Tasarım Kusurlarının (*Design Flaws, Defects, Smells*) Belirlenmesi

Kaliteli (doğru) bir tasarımın net bir tanımını yapmak ve bunu formüllerle (sayılarla) ifade etmek zor olsa da yıllar içinde deneyimle oluşmuş bazı görüşeler vardır.

Tasarım kusurları (ya da kod kusurları) kaliteli bir tasarımın sahip olması gereken özelliklerden sapmalar olarak tanımlanırlar.

Tasarım kusuru içeren program modülleri derlemede ya da programın çalışması sırasında hata vermeyebilirler.

Ancak bu yapısal kusurlar -türlerine bağlı olarak- aşağıdaki sorunlara neden olurlar ve özellikle bakım maliyetini yükseltirler.

- Yazılımın esnekliğini azaltırlar.
  - Yazılımı değiştirmek (değişen isteklere uyarlamak) zordur.
  - Yeni özellikler eklemek zordur.
- Tekrar kullanılabilirliği azaltırlar.
- Olası hatalara açıktırlar.
  - Sisteme yapılacak yeni eklemeler (ya da değişiklikler) kusurlu modüllerde sıklıkla mantık hatalarına neden olur.
- Hataları gizlerler.
  - Hatanın kaynağını (hangi sınıf, hangi metot) bulmak kolay değildir.

Tasarım kusurları, tasarımdaki kötü kokular (***bad smells***) olarak da adlandırılırlar.

Tasarım Kusurlarına Örnekler

Nesneye dayalı tasarımlarda sık karşılaşılan kusurlarla ilgili deneyimle oluşmuş görüşler bulunmaktadır.

Yazılım dünyasında isimlendirilmiş tasarım kusurlarına ilişkin örnekler:

- God Class: Karmaşık, uyumu kötü (fazla sorumluluk yüklenmiş) ve başka sınıfların verilerine yoğun olarak erişen sınıf
- Schizophrenic Class: Birden fazla iş yüklenmiş, ara yüzünde farklı konularda hizmetler barındıran, uyumu kötü sınıf
- Data Class: Anlamlı bir işi (sorumluluğu) olmayan, büyük ölçüde verilerden oluşan ve bu verileri doğrudan dışarıya açan sınıf
- Brain method: Çok fazla iş yapan, uzun, fazla dallanma ve fazla iç içe blok içeren karmaşık metot. Sınıfın bütün sorumluluğu tek bir metoda yüklendiğinde ortaya çıkar.
- Intensive Coupling: Bir metot çok sayıda başka metodu çağırıyor ve çağırılan bu metotlar başka sınıfta toplanmışlar.
- Shotgun Surgery: Sınıfın bir metoduna farklı çok sayıda sınıftan çağrı yapılıyor.
- Refused Parent Bequest: Alt sınıf, üst (taban) sınıflardan aldığı veri ve metotlardan yararlanmıyor.

Tasarım Kusurlarının Nedenleri:

1. Tasarımın iyi düşünülmeden kodlamaya geçilmiş olması. Tasarım kalıplarının (*design patterns*) dikkate alınmaması.

2. Zaman baskısı nedeniyle projenin gelecekteki durumu düşünülmeden anlık "çözümler" bulunması.

Teknik borç (*Technical debt*):

Tasarıma yeterli zaman ayrılmadığında başta zaman kazanıldığı düşünülse de daha sonra (bakım aşamasında) ödenecek bir borç (iş gücü) oluşur.

Bu borç tasarımdaki kusurlar nedeniyle oluşur.

Çoğunlukla daha sonra tasarımı düzeltmek için harcanan süre (borç) başlangıçta tasarımın doğru bir biçimde oluşturulmak için harcanandan daha fazladır.

Cunningham: "The debt incurred through the speeding up of software project development which results in a number of deficiencies ending up in high maintenance overheads".

McConnell:"A design or construction approach that's expedient in the short term but that creates a technical context in which the same work will cost more to do later than it would cost to do now (including increased cost over time)".

Tasarım kusurlarının metriklerle belirlenmesi (*Detection Strategies*)

Kusurların belirlenmesinde modellere gereksinim vardır.

- Model, tasarım kusurunu açık biçimde tanımlamalı (tarif etmeli);
- Alt düzeydeki küçük taneli (*fine-grained*) metrikler ile üst düzeydeki kusurlar arasında ilişki kurmalı.

Yöntemlerin Türleri:

1. Kural Tabanlı Yöntemler:

Belli bir kusuru bulmak için metriklerden oluşan kurallar belirlenir.

Örneğin; M1 > 150 ve M2 < 0.3 → A tipi kusur vardır.

Eğer bir sınıfın (veya metodun) metrik değerleri bu kurala uyuyorsa o sınıfta (metotta) belli bir kusur var demektir.

2. Öğrenme Tabanlı Yöntemler:

Kusurlu/kusursuz olduğu bilinen sınıfların (veya metotların) metrikleri kullanılarak eğitim ile bir model oluşturulur (lojistik regresyon, karar ağacı, Bayes ağı).

Bu model daha sonra başka sınıflarda (metotlarda) kusur olup olmadığını belirlemek için kullanılır.

Tasarım kusurlarının kural tabanlı yöntemlerle belirlenmesi

Modelin oluşturulması:

Kusurları bulurken kullanılacak model GQM yöntemi ile oluşturulabilir.

Hedef (G) : Belli bir kusuru bulmak (kusurun tanımı yapılır)

Kusurun tanımı yapılırken iyi tasarımın özellikleri dikkate alınır, bu özelliklerden yaygın olarak karşılaşılan sapmalar kusur olarak tarif edilir.

Sorular (Q): Kusurun tanımında yer alan sorular (belirtiler "*symptoms*") sorulur.

Örneğin: Metot karmaşık mı?

Sınıfın uyumu düşük mü?

Metot çok sayıda başka metodu çağırıyor mu?

Metrikler (M): Bu soruların yanıtlarını verecek metrikler belirlenir.

Birden çok sorunun yanıtlarının nasıl bir araya getirilip kusurun varlığı/yokluğu konusunda nasıl karar verileceğine ilişkin bir denklem oluşturulur.

Karar aşamasındaki denklemlerde metrikler üzerinde aritmetik ve/veya mantıksal işlemler yapılır, sonuçlar önceden belirlenen bazı eşik değerleri ile karşılaştırılır.

Eşik değerleri iyi tasarımların özelliklerini temsil ederler.

Örnek: "God Class" belirlemek için örnek bir modelin oluşturulması

- Hedef (G) : "God Class" belirlemek. Karmaşık, uyumu kötü ve başka sınıfların verilerine yoğun olarak erişen sınıf
- Sorular (Q):
  - Q1. Sınıfın karmaşıklığı yüksek mi?
  - Q2. Sınıfın uyumu kötü mü?
  - Q3. Sınıf başka sınıfların verilerine yoğun olarak erişiyor mu?
- Metrikler (M):
  - M1. WMC (*Weighted methods per class*)
  - M2. TCC (*Tight Class Cohesion*)
  - M3. ATFD (*Access to Foreign Data*):
- Metrik-Hedef ilişkisi:
  - God Class = (WMC > çok yüksek) VE (TCC < düşük) VE (ATFD > AZ)

Buradaki sorun eşik değerlerinin belirlenmesidir.

"Az", "çok", "çok yüksek" hangi değerlere karşı düşmektedir?

Örnek bir çalışma:

Bu konuda bir çok çalışma yapılmış ve yayımlanmıştır.
Ders kapsamında ilk olarak aşağıdaki kitaptaki çalışma ele alınacaktır.

M. Lanza, R. Marinescu, *Object-oriented metrics in practice : using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer, 2006.

Bu kitapta tasarım kusurlarına tasarım uyumsuzlukları (*disharmonies*) adı verilmiştir.
Kitapta 11 adet kusur tanımlanmış ve her birinin belirlenmesi için metrikler kullanan kural tabanlı bir model önerilmiştir.
Ayrıca bu kusurların düzeltilmesi ile ilgili önerilerde kitapta yer almaktadır.

Metrikler ve Eşik Değerleri (*Thresholds*)

Metriklerin değerlerinin doğru bir biçimde yorumlanabilmesi için anlamlı eşik değerlerine (referans noktalarına) gerek vardır. Normal, az, çok, yüksek, düşük vs.
Örneğin,
Bir insan hangi durumda uzun boyludur? 1.85m ya da 2m'den mi uzunsa?
Bir yazılım metodunda "normalde" kaç satır olur?

