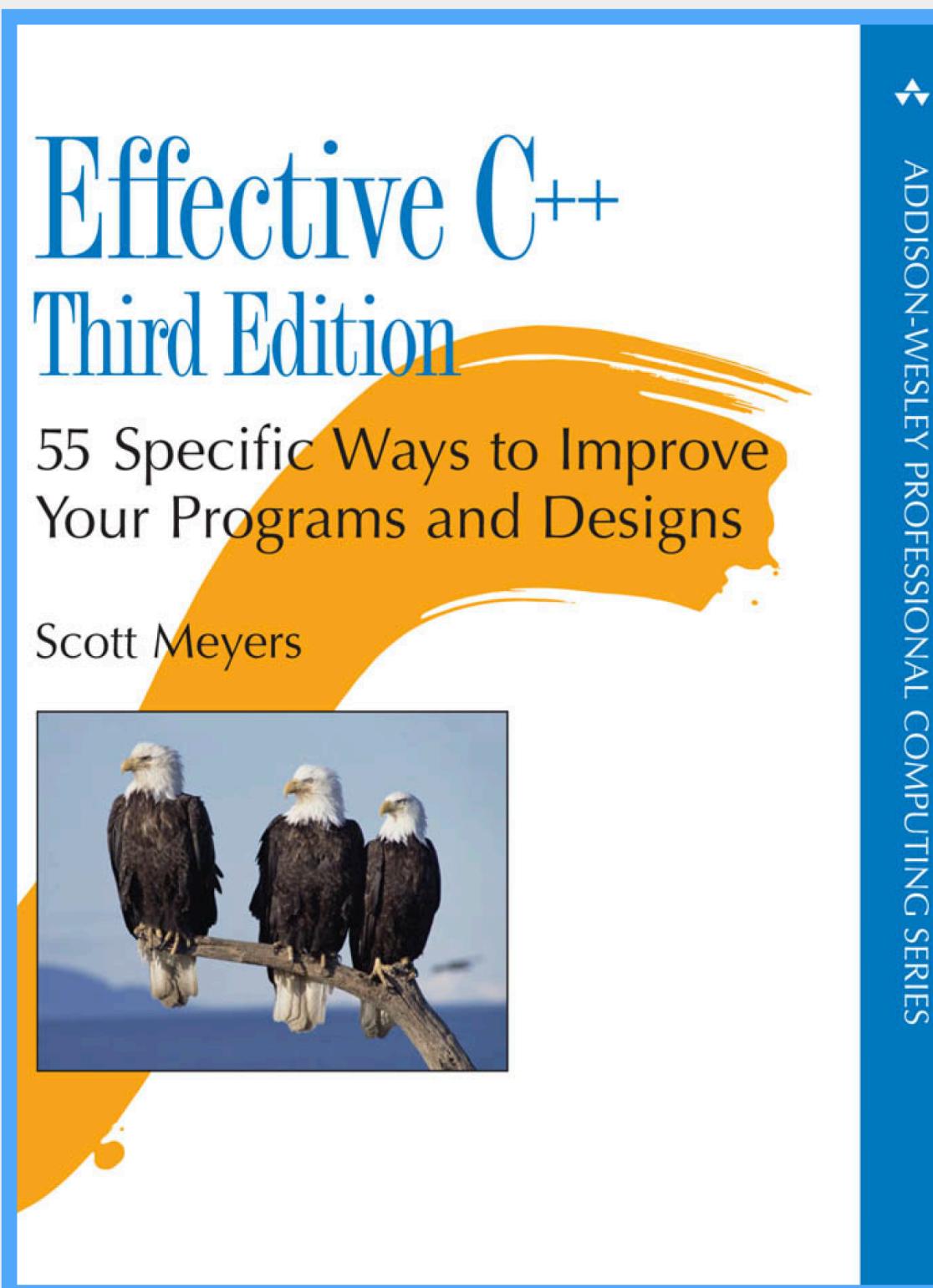


- Kullanılan dilin olabildiğince herkesçe kabul görmüş şekilde, etkin (effective) ve standart yapılarla kullanılması, anlaşırılığı arttırmır.
- Aynı problemi farklı şekilde kodlayarak çözen kod parçaları, anlaşılma problemine sebep olurlar.



- Her dilin, “idiom” denen kendine has deyimleri ya da kalıpları vardır.
- Dillerde, uzun süren gelişim sürecinde aynı problemi çözmek için çok farklı deyimler geliştirilir.
- Bu deyimleri seçmek ve onlara uymak, standart kod yazmak açısından önemlidir.

# Etkin Dil Kullanımı - I





## Item 7: Prefer Immutable Atomic Value Types

Immutable types are simple: After they are created, they are constant. If you validate the parameters used to construct the object, you know that it is in a valid state from that point forward. You cannot change the object's internal state to make it invalid. You save yourself a lot of otherwise necessary error checking by disallowing any state changes after an object has been constructed. Immutable types are inherently thread safe: Multiple readers can access the same contents. If the internal state cannot change, there is no chance for different threads to see inconsistent views of the data. Immutable types can be exported from your objects safely. The caller cannot modify the internal state of your objects. Immutable types work better in hash-based collections. The value returned by `Object.GetHashCode()` must be an instance invariant (see [Item 10](#)); that's always true for immutable types.

You can also create factory methods to initialize the structure. Factories make it easier to create common values. The .NET Framework `Color` type follows this strategy to initialize system colors. The static methods `Color.FromKnownColor()` and `Color.FromName()` return a copy of a color value that represents the current value for a given system color.