Yazılım dünyasında çok doğru ve kesin eşik değerleri belirlenemese de pratikte kullanılabilecek değerlerin belirlenebilmesi amaçlanır.
Metrikler için eşik değerleri belirlerken iki kaynaktan yararlanılır:

1. İstatistiksel Yöntem:

Var olan yazılım projeleri incelenir ve belli metrik değerlerinin istatistiksel dağılımı (ortalama "AVG", ortanca değer (*median*), standart sapma "STDEV") belirlenir.

2. Anlamsal Eşik Değerleri:

Deneyimler (gözlemler) sonucu oluşmuş, genelde kabul gören eşik değerleri.
Örneğin iç içe 3 veya daha az döngü "normal" daha çoğu "fazla" olarak nitelendirilebilir.

Metrikler ve Eşik Değerleri (*Thresholds*) (devamı)

1. İstatistiksel Yöntem:

Var olan yazılım projeleri incelenir ve belli metrik değerlerinin istatistiksel dağılımı (ortalama "AVG", ortanca değer (*median*), standart sapma "STDEV") belirlenir.

a) Ortalama ve standart sapma kullanılması:

Eğer metrik en azından aralık (*interval*) ölçeğinde ise ortalama ve standart sapma kullanılabilir.

Buna göre aşağıdaki sınıflandırma yapılabilir:

Düşük: AVG - STDEV

Yüksek: AVG + STDEV

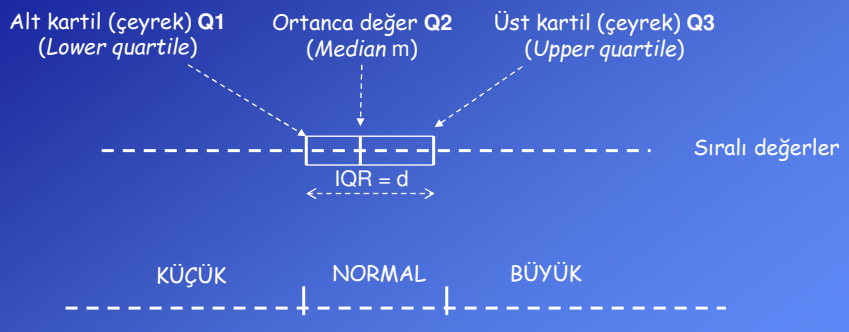
Çok Yüksek: (AVG + STDEV) · 1.5

Ortalamanın kullanıldığı bu tür eşikler daha çok boyutla ilgili ve sayma yoluyla elde edilen metrikler için uygundur. Örneğin satır sayısı, metot sayısı gibi.

b) Eşik değerlerinin belirlenmesinde ortanca değer (medyan) kullanımı:

Sırasal ölçekte değer üreten metrikler için ortalama yerine medyan kullanılmalıdır.

Örneğin **Kutu Çizimi (*Box plot*)**:

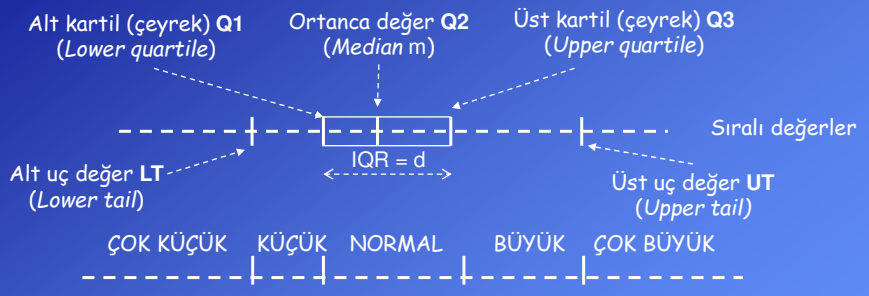


Q2 (m): Ortanca değer (*median*). Değerler sıralandığında ortadaki elemanın değeri
Q1: Alt çeyrek. Ortanca değerden düşük değerdeki elemanların ortanca değeri
Q3: Üst çeyrek. Ortanca değerden büyük değerdeki elemanların ortanca değeri
IQR: InterQuartile Range

b) Ortanca değer (medyan) kullanımı (devamı):

En az aralık ölçeğinde değer üreten metrikler için de medyan kullanılabilir.
Böylece aşırı değerlere sahip birimler (*outlier*) değerlendirme dışı bırakılabilir.

Örneğin **Kutu Çizimi (*Box plot*)**:



Q2 (m): Ortanca değer (*median*). Değerler sıralandığında ortadaki elemanın değeri
Q1: Alt çeyrek. Ortanca değerden düşük değerdeki elemanların ortanca değeri
Q3: Üst çeyrek. Ortanca değerden büyük değerdeki elemanların ortanca değeri
LT: Alt uç değer. LT = Q1 - 1.5*d .
UT: Üst uç değer. UT = Q3 + 1.5*d .

Metrik en az aralık ölçeğinde olduğu için bu şekilde aritmetik işlemler yapılabilir.

Örnek çalışmada istatiksel yöntemin kullanılması:

45 Java ve 37 C++ projesi incelenerek istatistik değerleri elde edilmiş ve bazı metrikler için eşik değerleri elde edilmiştir.

Projeler değişik uygulama alanlarından ve değişik boyutlarda (satır sayısı: 20,000 - 2,000,000) seçilmiştir.

Yazılımların bir kısmı ücretli, ticari yazılımlar, bir kısmı ise açık kaynaklı yazılımlardır.

Eleştiri:

- Mantıklı ve kabul edilebilir eşik değerleri elde etmek için fazla hata içermediği ve güvenilir olduğu düşünülen belli yazılım projelerinin incelenmesi gerekmez mi?
- Eşik değerleri uygulama alanına bağlı olabilir. Eşik değerlerinin belli alanlardaki projelerde ayrı ayrı toplanması gerekmez mi?

Örnek çalışmada istatiksel yöntemin kullanılması (devamı):

İncelenen metrikler:

- Bir sınıftaki ortalama metot sayısı (NOM/Class)
- Bir metottaki ortalama kod satırı sayısı (LOC/Method)
- Kod satırı başına düşen dallanma noktası (CYCLO/LOC)

Seçilmelerinin nedenleri:

- Projenin boyutu ve karmaşıklığı hakkına bilgi veren temel metriklerdir.
- Birbirlerinden bağımsızdırlar; farklı bilgiler verirler.
- Projenin boyutundan bağımsızdırlar.

İstatistiksel değerler ve eşikler:

- Ortalama (AVG) metriklerin tipik değerlerini verir.
- Standart sapma (STDEV) değerlerin ne kadar dağıldığını gösterir.
  Not: Standart sapma, varyansın kare köküdür.
- Alt sınır: AVG - STDEV
- Yüksek sınır: AVG + STDEV
- Çok Yüksek: (AVG + STDEV)·1.5

Örnek çalışmada istatiksel yöntemin kullanılması (devamı):

Elde edilen eşik değerleri:

	Java				C++			
Metric	Low	Ave-rage	High	Very High	Low	Ave-rage	High	Very High
CYCLO/Line of Code	0.16	0.20	0.24	0.36	0.20	0.25	0.30	0.45
LOC/Method	7	10	13	19.5	5	10	16	24
NOM/Class	4	7	10	15	4	9	15	22.5

Bu eşik değerleri ilgili metriklerden türetilebilen diğer metriklerin eşik değerlerini hesaplamak için de kullanılabilir.

Örnek: CK WMC (*Weighted methods per class*) metriği aşağıdaki gibi hesaplanır:

$$WMC = \frac{CYCLO}{LOC} \cdot \frac{LOC}{Method} \cdot \frac{NOM}{Class}$$

Türetilen eşik değerleri:

	Java				C++			
Metric	Low	Ave-rage	High	Very High	Low	Ave-rage	High	Very High
WMC	5	14	31	47	4	23	72	108
AMW	1.1	2.0	3.1	4.7	1.0	2.5	4.8	7.0
LOC/Class	28	70	130	195	20	90	240	360
NOM/Class	4	7	10	15	4	9	15	23

AMW: *Average Method Weight* CYCLO /Method veya WMC/NOM

2. Anlamsal Eşik Değerleri:

Deneyimler (gözlemler) sonucu oluşmuş, genelde kabul gören eşik değerleri.

Örneğin:

Üç ve daha az katı olan bina kısadır. 10'dan fazla katı olan bina yüksektir. 20'den fazla katı olan bina çok yüksektir.

Yüz metre mesafedeki bir yer yürümek için yakındır. İki kilometre mesafedeki bir yer yürümek için uzaktır.

Aslında bunlar da uzun süreler içinde yapılan gözlemlerin istatistiksel değerlendirmelerine dayanır.

Yazılımlar için de çeşitli algılar vardır.

Örneğin:

İç içe 3 veya daha az döngü "normal", daha çoğu "fazla" olarak nitelendirilebilir.

Bir metodun üç veya daha az parametresi olması normaldir. Üç, altı arası parametre fazladır. Altıdan çok parametre çok fazladır.

2. Anlamsal Eşik Değerleri (devamı):

Anlamsal eşikler ikiye ayrılabilir:

A. Genel kabul gören kesirler: 1/2, 1/3, 2/3, 3/4 veya 0.5, 0.33, 0.75 gibi

Değerleri 0 - 1 arası değişen normalize metriklerle çalışıldığında bu tür kesirlerin eşik değeri olarak kullanılması önerilir.

Neden?
Zaten kusur belirleme algoritması bir bilgisayarda koşacak.
İnsan gözüne (aklına) tanıdık gelen kesirlerin kullanılmasına ne gerek var?

B. Genel kabul gören mutlak değerler:

0 - 7 arası değişen tamsayılar da genel kabul gören anlamlara sahiptirler.

İnsanın kısa süreli belleğinin üst sınırının 7 olduğu gösterilmiştir.

Çalışmada kullanılan bazı anlamsal değerler.

Numeric Value	Semantic Label
0	NONE
1	ONE/SHALLOW
2 – 5	TWO, THREE/FEW/ SEVERAL
7 – 8	Short Memory Capacity

(Kısa süreli belleğin sınırı)

Yazılımın Uygunluğu (veya Uyumu) (*Harmony*)

Bir nesneye dayalı yazılımın tasarımının uygunluğu, sınıf temelli olarak ele alınan üç uyum (*harmony*) bileşeninden oluşur.

1. Kimlik Uyumu (***Identity Harmony***): Bir birimin (sınıf veya metot) var olma nedeni gerçekçi midir? Belli bir işi var mı? Çok fazla (çok az) iş mi yapıyor?
2. İşbirliği Uyumu (***Collaboration Harmony***): Bir sınıfın diğer sınıflarla etkileşimi nasıldır? Tüm işi kendi mi yapıyor? Diğer sınıflarla çok mu iletişimde bulunuyor?
3. Sınıflandırma (Hiyerarşik) Uyumu (***Classification Harmony***): Bir sınıfın kalıtım/türetim uyumu nasıldır? Kalıtımla aldığı tüm özellikleri kullanıyor mu, çoğunu değiştiriyor mu?

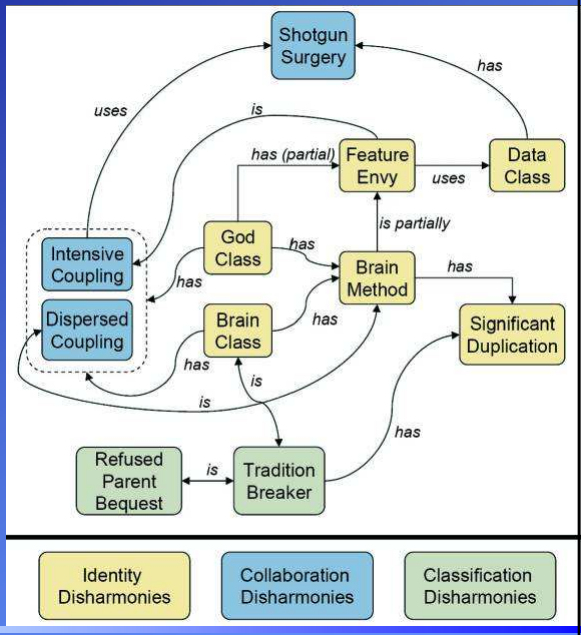
Bir yazılımdaki her birim (örneğin her sınıf);

1. Kendisiyle uyumlu olmalıdır: Çok küçük, çok büyük, çok karmaşık, çok basit olmamalıdır.
2. İşbirliği yaptığı birimlerle uyumlu olmalıdır: Çok fazla birimle iletişimde olmamalı, hiç kimse ile iletişimde bulunmadan her işi kendi yapmamalı.
3. Taban sınıfları (ataları) ve ardıllarıyla uyumlu olmalı.

Örnek çalışmada ele alınan kusurlar ve ilişkileri

Bu çalışmada kusurlar uyumsuzluk (*disharmony*) olarak adlandırılmıştır.

M. Lanza, R.Marinescu, *Object-oriented metrics in practice : using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer, 2006.



Kimlik Uyumsuzlukları (*Identity Disharmonies*)

Kimlik uyumsuzlukları tek bir birimi (sınıf, metot) etkilerler.

Bu nedenle bu tür kusurlara birimlerin bağımsız olarak incelenmesiyle belirlenebilir.

God Class, Brain Class, Brain Method, Feature Envy, Data Class, Duplication.

Kimlik uyumunun sağlayan bileşenleri:

Bir birimin kimlik uyumunu belirleyen üç bileşen vardır:

1. Boyut/Orantı (*Proportion*) Bileşeni:
   Sınıf ve metotların boyutları uygun olmalı (çok küçük ya da çok büyük olmamalı).
   Yazılımın karmaşıklığı (kod) sadece belli birimlere toplanmamalı.
2. Sunum (*Presentation*) Bileşeni:
   Her sınıf kendi kimliğini dışarıya uygun hizmetler içeren bir arayüz üzerinden yansıtmalı.
   Sınıfın tek bir konuda sorumluğu olmalı.
   Sınıftaki metotları belli bir sorumluluk konusunda yoğunlaşmalı.
   Sınıfın bazı metotları veri erişimi için (set/get) kullanılabilir.
   Ayrıca başka sınıflara görev aktaran metotlar (*delegator*) bulunabilir.
   Ancak bu tür (işlevsel olmayan) metotların oranı belli bir değeri aşmamalı, sınıfa özgü sorumlukları yerine getiren metotlar da olmalı.

2. Sunum (*Presentation*) Bileşeni (devamı):
   Sınıfın kendine özgün (diğer birimlerden farklı) davranışı olmalı.
   Kod (görev) kopyaları oluşmamalı.
   Veriler ve içişler gizli, hizmetler açık olmalı.
3. Gerçekleme (*Implementation*) Bileşeni:
   Sınıftaki veriler ve işlemler birbirleri ile ilgili ve sınıfla anlamsal bütünlük içinde olmalı.
   Tüm metotlar sınıfların niteliklerini büyük oranda kullanmalı.
   Birbirleriyle ilgileri olmayan veri ve metot kümelerini aynı sınıfta barındırmaktan kaçınılmalı.

➢ **God Class:**

"God Class" çok fazla iş yapar ve diğer sınıfların verilerini kullanır.

Sistemdeki bir çok sorumluluğun tek bir sınıfa toplanması durumda oluşurlar.

Tekrar kullanılabilirliği ve anlaşılırlığı azaltır.

"God class" belirlemek için **kullanılan metrikler:**

- **ATFD** (*Access to Foreign Data*):

Bir sınıftan, niteliklerine doğrudan ya da dolaylı olarak erişilen diğer sınıfların sayısı

Bu metrik, örnek çalışmanın yazarlarından Radu Marinescu tarafından tanımlanmıştır.

Radu Marinescu, **Detection Strategies: Metrics-Based Rules for Detecting Design Flaws**, Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM 2004).

Radu Marinescu, **Detecting Design Flaws via Metrics in Object-Oriented Systems**, Proceedings of 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39), 2001.

"God class" belirlemek için kullanılan metrikler (devamı):

- **WMC** (*Weighted Method Count*):

CK Metriği . Metotların karmaşıklığı (sayısı) hakkında bilgi verir.