## Item 1: Consider static factory methods instead of constructors

The traditional way for a class to allow a client to obtain an instance is to provide a public constructor. There is another technique that should be a part of every programmer's toolkit. A class can provide a public *static factory method*, which is simply a static method that returns an instance of the class. Here's a simple example from `Boolean` (the *boxed primitive* class for `boolean`). This method translates a `boolean` primitive value into a `Boolean` object reference:

```
public static Boolean valueOf(boolean b) {  
    return b ? Boolean.TRUE : Boolean.FALSE;  
}
```



- Bir nesne döndürmesi beklenen bir metodun nesne döndürmemesi durumu nasıl tasarlanmalıdır?
  - **null** döndürüp, çağrıran kodun null kontrolü yapmasıyla
  - Sıradışı durum fırlatarak
  - **Optional** vb. farklı dillerdeki çözümlerle mi?



```
public interface CustomerRepository{  
    Customer fetchCustomerById(int id);  
    ...  
}
```

```
Customer customer = customerRep.fetchCustomerById(id);  
if(customer != null)  
    showCustomerInfo(customer);  
else  
    showErrorForCustomerInfo(id);
```



```
public interface CustomerRepository{  
    Customer fetchCustomerById(int id)  
        throws NoSuchCustomerException;  
    ...  
}
```

```
try{  
    Customer customer = customerRep.fetchCustomerById(id);  
    showCustomerInfo(customer);  
}  
catch(NoSuchCustomerException e){  
    showErrorForCustomerInfo(id);  
}
```



```
public interface CustomerRepository{  
    Optional<Customer> fetchCustomerById(int id);  
    ...  
}
```

```
Optional<Customer> optional = repository.fetchCustomerById(id);  
if(optional.isPresent())  
    showCustomerInfo(optional.get());  
else  
    showErrorForCustomerInfo(id);
```



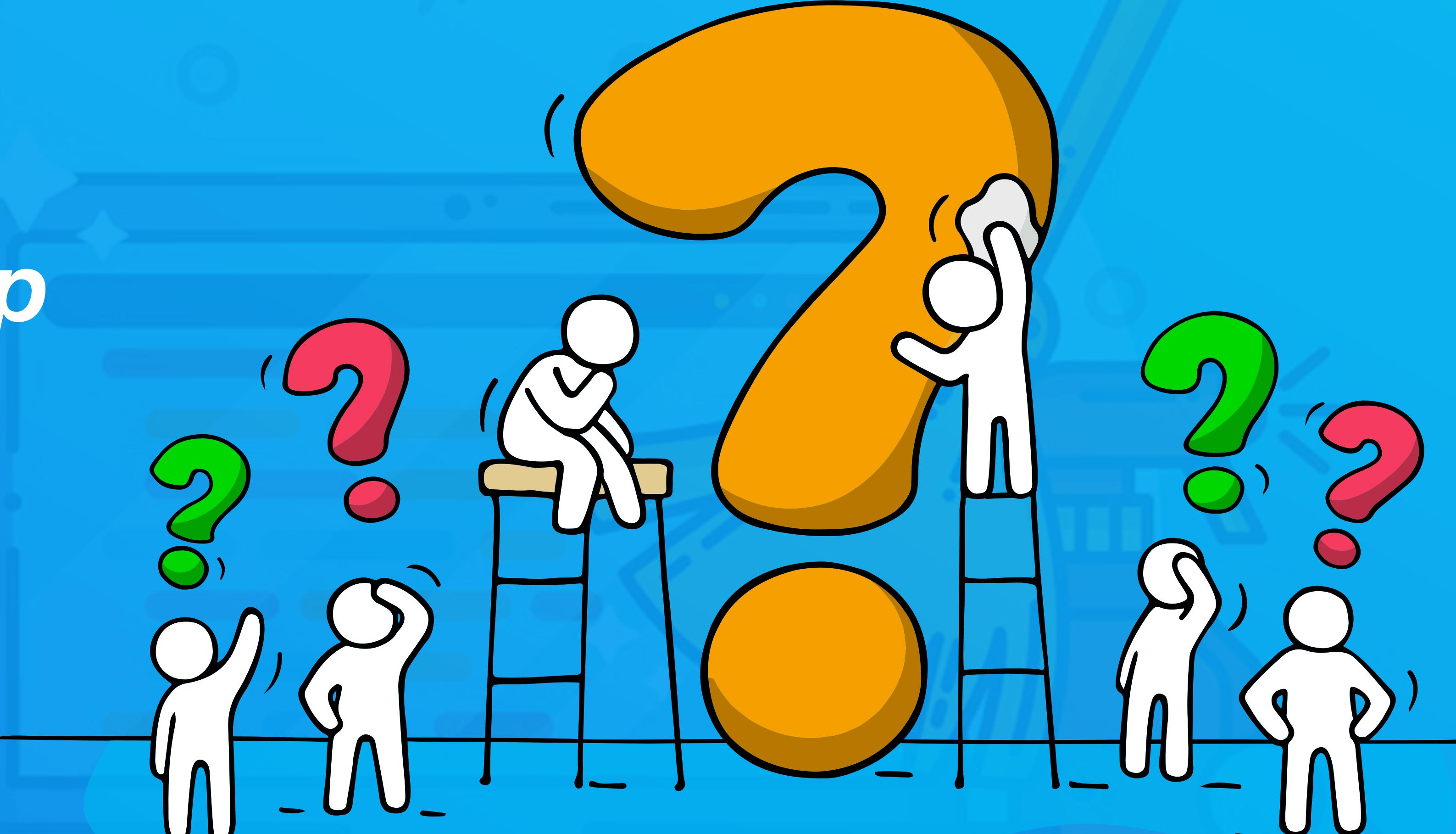
- Tasarım kalıpları da (design patterns), dillerden bağımsız, sık tekrarlanan problemlere genel çözümler sunarlar.
- Nesne yaratmak ya da durumlu bir nesnenin hayat döngüsünü yönetmek gibi.
- Tasarım kalıpları zengin bir nesne rol kataloğu kazandırır.
- Tasarım kalıplarını yaygın bir şekilde kullanmak, dili daha etkin kullanmanızı, daha standart ve kaliteli koda nihayetinde de temiz koda sahip olmanızı sağlar.