- **TCC** (*Tight Class Cohesion*):

Bir sınıftaki doğrudan (sıkı) bağlı metot çiftlerinin sayısı, tüm olası metot çiftlerinin sayısına bölünür.

$$TCC = NDC(C) / NP(C)$$

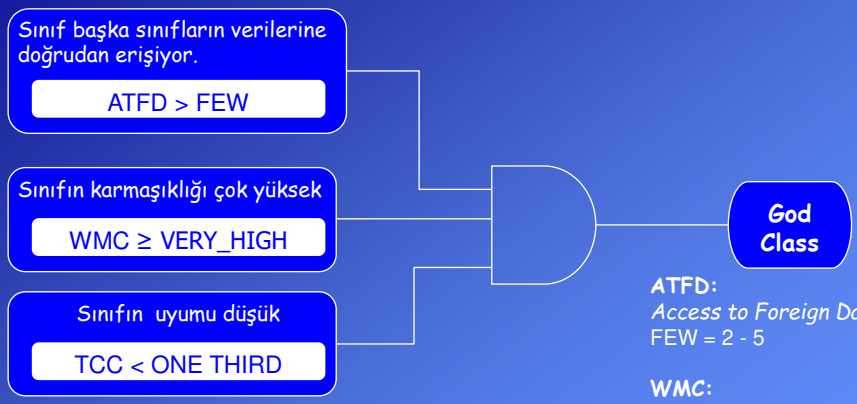
NDC: Number of directly connected methods
NP: Number of possible pairs

İki metodun doğrudan (sıkı) bağlı olması, ait oldukları sınıfın en azından bir tane niteliğini ortak kullandıkları anlamına gelir.

C sınıfında N adet metot varsa $NP(C) = N(N-1)/2$ (Olası tüm çiftler)

J.M. Bieman and B.K. Kang. "Cohesion and reuse in an object oriented system", In Proceedings ACM Symposium on Software Reusability, April 1995.

"God class" belirlemek için kullanılan yöntem:



ATFD:
Access to Foreign Data
FEW = 2 - 5

WMC:
Weighted Method Count
VERY HIGH = 47 (Java)
VERY HIGH = 108 (C++)

TCC:
Tight Class Cohesion
ONE THIRD = 0.33

Eğer sınıf başka sınıfların verilerine erişiyorsa (ATFD),
Çok sayıda (karmaşık) metodu varsa (WMC),
Metot uyumu kötü ise (TCC)
"God class" olmaya adaydır.

➢ **Feature Envy:**

(Başka sınıfların verilerine imrenmek)

Üyesi olmadığı sınıfların niteliklerine doğrudan veya metotlar üzerinden (get/set) çok erişen metotları ifade eder.

Metodun yanlış sınıfta olduğunu veya sınıf organizasyonunun kötü yapıldığını gösterir.

Belirlemek için kullanılan metrikler:

- **ATFD** (*Access to Foreign Data*):

Bir metottan niteliklerine doğrudan ya da dolaylı olarak erişilen diğer sınıfların sayısı. (Buradaki ATFD tanımı "God class" belirlemede kullanılandan farklıdır.)

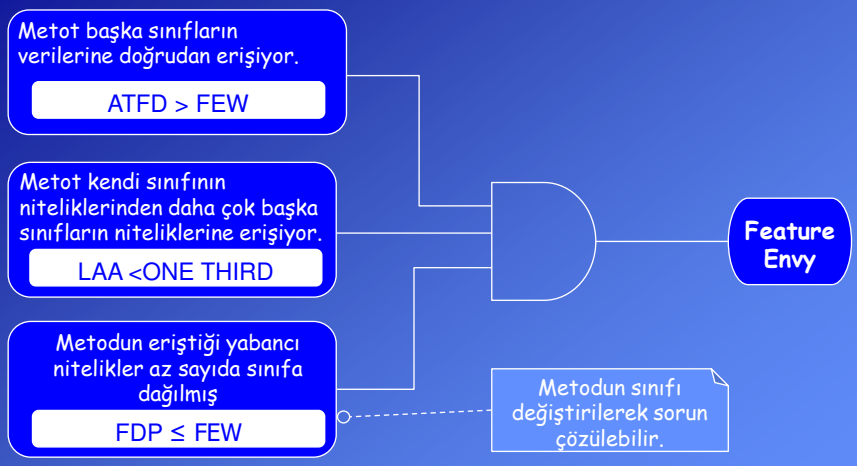
- **LAA** (*Locality of Attribute Accesses*):

Metodun kendi sınıfından eriştiği niteliklerin sayısının, erişilen tüm niteliklerin (doğrudan ya da dolaylı) (kendi sınıfı ve başka sınıfa ait) sayısına oranı.

- **FDP** (*Foreign Data Providers*):

Erişilen yabancı niteliklerin ait olduğu farklı sınıf sayısı.
"Feature Envy" sorununda bu değer küçüktür. (Daha çok belli bir sınıfa erişiyor.) Bu da metodun yanlış sınıfta olduğuna işaret eder.
Eğer FDP değeri yüksekse bu metot bir denetçi (*controller*) ya da "brain method" olabilir.

"Feature Envy" belirlemek için kullanılan yöntem:



ATFD (*Access to Foreign Data*) Metot için tanımlı FEW = 2 - 5

LAA (*Locality of Attribute Accesses*)

FDP (*Foreign Data Providers*)

➢ **Data Class:**

Bu tür sınıflar diğer sınıfların çok eriştiği verilere sahiptirler ancak kendileri bu veriler üzerinde işlem yapmazlar.

İlgili verilerin ve fonksiyonların birlikte tasarlanmadığını (*encapsulation problem*) göstergesi olabilir.

Belirlemek için kullanılan metrikler:

- **WOC** (*Weight Of Class*):

Bir sınıfın ara yüzünde yer alan (*public*), sırf erişim amaçlı olmayan (*non-accessor*) metotların sınıftaki tüm açık (*public*) metotlara oranı.

Sınıfın belli bir hizmet vermesi için bu değerin 1.0'a yakın olması tercih edilir.

Bu değerin küçük olması (WOC<0.33) sınıfın veri sınıfı olduğu şüphesi uyandırır.

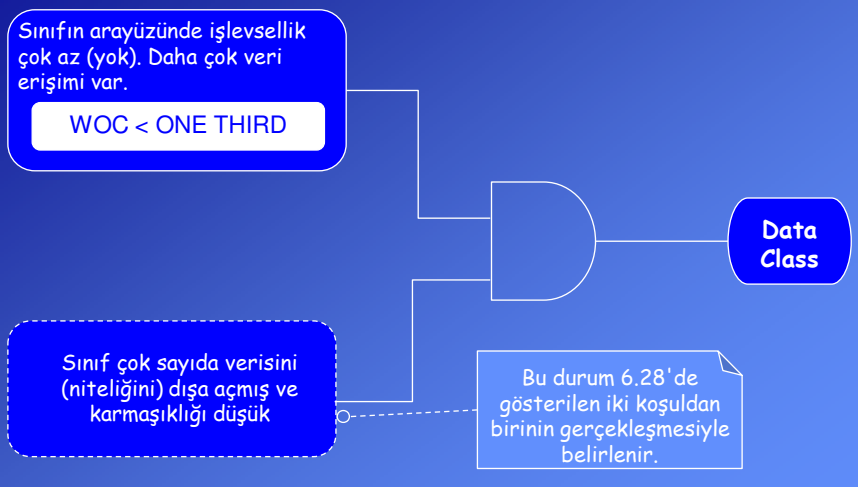
- **NOPA** (*Number Of Public Attributes*):

Verilerin (niteliklerin) açık (*public*) olması istenmeyen bir özelliktir.

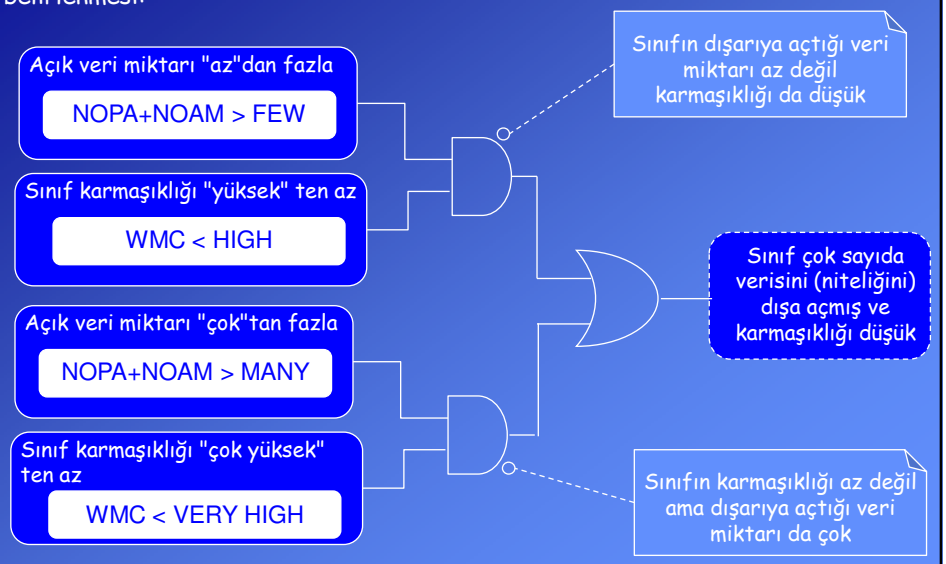
- **NOAM** (*Number Of Accessor Methods*):

Erişim (*accessor*) metotlarının sayısı

"Data Class" belirlemek için kullanılan yöntem:



"Sınıf çok sayıda verisini (niteliğini) dışa açmış ve karmaşıklığı düşük" koşulunun belirlenmesi:



➢ **Brain Method:**

Bir metodun sınıfa ait işlerin çoğunu yüklenmesiyle bu kusur ortaya çıkar.

Projenin başlangıcında metotlar sağlıklı olsa da ileriki aşamalarda yeni işlevler hep aynı metoda eklenirse metodun boyutu ve karmaşıklığı artar.

Sonuçta anlaşılması ve bakımı zor bir metot ortaya çıkar.

Belirlemek için kullanılan metrikler:

- **LOC** (*Line of Code*): Metot aşırı uzun mu?
- **CYCLO** (*Cyclomatic complexity*): McCabe dallanma sayısı
- **MAXNESTING** (*Maximum Nesting Level*)

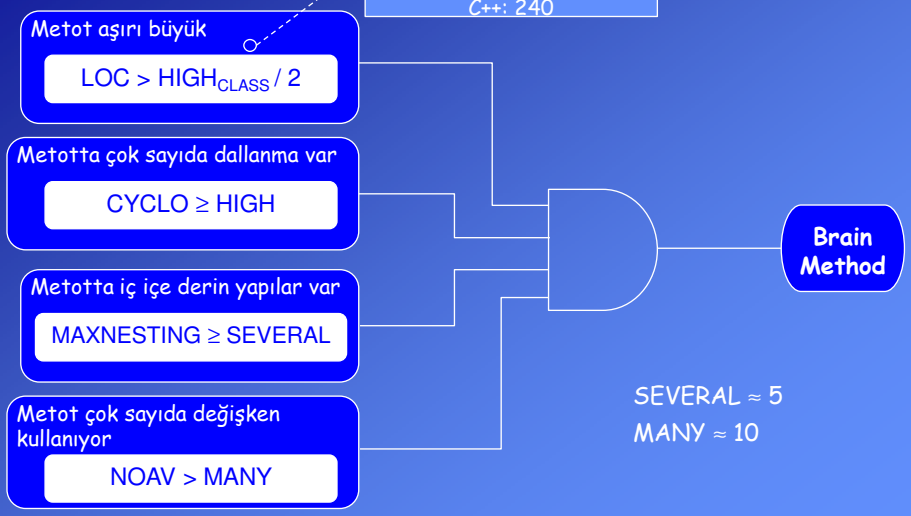
Metottaki iç içe yapıların en büyük derinliği.

CYCLO ve MAXNESTING metrikleri ile iç içe çok sayıda dallanma olup olmadığı sınanır (if-then-else, switch-case).
Bu yapıların fazlalığı çok şekillilik (*polymorphism*) özelliğinin kullanılmadığı anlamına gelebilir.
Bu durum metodun esnekliğini azaltır.

- **NOAV** (*Number of Accessed Variables*) Metotta erişilen toplam değişken, parametre, nitelik sayısı

"Brain Method" belirlemek için kullanılan yöntem:

$HIGH_{CLASS}$: LOC/Class için HIGH eşik değeri
Java:130
C++: 240



SEVERAL ≈ 5
MANY ≈ 10

İşbirliği Uyumsuzlukları (*Collaboration Disharmonies*)

Nesneye dayalı tasarımın sınıflar arası işbirliği konusundaki önerileri:

- Bir sınıf iyi tanımlanmış bir sorumluluğu olmalı, kendisi ile ilgili olmayan işleri başka sınıflardan hizmet alarak (işbirliği ile) (*delegation*) yerine getirmeli.
- İyi bir nesneye dayalı yazılımda sınıflar arası işbirliği metotlar üzerinden olmalı.
- Bir metot (sınıf) başka sınıflardan sınırlı sayıda hizmet almalı.
- Bir metot (sınıf) sınırlı sayıda yabancı sınıfa bağımlı olmalı.
- Bir metodun bağımlı olduğu (hizmet aldığı) diğer işlemlerin yeri aşağıdaki sıraya göre tercih edilir.
  (a) Aynı sınıf içinde, (b) aynı türetim zincirinde, (c) aynı pakette veya alt sistemde.
- Sınıflar (metotlar) arası bağımlılık zincirleme olarak çok yayılmamalı.

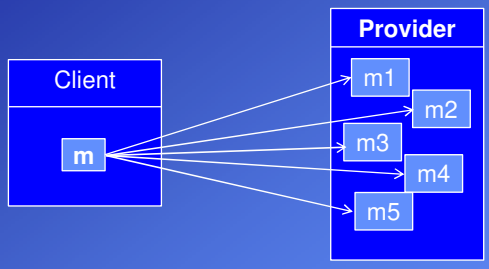
Kitapta tanımlanan uyumsuzluklar:

Intensive Coupling, Dispersed Coupling, Shotgun Surgery

M. Lanza, R.Marinescu, *Object-oriented metrics in practice : using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer, 2006.

➢ **Intensive Coupling** (Yoğun Bağımlılık):

Bir metot, çok sayıda başka metodu çağırmaktadır ve
bu yabancı metotlar aynı sınıfta (veya çok az sayıda sınıfta) toplanmıştır.



Kusurun belirlenmesi:

İki koşulun geçerliliği sınanacak.

1. Metot az sayıda başka sınıfa toplanmış çok sayıda metot çağırıyor mu?
2. Metodun karmaşıklığı "az" değil mi ("az"dan fazla mı)?
   İkinci koşulun nedeni bilerek yazılmış bazı zararsız metotları elemektir.
   Bu metotlar belli sınıfları uygun şekilde başlatmak ya da koşullamak için yazılmış olabilir.

"Intensive Coupling" belirlemek için kullanılan yöntem:

Metodun karmaşıklığı "az"dan fazladır.
İç içe blok sayısı "az"dan fazla
MAXNESTING > SHALLOW

MAXNESTING :
Metottaki iç içe blokların (if, döngü) en büyük sayısı. Karmaşıklığı ifade eder.
SHALLOW : 1-2

Intensive Coupling

Metot az sayıda sınıfa toplanmış çok sayıda metot çağırıyor.

Bu durum 6.34'te gösterilen iki koşuldan birinin gerçekleşmesiyle belirlenir.

Kullanılan metrikler:

CINT (*Coupling Intensity*):Bir metottan çağrılan farklı metotların sayısı

CDISP (*Coupling Dispersion*):
Bir metottan çağrılan diğer metotların ait olduğu sınıfların sayısı bölü CINT
Bağımlılığın farklı sınıflara nasıl yayıldığını gösterir.
Bu değerin küçük olması bağımlılığın daha çok aynı sınıfa olduğu anlamına gelir.

"Metot az sayıda sınıfa toplanmış çok sayıda metot çağırıyor" koşulunun belirlenmesi:

Çok sayıda başka metot çağırıyor
CINT > SHORT MEM CAP

Metotlar az sayıda sınıfa dağılmış
CDISP < HALF

Çağırılan metot sayısı 7'den fazla (Akılda tutulamayacak kadar çok).
Çağırılan metotlar 2-3 sınıfa dağılmış.
Ortalama olarak aynı sınıfa ait 2'den fazla metot çağırılıyor.

Metot az sayıda sınıfa toplanmış çok sayıda metot çağırıyor

Çağırılan metot sayısı "az"dan fazla
CINT > FEW

Metotlar çok az sayıda sınıfa dağılmış
CDISP < A QUARTER

Çağırılan metot sayısı 3-7 arası.
Çağırılan metotlar 1-2 sınıfta toplanmış.
Ortalama olarak aynı sınıftan 4 metot çağırılıyor.