# Çerçeveler (Frameworks)



- Çerçeveler de (frameworks) kullandıkları yapılar ve kalıplar ile kodların olabildiğince standart olmasını sağlarlar.
- Çerçeve kullandığınızda, onun yaklaşımlarına ve kullanımlarına uyun.

# *Soru ve Cevap Zamani!*





**s** selsoft  
build better, deliver faster

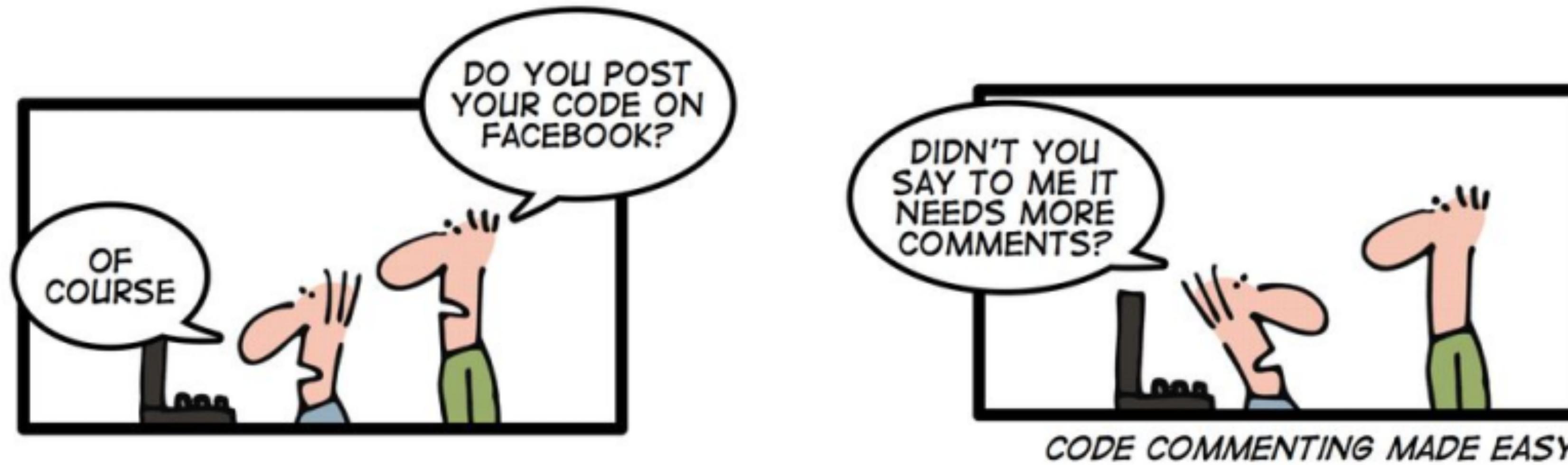
**Dokümantasyon**

# Dokümantasyon



- Anlaşıllır kodun son şartı, gerektiği kadar dokümante edilmiş (commented - documented) olmasıdır.
- Anlaşıllır kod için iyi isimlendirme ve odaklı olmaktan kaynaklanan kısalık yeterli değilse, kod açıklanmalıdır.
- Kodu açıklamak, kod dokümantasyonu (code documentation) ile olur.

# Gerekli Yorum



# Dokümantasyon Gerekli mi? - I



- Eğer bir kodun anlaşılması için dokümantasyona ihtiyaç duyuluyorsa, bunun çözümü öncelikle dokümantasyon değildir.
- Kodda yazılan her yorumun bir özür olduğu söylenir:
  - Anlaşılmaz ve kötü bir kod yazdım, anlamamanız için yorumu da okumanız gereklidir!
  - Dolayısıyla kodunuza yorum yazarak onu anlaşılır hale getirmeyin, kodunuzu iyileştirerek (refactoring) anlaşılır hale getirin.

# Dokümantasyon Gerekli mi? - II



```
// Stupid code, annoying comment!
double tp = p * 1.18; // Multiply price with 1.18

// Stupid code, stupid code-fixing comment!
double tp = p * 1.18; // 0.18 is the ratio for VAT

// Good code, no comment!
double totalPrice = price * (1 + VALUE_ADDED_TAX_RATIO);
```

# Dokümantasyon Gerekli mi? - III



- Eğer bir sınıfın ya da değişkenin neyi soyutladığı, neyi temsil ettiği, bir metodun ne yaptığı vs. isminden anlaşılmıyorsa, öncelikle refactoring ile iyi isim verilerek anlaşılır hale getirilmelidir.
- Örneğin sınıfı bölüp, parçalamak, metot sayısını azaltmak vs. sınıfın anlaşılabilirliğini arttırmak ve dokümantasyon ihtiyacını azaltır.
- Buna rağmen halen anlaşılma problemi olacaksa kısaca ne işe yaradığı ve nasıl kullanıldığı açıklanabilir.

# Dokümantasyon Gerekli mi? - IV



- Benzer şekilde, eğer bir metodun ne yaptığı, arayüzünden, yani isminden, geçilen parametrelerden, dönüş tipi ve fırlattığı sıra dışı durumlardan anlaşılımıyorsa, önce iyileştirme denenmelidir.
- Metodu bölüp, parçalamak en temel iyileştirmekdir çünkü belki de metot çok iş yapıyordur ve isim vermek zordur,
  - Parametre sayısını, içindeki karar mekanizmalarını azaltmak vs.
- Buna rağmen halen anlaşılma problemi olacaksa kısaca ne işe yaradığı açıklanabilir ve parametreleri, dönüşü tipi ile fırlattığı sıra dışı durumlar açıklanabilir.

# Dokümantasyon Gerekli mi? - V



```
/**  
 * This method does a few things in a row.  
 * It first reads the given Excel file and convert  
 * it into list of policies. Then renew each policy  
 * in the list. If there is an error in renewing any  
 * policy it produces an error code and description  
 * and put them another array....  
 */  
public Map<Policy, Error> processExcel(...){...}
```

```
public Sheet readExcelForRenewal(...){...}
```

```
public List<Policy> readPoliciesInSheet(...){...}
```

```
public Map<Policy, Error> renewPolicy(...){...}
```

# Kod Dokümantasyonu - I



- Kod dokümantasyonunun iki türünden bahsedilebilir:
  - Kod içi dokümantasyon, ihtiyaca bağlı olarak kısa “//” ya da “/\* \*/” notları halinde yapılabilir.
  - API dokümantasyonu, kodun arayüzünün dokümantasyonudur.
  - API dokümantasyonu çok daha önemli ve stratejiktir.
  - Debugger kullanmaya harcanan vakti API dokümantasyonuna harcamak çok daha faydalıdır.

# Kod Dokümantasyonu - II



- Kod içi yorumlar daha çok karmaşık algoritmalarda geçen yerel değişkenleri, bazı adımları, kontrolleri, beklenen ve beklenmeyen durumları, hesapları vs. daha anlaşılır kılmak için yapılabilir.
- Kod içi dokümantasyon, açık olanı, görüneni değil, iyi isimlendirmeye rağmen görünmeyeni, açık olmayanı açıklamalıdır.

# Kod Dokümantasyonu - III



- İkinci tür kod dokümantasyonu, API dokümantasyonudur.
- Kodun anlaşılmasından kasıt öncelikle APInin anlaşılmasıdır.
- API dokümantasyonu sınıf/arayüz ve metodlar için yapılır.
  - Metotların parametreleri, varsa dönüş değeri ve fırlattığı sıra dışı durumlar açıklanmalıdır.
- Özel formatı ve araçları vardır.

# Java API Dokümantasyonu - I



- "**/\*\*     \*/**" Javadoc ile dokümantasyon oluşturmak amacıyla kullanılan yorumdur.

```
/**  
 * Bu birden fazla satırı <b>Javadoc</b> yorumu yapmanın  
 * yoludur. Javadoc, bir Java aracı ve iç  
 * dokümantasyon oluşturma formatıdır.  
 * @see http://www.javaturk.org  
 * @see http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html  
 * @author Akin Kaldiroglu  
 */
```

# Java API Dokümantasyonu - II



```
/**  
 * A <code>Bulb</code> is a bulb. This class simulates a bulb. It has a default  
 * constructor that creates a bulb with a max luminescence of 60 and another  
 * constructor that creates a bulb with specified max luminescence.  
 * <p> This class has methods to turn on and turn off the bulb and to increase and  
 * to decrease the current luminescence of the bulb.  
 *  
 * @author Akin Kaldiroglu  
 * @author James Gosling  
 * @version 1.3.7  
 * @see <a href="http://www.javaturk.org">JavaTurk.org</a>  
 * @see "To better understand check Internet as to how a bulb works!"  
 * @since 1.2  
 */  
public class Bulb { ... }
```

```
/**  
 * Brightens the bulb to specified value.  
 *  
 * @param value New luminescence of the bulb.  
 * @exception IllegalArgumentException When passed value is negative.  
 * @throws BulbBlownException When passed value is greater than {@link #STANDARD_MAX_LUMINESCENCE}  
 * argument.  
 */  
public void brighten(int value) throws IllegalArgumentException, BulbBlownException {  
    ...  
}
```

# C# API Dokümantasyonu - I



- C#'da API dokümantasyonu, sınıf ve üyeleri için özel formatta yazılan açıklamalarla yapılır.
- Documenting your code with XML comments:
  - <https://docs.microsoft.com/en-us/dotnet/csharp/codedoc>