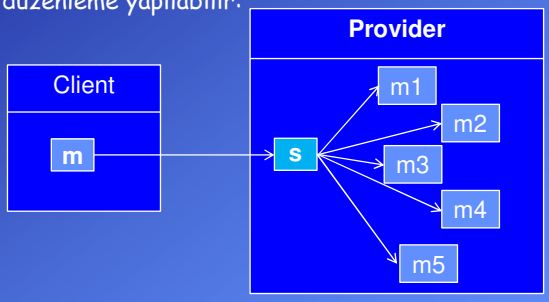
Olası düzenlemeler (*refactoring*):

A) "Intensive Coupling" sorunu bir metodun yanlış sınıfta olmasından kaynaklanabilir.

Bu durumda metodun yerini veya sınıfların sorumluluklarını değiştirmek gerekir.

B) Bazı durumlarda ise sorun hizmet sağlayan sınıfın yapısından kaynaklanır.

Bu durumda aşağıdaki düzenleme yapılabilir:



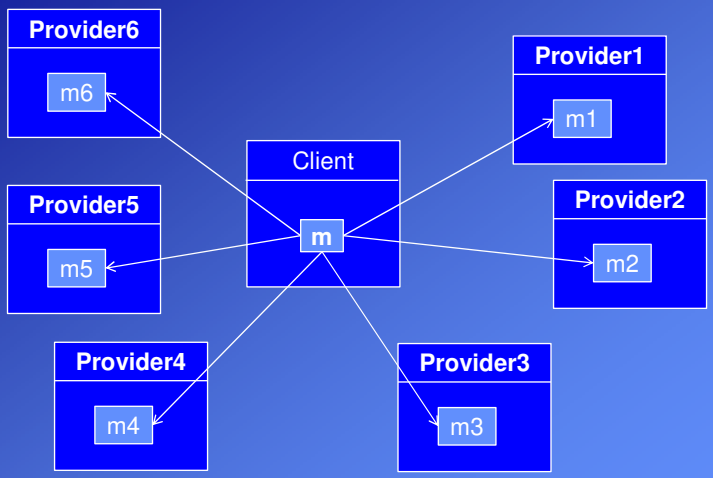
Burada s hizmet sağlayıcı (Provider) sınıfın hizmetleri için bir "kapı" olabilir.

Hizmet odaklı (*service-oriented*) yapılarda sınıfların (hizmetlerin) bu şekilde tasarlanması gerekir.

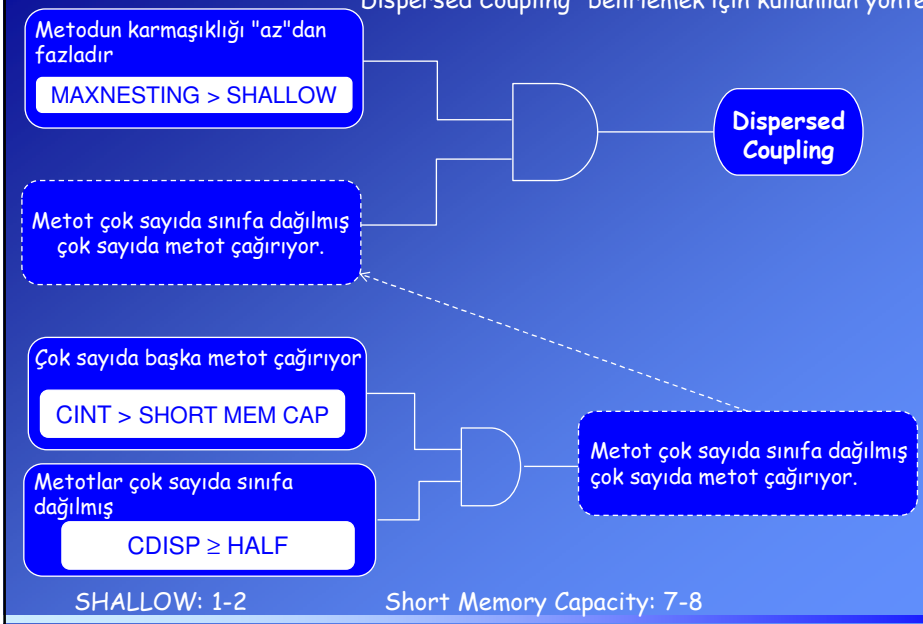
Diğer bir çözüm ise araya bir cephe sınıfı (*Facade*) koymak olabilir.

➢ **Dispersed Coupling** (Dağılmış Bağımlılık) :

Bir metot çok sayıda başka metodu çağırmaktadır ve bu yabancı metotlar çok sayıda farklı sınıfa dağılmışlardır.



"Dispersed Coupling" belirlemek için kullanılan yöntem:



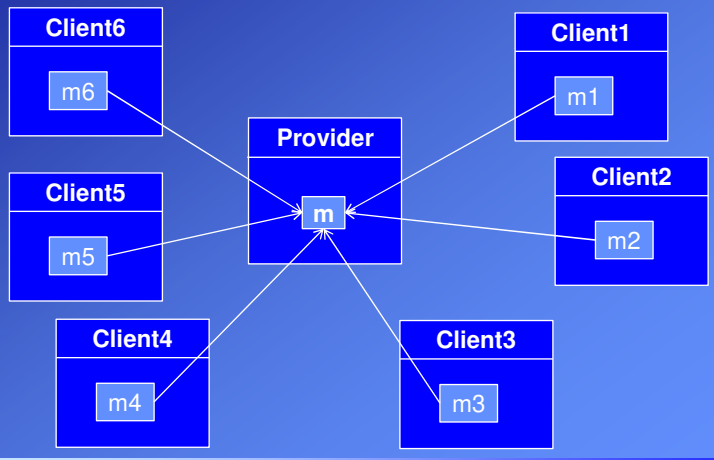
SHALLOW: 1-2 Short Memory Capacity: 7-8

- **Shotgun Surgery** (Çok sınıftan tek metoda bağımlılık):

Bir metoda çok sayıda farklı sınıfın metodundan çağrı geldiği durumda oluşur.

Aynı sınıfın farklı metotlarından da çağrı gelebilir.

Bu sorunun olduğu yazılımlarda bir metotta yapılacak bir değişiklik yazılımda birçok değişikliğe neden olabilir.



"Shotgun Surgery" belirlenmesi:

CM (*Changing Methods*) :
Bir metodu çağıran farklı metotların sayısı

CC (*Changing Classes*) :
Ölçülen metodu çağıran metotların ait olduğu farklı sınıfların sayısı

Sınıflandırma (Hiyerarşi) Uyumsuzlukları (*Classification Disharmonies*)

Nesneye dayalı tasarımda kalıtımın (türetimin) iki temel amacı vardır:

1. Daha genel yapılardan daha özel yapılar türetmek.
   Özel yapılar oluşturulurken genel yapıların ortak özellikleri tekrar kullanılmış (*reusability*) olur.
2. Aynı arayüze (*interface*) sahip sınıflar oluşturmak.
   Bu sınıflar ortak bazı sorumlulukları yerine getirirler ve birbirlerinin yerine geçebilirler (Örneğin GoF stratejileri).

Ancak türetimin yanlış kullanılması tasarım kusurlarına neden olabilir.

Özellikle tekrar kullanımı arttırmak için sadece kalıtımın (türetimin) kullanılması, sahip olma ilişkisinin göz ardı edilmesi sorunlara neden olur.

Uygun kalıtım (türetim) ile ilgili kurallar:

- Türetim ağaçlarının boyutları uygun olmalı (fazla geniş, fazla uzun değil).
  Türetim zincirlerinin hiç bulunmaması da sorun göstergesi olabilir.
  Çok geniş ağaçlar alt sınıfların kopyala yapıştır yöntemiyle yaratıldığının işareti olabilir.
  Çok derin (uzun) ağaçlar yazılımın anlaşılmasını ve bakımını zorlaştırır.
- Sınıflar türetim zinciri içinde kendilerinden önce ve sonra gelen sınıflar ile uyum içinde olmalıdır.
  Üst sınıflardan alınan ve yeniden tanımlanan üyelerin oranı uygun olmalı.
  Üst sınıfın üyelerinin büyük çoğunluğu reddedilmemeli.
- Üst (taban) sınıflar kendilerinden sonra gelen (türetilen) sınıflara bağlı olmamalı.
- Alt sınıf (türeyen), üst sınıfın metotlarını sadece çağırarak kullanmamalı.
  Bu tür kullanım "has-a" ilişkisi için daha uygun olur.
  Alt sınıf üst sınıfın metotlarını yeniden tanımlamalı, daha özel metotlar yaratmak için gerektiğinde çağırmalı.