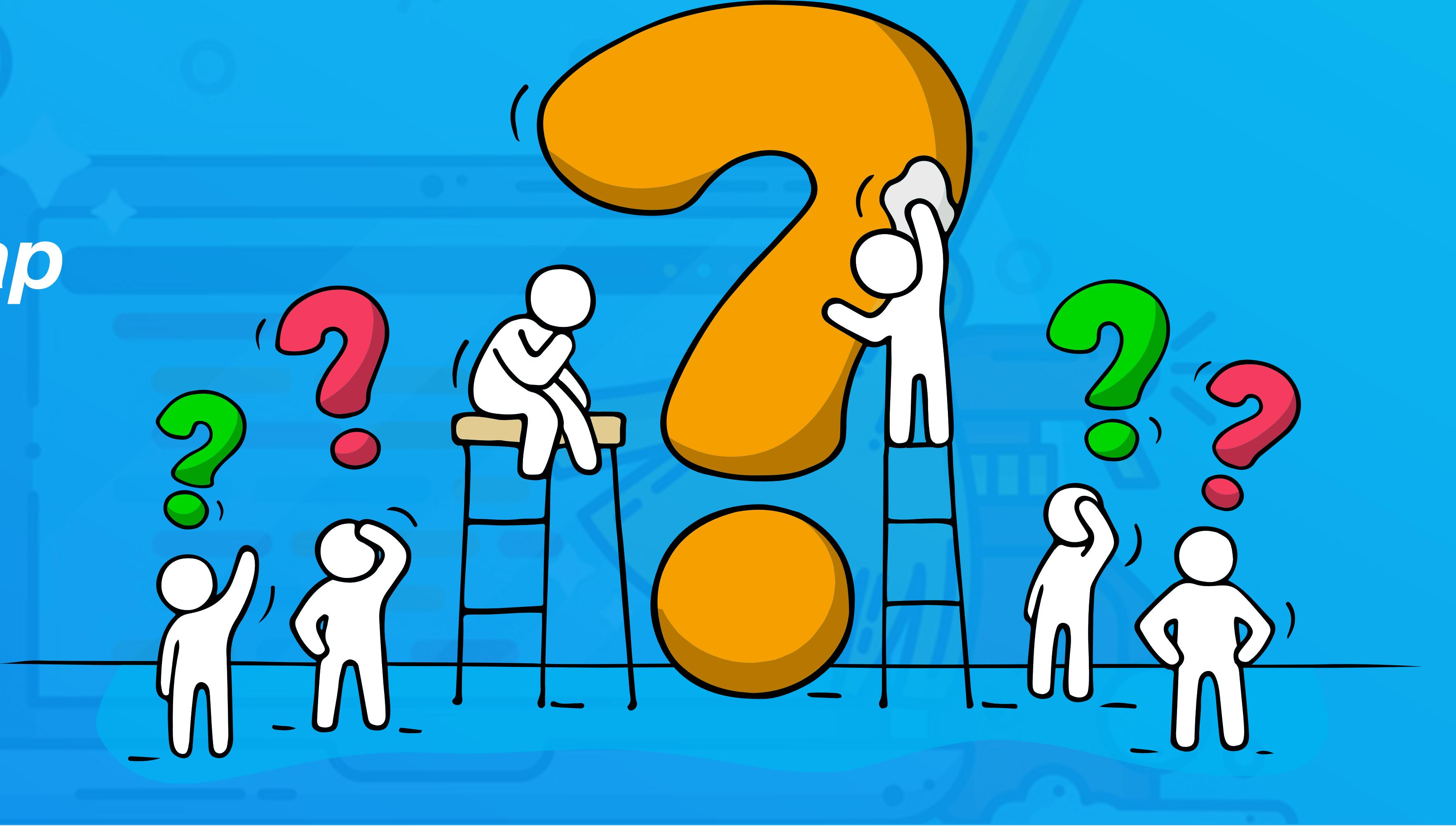
# C# API Dokümantasyonu - II



```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{ ... }
```

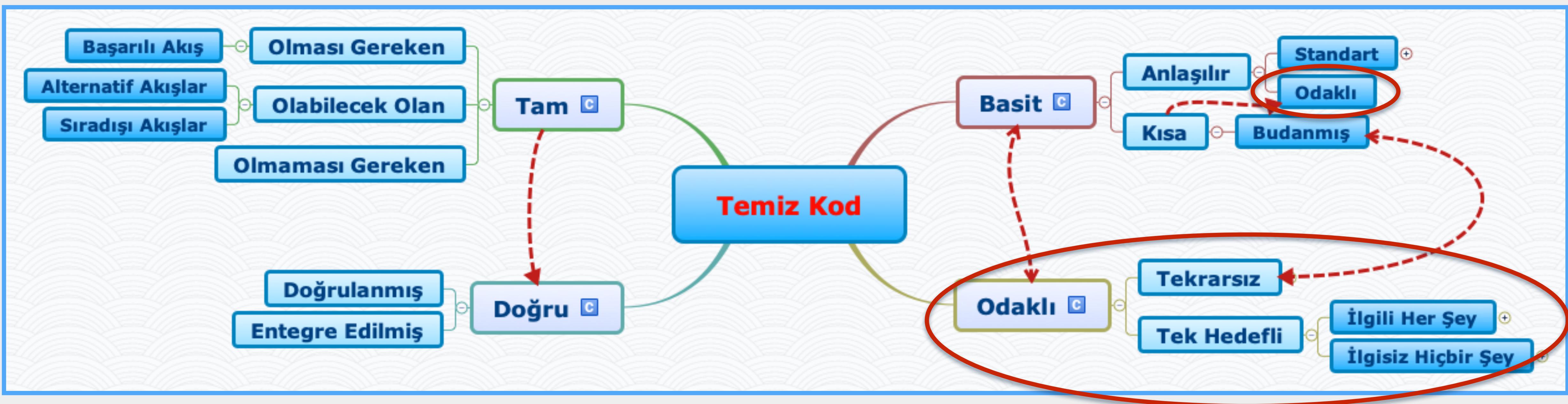
```
/// Adds two doubles and returns the result.
/// </summary>
/// <returns>
/// The sum of two doubles.
/// </returns>
/// <exception cref="System.OverflowException">Thrown when one parameter is max
/// and the other is greater than zero.</exception>
/// See <see cref="Math.Add(int, int)"> to add integers.
/// <param name="a">A double precision number.</param>
/// <param name="b">A double precision number.</param>
public static double Add(double a, double b)
{ ... }
```

# *Soru ve Cevap Zamani!*



# Odaklılık

# Temiz Kod Çerçevesi



# Anlaşırlır Kod - içerik Şartları - I



- Anlaşırlırlık, sekilden çok içerikle ilgilidir.
- İçerik açısından bakıldığından anlaşır kod, olabildiğince okuyanın konuya çok hakim olmasına gerek bırakmadan, çok detaya girmeden, nispeten yüzeysel olarak hızlıca kavranan koddur.
- Dolayısıyla öncelikle programcı, kendisi için anlaşır kod yazmalıdır.
  - Kodunu yazdıktan sonra dön bir bak, 3 ay ya da 1 yıl sonra bu kodu anlaman, hatasını bulman ve düzeltmen ne kadar sürer?

# Anlamak İçin Kod



- **Martin Fowler**, Refactoring isimli kitabında şöyle der:

**Herhangi bir insan bilgisayarın anlayabileceği kod yazabilir. İyi programcılar ise insanların anlayabileceği kod yazar.**

**Any fool can write code that a computer can understand. Good programmers write code that humans can understand.**

# Anlaşıllır Kod - İçerik Şartları - II



- Anlaşıllır kodun en temel içerik özelliği, odaklı olmasıdır.
- Odaklı kod, çatı (framework)-bileşen-paket-arayüz-sınıf-metot/fonksiyon-blok-değişken, ne olursa olsun, sadece ve sadece bir şeyi halleder.
- Odaklı kod, bir yerde birden fazla şeyi bir araya getirmez.
- Odaklı kod aynı zamanda kısadır, küçüktür.
- Yazılımda hiç bir soyutlamanın büyüğü makbul değildir.
- Kodlama, anlama, anlatma, test, değişim vs. için odaklı kod şarttır.

# Odaklılık İlkesi



- Odaklı kodun iki özelliği vardır:
  - **Tek hedefli**
  - **Tekrarsız**
- Tek hedeflilik, bir seferde sadece bir şeyi hedeflemek, onu yapmak, halletmek demektir.
- Bir şeyi değiştirmenin sadece bir sebebi olmalıdır.
- Yapılan işi sadece ve sadece bir yerde ve bir kere yapmak, tekrarsız kodun gereğidir.

# Odaklı Kod





- Odaklı kodun elde etmek için geliştirilmiş prensipler ve teknikler vardır.

**Separation of concerns**

**Single-responsibility principle**

**High-cohesion**

**Low-coupling**

- Bu prensipler daha detaylı olan teknikleriyle birlikte ileriki bölümlerde ele alınacaklar.

# Doğru Soyutlama Seviyesi



- Bu prensipleri uygularken doğru soyutlama seviyesini belirlemek çok önemlidir:
  - Bir pakette bir araya getirilecek sorumluluklar grubunu bir sınıfa sığdırmak,
  - Bir sınıfta ya da bir kaç metotta yapılacak sorumlulukları birleştirip bir metotta yapmak,
  - Bir metotta yapılacakları bir bloka hatta satıra sığdırmak,
  - Bir blokta yapılacak adımları, tek bir satırda, bir adım olarak yapmak
  - çok sık rastlanan türden yanlış soyutlamalardır.

# Kısa Kod İçin Budama



- Odaklı dolayısıyla da küçük soyutlamalar bir seferde elde edilemez.
- Soyutlamalarınızı devamlı surette budayarak kısa ve küçük soyutlamalara ulaşabilirsiniz.

## Sürekli İyileştirme ya da Continuous Refactoring

- Unutmayın, mükemmellığa ekleyerek değil, çıkararak ulaşabilirsiniz.

# Kısa Kod İçin Budama



- **B. Pascal** The Provincial Letters isimli eserinin 16. mektubunda şöyle der:

I have only made this letter longer because I have not had the time to make it shorter.

Bu mektubu böyle uzun yapabildim çünkü kısa tutmak için zamanım yoktu.

# Tekrarsız Kod



- Odaklanmak aynı zamanda, bir şeyi sadece bir yerde yapmak, birden fazla yerde yapmamak, tekrardan kaçınmak, tektrsız kod yazmak demektir.
- Tekrarsız kod, ancak **tasarımla** ve sonrasında gerekiğinde **refactoring** ile elde edilir.
- Merkezi ve ortak bir tasarımla şekillenmeyen, sonrasında ciddi bir koordinasyon ve iletişime ortamında yazılmayan kodların pek çok yerde tekrara sahip olması kaçınılmazdır.

# Don't Repeat Yourself - I



- Don't Repeat Yourself, DRY, tekrarsız kod yazmanın sloganıdır:

**Every piece of knowledge must have a single, unambiguous,  
authoritative representation within a system.**

**Her bilgi parçasının sistemde, tek, açık, güvenilir bir ifadesi olmak  
zorundadır.**

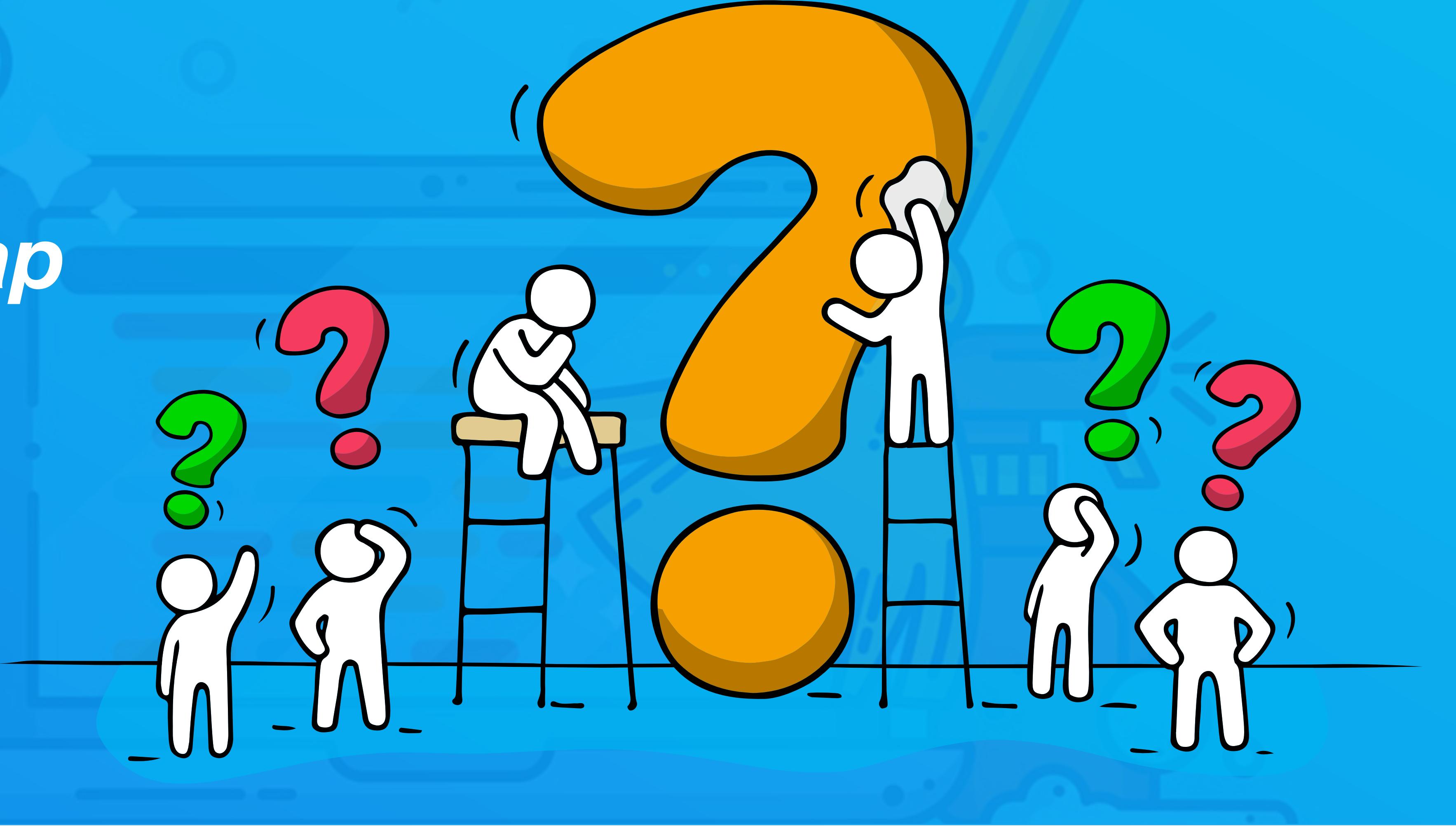
- Davranış, bilgi, kısıt, vs. ne olursa olsun sadece bir yerde olsun.