➢ **Refused Parent Bequest** (Kalıtımın Reddi):

Türeyen sınıf üst sınıftan gelen kalıtımdan (ortak üyeler) yararlanmalıdır.

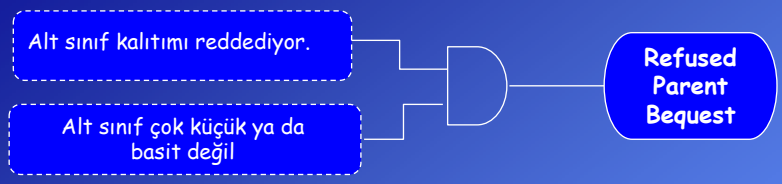
Kalıtım (*inheritance*) özelliğinin temel amaçlarından biri tekrar kullanımdır (*reusability*) diğeri ise ortak ara yüz oluşturmaktır.

Eğer alt sınıf üst sınıftan gelen özelliklerden yararlanmıyorsa türetim ağacının yapısında sorun olduğu şüphesi oluşur.

Kusurun belirlenmesinde varsayımlar:

1. İncelenen sınıfın üst (taban) sınıfı vardır.
2. Üst sınıf üçüncü parti bir sınıf değildir (örneğin bir arşiv sınıfı), veya bir ara yüz (*interface* "Java") değildir.

Kalıtım reddi uyumsuzluğunun belirlenmesi için iki koşulun oluşması gerekir:



Bir alt sınıfın üst sınıftan gelen kalıtımı kullanması üç şekilde olur:

- Üst sınıfın korunan (*protected*) niteliklerine (verilerine) erişir.
- Üst sınıfın korunan (*protected*) metotlarını çağırır.
- Üst sınıfın metotlarını örterek günceller (*overriding*).

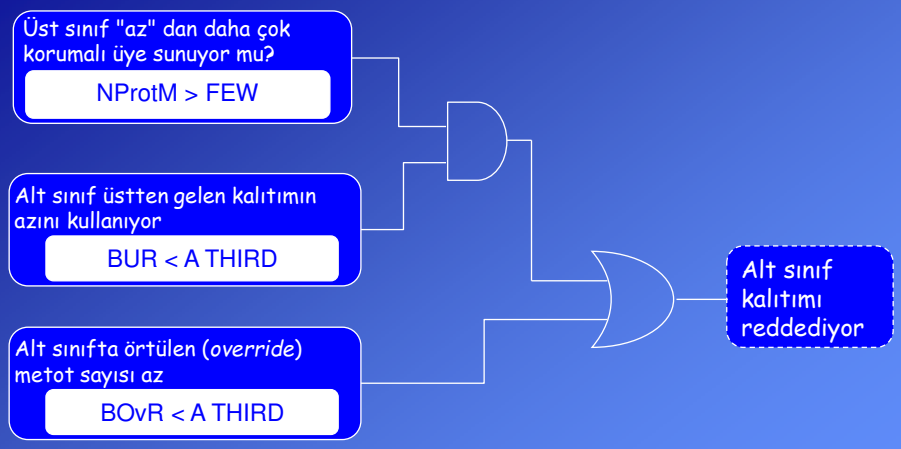
Bunu ölçmek için kullanılan metrikler:

BUR (Base class Usage Ratio) : Alt sınıfta erişilen korunan (*protected*) üyelerin oranı.

BOvR (Base-class Overriding Ratio): Örtülen ve güncellenen üst sınıf metot oranı

NPrM (Number of Protected Members): Üst sınıftaki korunan (*protected*) üye sayısı

1. Koşul: Üst sınıftan gelen kalıtım göz ardı ediliyor mu?

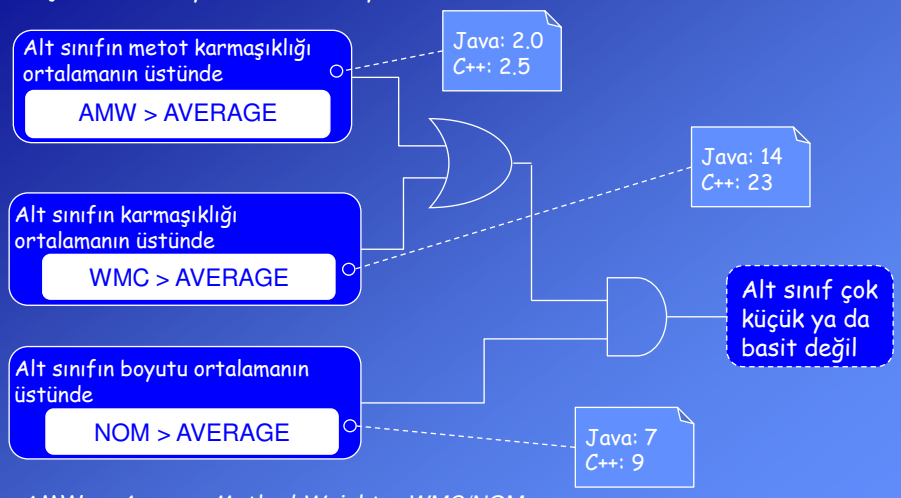


NProtM (Number of Protected Members): Üst sınıftaki korunan (*protected*) üye sayısı

BUR (Base class Usage Ratio) : Alt sınıfta erişilen korunan üyelerin oranı.

BOvR (Base-class Overriding Ratio): Örtülen ve güncellenen üst sınıf metot oranı

2. Koşul: Alt sınıf yeteri kadar büyük mü?



AMW: *Average Method Weight* *WMC/NOM*
WMC: *Weighted Method Count*
NOM: *Number Of Methods*

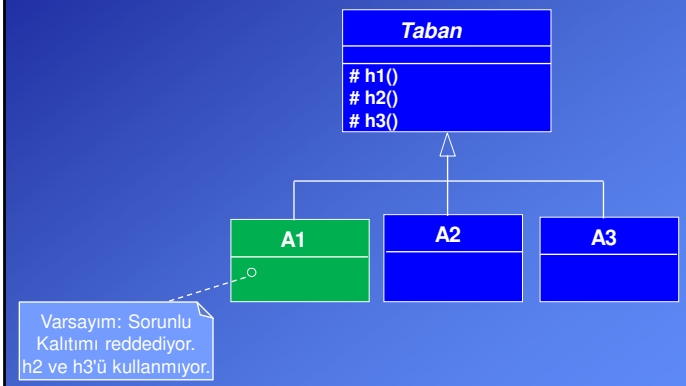
Kalıtım reddi uyumsuzluğunun giderilmesi:

Bu sorunun oluşmasının çeşitli nedenleri olabilir.

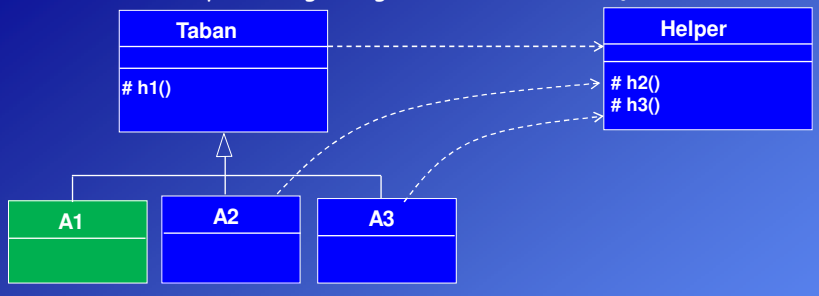
Eğer türetim zinciri hatalıysa yapı yeniden oluşturulmalıdır.

Diğer bir neden ise taban sınıfın çok sayıda alt sınıfı olması ve taban sınıftaki bazı özelliklerin bazı sınıfları çok ilgilendirmemesi.

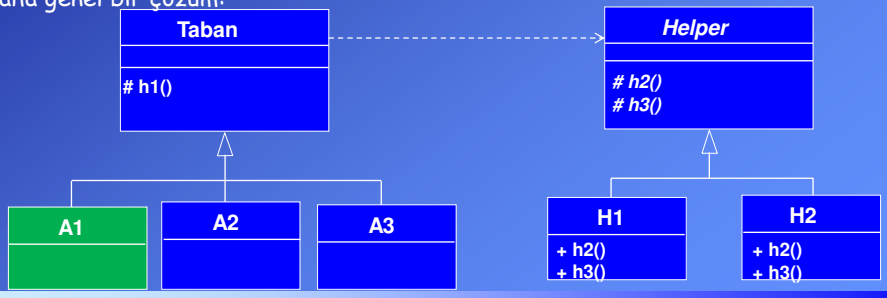
Bu durumda GoF strateji (veya köprü) kalıbı kullanılarak sorun çözülebilir.