# Don't Repeat Yourself - II



- DRY prensibini sadece tasarım ile gerçekleştirmek imkansızdır.
- Kodlarken refactoring ile tekleştirmeye gidilmelidir.
- Tekleştirme, cut-paste ile sağlanır.
  - Birbirine benzeyen kodlar, tek bir yapı altında toplanır.
  - Tekrar kullanım (reusability) artar.
- **Copy-paste kullanmayın, cut-paste kullanın!**

# *Soru ve Cevap Zamani!*



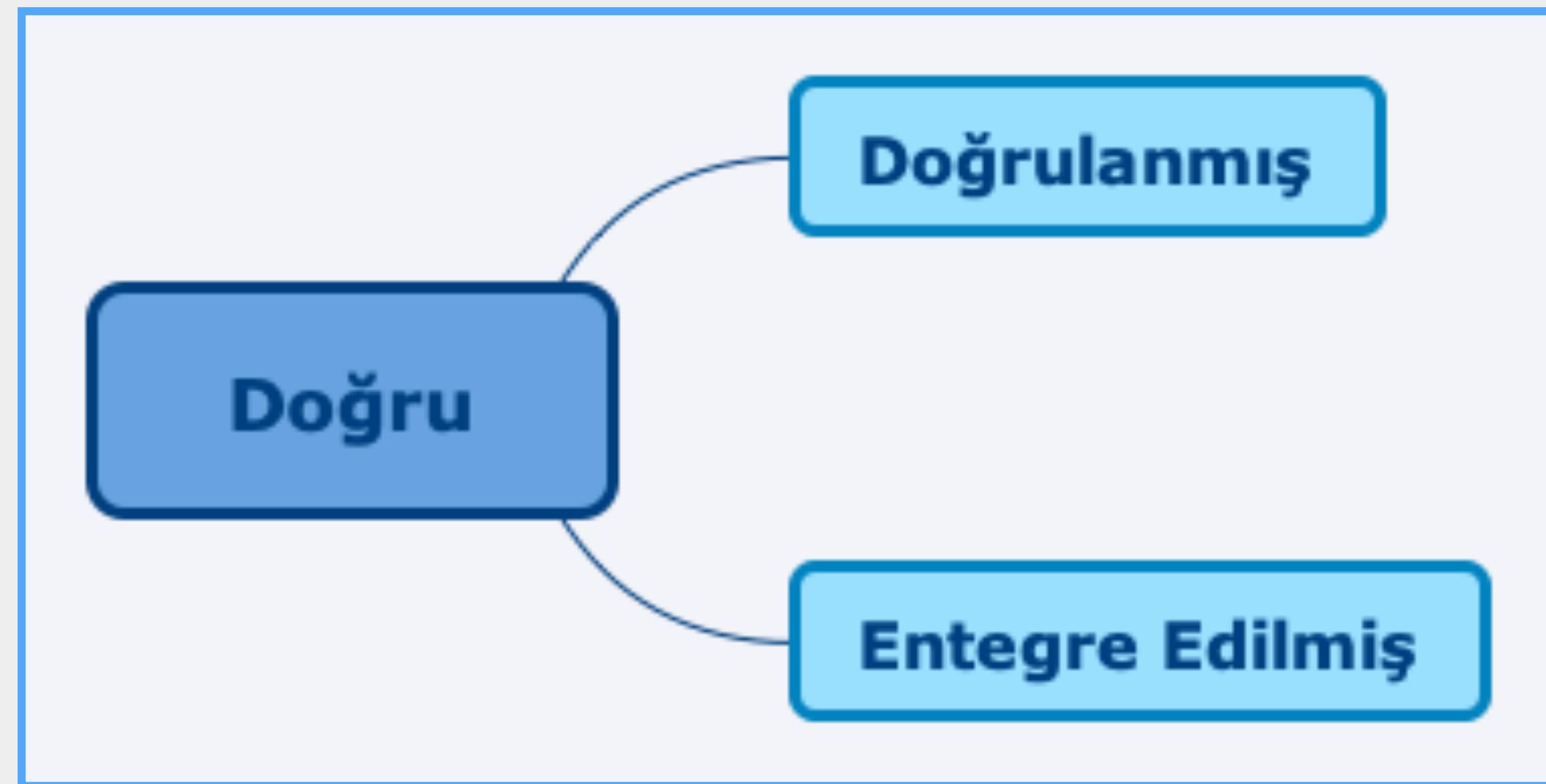
# Doğruluk

# Doğru Kod - I

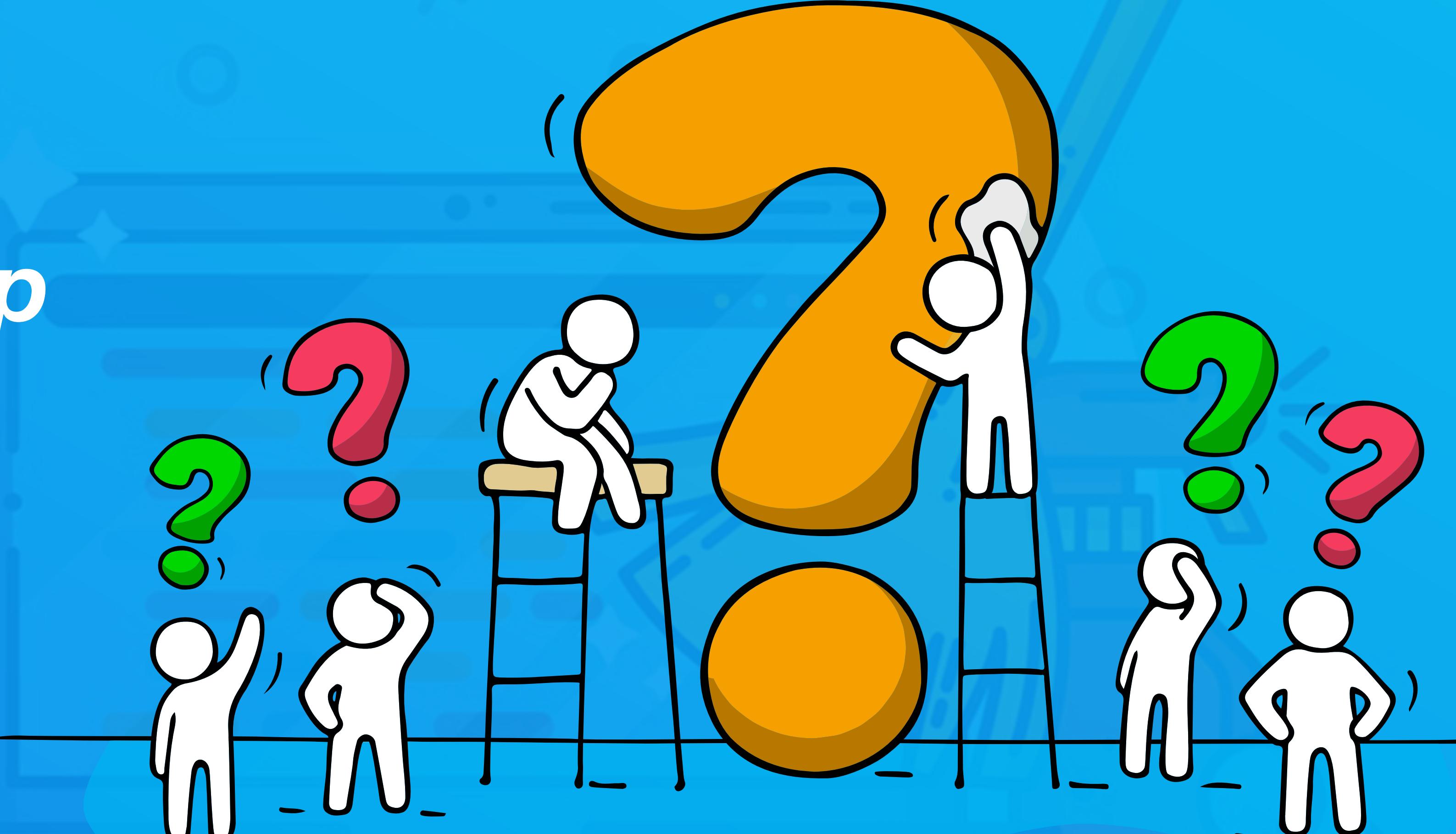


- Temiz kod doğru olmalıdır.
- Bir kodun kendi başına doğruluğu ancak **birim testiyle (unit test)** sağlanabilir.
- Bir kodun çevresindeki diğer kod parçalarıyla birlikte doğru çalıştığı ise ancak **entegrasyon testiyle** sağlanabilir.
- Birim ve entegrasyon testleri TDD yaklaşımı ile yapılrsa, doğruluk ve tamlık konusunda iyi seviyeler elde edilebilir.
- Kod doğrulama yaklaşımı olarak TDD ve birim testleri ileride ele alınacaktır.

# Doğru Kod - II



# *Soru ve Cevap Zamani!*



# Tamlik



**selsoft**  
build better, deliver faster



- Temiz kod, tam olmalıdır.
- Kodun tam olmasının üç şartı vardır:
  - Olması gerekeni yapması,
  - Olmaması gerekene karşı önlem alması, engellemesi,
  - Olabilecek olanı da öngörmesi.
- Sadece ilk maddeye odaklanmak kodunuzu tam yapmaz.

# Tam Kod - II



# Tam Kod - III



- Kodun basit, odaklı ve doğru olması da tamlığını garanti etmez.
- Temiz kodun tamliğini sağlamak üzere sıra dışı durum yönetimi ve savunmacı programlama vb. teknikler ileride ele alınacaktır.

# Bölüm Sonu

*Soru ve Cevap  
Zamani!*