Kalıtım reddi uyumsuzluğunun giderilmesi: "has-a" ilişkisinin kullanılması



Daha genel bir çözüm:



- **Tradition Breaker** (Geleneğin Sürdürülmemesi):

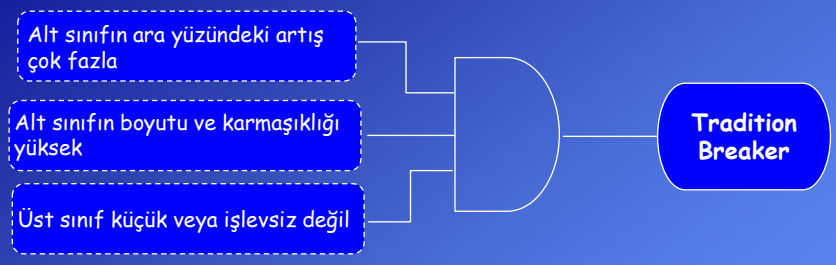
Normalde bir sınıfın ara yüzü (dışarıya verdiği hizmetler) evrimsel olarak alt sınıflarda güncellenir ve genişletilir.

Alt sınıf üst sınıfın "geleneğini sürdürmeli", bir anda yukarıdan aldığı tüm hizmetleri değiştirip tamamen farklı hizmetler sunmamalı.

Kusurun belirlenmesinde varsayımlar:

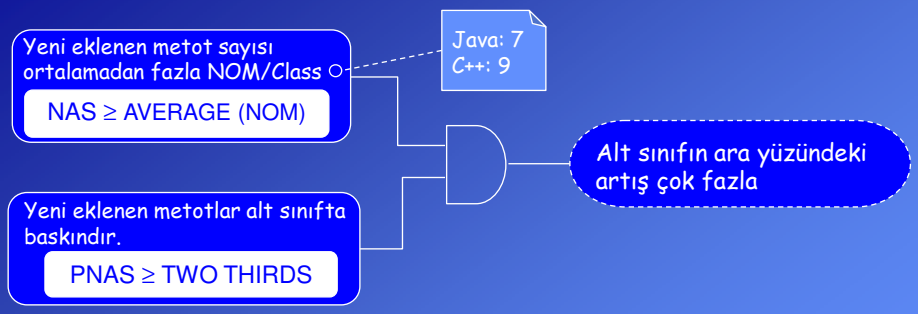
1. İncelenen sınıfın üst (taban) sınıfı vardır.
2. Üst sınıf üçüncü parti bir sınıf değildir (örneğin bir arşiv sınıfı), veya bir ara yüz (*interface* "Java") değildir.

Tradition Breaker uyumsuzluğunun belirlenmesi için üç koşulun oluşması gerekir:



1. Alt sınıfın ara yüzü, üst sınıfla karşılaştırıldığında çok büyümüştür.
2. Alt sınıf tek başına da büyük ve karmaşıktır.
3. Üst sınıfta yeteri kadar işlev vardır. Aksi durumda bir "gelenekten" söz edilemez.

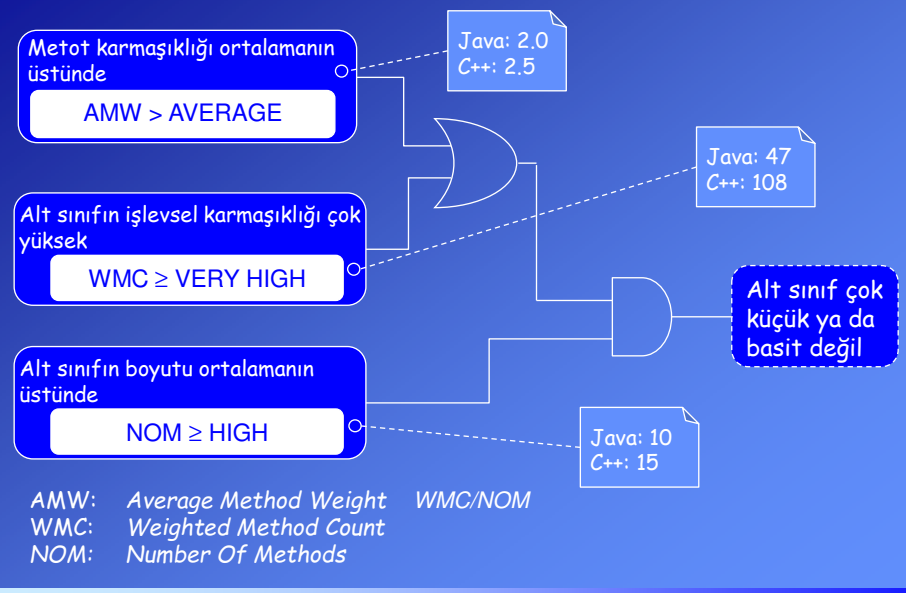
1. Koşul: Alt sınıfın ara yüzündeki artış çok fazla



NAS (*Number of Added Services*) : Alt sınıfta yeni eklenen "public" metotların sayısı. Bunlar üst sınıfta olmayan metotlardır.

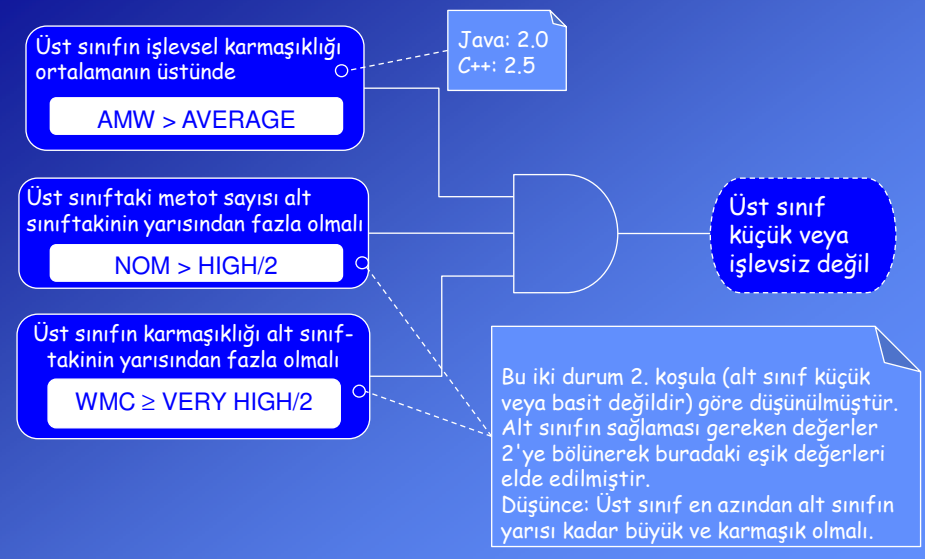
PNAS (*Percentage of Newly Added Services*): Alt sınıfta eklenen "public" metotların sayısının toplam "public" metotların sayısına oranı

2. Koşul: Alt sınıf yeteri kadar büyük ve karmaşıktır



AMW: *Average Method Weight* *WMC/NOM*
WMC: *Weighted Method Count*
NOM: *Number Of Methods*

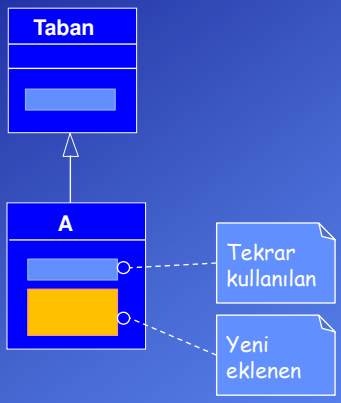
3. Koşul: Üst sınıf küçük veya işlevsiz değil



Geleneğin sürdürülmemesi sorununun giderilmesi:

Bu sorunun oluşmasının da temel nedeni türetim zincirinin hatalı kurulması olabilir.
Çözüm yöntemlerinden biri gerekli yerlerde "has-a" ilişkisinin kullanılmasıdır.

Sorun:



Düzenleme: