11. Ders: JAVA ile Nesne Yönelimli Programlama
Class Design Guidelines (Sınıf Tasarım Kılavuzları)

Fırat Üniversitesi Teknoloji Fakültesi Yazılım Mühendisliği Bölümü

YMH112 Algoritma ve Programlama-II

Dr. Öğr. Üyesi Yaman Akbulut

JAVA ile Nesne Yönelimli Programlama

- http://www.kriptarium.com/algoritma.html (Yardımcı kaynak)

- JAVA ile Nesne Yönelimli Programlama

  - Ders 29: Java Polimorfizmi / Yöntem Aşırı Yüklemesi (Method Overloading) (izle)
  - Ders 30: Java Polimorfizmi / Yöntem Geçersiz Kılma (Method Overriding) (izle)
  - Ders 31: super Anahtar Sözcüğünün Kullanımı (izle)
  - Ders 32: final Anahtar Sözcüğünün Kullanımı (izle)
  - Ders 33: Soyut Sınıf (Abstract Class) (izle)
  - Ders 34: Arayüz (Interface) (izle)

Java Keywords

abstract	double	int	super
assert	else	interface	switch
boolean	enum	long	synchronized
break	extends	native	this
byte	final	new	throw
case	finally	package	throws
catch	float	private	transient
char	for	protected	try
class	goto	public	void
const	if	return	volatile
continue	implements	short	while
default	import	static	
do	instanceof	strictfp	

Kaynak: Introduction to JAVA Programming 10th Edition, Y. Daniel Liang

1. Cohesion (Uyum)

Bir sınıf tek bir varlığı tanımlamalı ve tüm sınıf işlemleri tutarlı bir amacı desteklemek için mantıksal olarak birbirine uymalıdır.

Örneğin, öğrenciler için bir sınıf kullanabilirsiniz,

ancak öğrenciler ve personel farklı varlıklar olduğundan, öğrencileri ve personeli aynı sınıfta birleştirmemelisiniz.

Birçok sorumluluğu olan tek bir varlık, sorumlulukları ayırmak için birkaç sınıfa ayrılabilir.

1. Cohesion (Uyum)

String, StringBuilder ve StringBuffer sınıflarının tümü, örneğin dizelerle (string) ilgilenir, ancak farklı sorumluluklara sahiptir.

String sınıfı değişmez dizelerle (string) ilgilenir,

StringBuilder sınıfı değiştirilebilir dizeler (string) oluşturmak içindir

ve StringBuffer sınıfı, StringBuffer'ın dizeleri (string) güncellemek için senkronize metotlar içermesi dışında StringBuilder'a benzer.

2. Consistency (Tutarlılık)

Standart Java programlama stilini ve adlandırma kurallarını izleyiniz.

Sınıflar, veri alanları ve metotlar için bilgilendirici adlar seçiniz.

Popüler bir stil, veri bildirimini yapıcıdan önce ve yapıcıları metotlardan önce yerleştirmektir.

İsimleri tutarlı hale getirin. Benzer işlemler için farklı isimler seçmek iyi bir uygulama değildir.

Örneğin, length() metodu bir String, StringBuilder ve StringBuffer'ın boyutunu döndürür. Bu sınıflarda bu metot için farklı isimler kullanılsaydı tutarsızlık olurdu.

2. Consistency (Tutarlılık)

Genel olarak, varsayılan bir örnek oluşturmak için tutarlı bir şekilde genel argümanı olmayan (parametresiz) bir yapıcı sağlamalısınız.

Bir sınıf bağımsız argümanı olmayan (parametresiz) bir yapıcıyı desteklemiyorsa, nedenini belgeleyiniz.

Açık bir şekilde hiçbir yapıcı tanımlanmadıysa, boş bir gövdeye sahip genel bir varsayılan argümanı olmayan (parametresiz) yapıcı varsayılır.

Kullanıcıların bir sınıf için nesne oluşturmasını önlemek istiyorsanız, Math sınıfında olduğu gibi sınıfta özel bir yapıcı bildirebilirsiniz.

3. Encapsulation (Kapsülleme)

Bir sınıf, verilerini istemciler tarafından doğrudan erişimden gizlemek için private değiştiriciyi kullanmalıdır.

Bu, sınıfın bakımını kolaylaştırır.

Yalnızca veri alanının okunabilir olmasını istiyorsanız alıcı (get) metodu sağlayın ve yalnızca veri alanının güncellenebilir olmasını istiyorsanız bir ayarlayıcı (set) metodu sağlayınız.

Örneğin, Rational sınıfı, pay ve payda için bir alıcı (get) metodu sağlar, ancak bir Rational nesnesi değişmez olduğu için ayarlayıcı (set) bir metot sağlamaz.

4. Clarity (Açıklık, Netlik)

Uyum, tutarlılık ve kapsülleme, tasarım netliği elde etmek için iyi kılavuzlardır.

Ek olarak, bir sınıfın açıklanması ve anlaşılması kolay açık bir sözleşmesi olmalıdır.

Kullanıcılar sınıfları birçok farklı kombinasyonda, düzende ve ortamda birleştirebilir.

Bu nedenle, kullanıcının onu nasıl ve ne zaman kullanabileceği konusunda hiçbir kısıtlama getirmeyen bir sınıf tasarlamalısınız.

4. Clarity (Açıklık, Netlik)

Özellikleri, kullanıcının herhangi bir sırayla ve oluşum sıralarından bağımsız olarak işlev gören herhangi bir değer ve tasarım yöntemi kombinasyonu ile ayarlamasına izin verecek şekilde tasarlayınız.

Metotlar, kafa karışıklığına neden olmadan sezgisel olarak tanımlanmalıdır.

Örneğin, String sınıfındaki substring (int beginIndex, int endIndex) yöntemi biraz kafa karıştırıcıdır.

Metot, endIndex yerine beginIndex'ten endIndex – 1'e bir alt dize döndürür.

beginIndex'ten endIndex'e bir alt dizeyi döndürmek daha sezgisel olacaktır.

4. Clarity (Açıklık, Netlik)

Diğer veri alanlarından türetilebilecek bir veri alanı beyan etmemelisiniz.

Örneğin, aşağıdaki Kisi sınıfının iki veri alanı vardır: dogumTarihi ve yas.

yas, dogumTarihinden türetilebildiğinden, yas veri alanı olarak beyan edilmemelidir.

```
public class Kisi {
    private java.util.Date dogumTarihi;
    private int yas;
    ...
}
```

5. Completeness (Bütünlük, Tamlık)

Sınıflar, birçok farklı müşteri tarafından kullanılmak üzere tasarlanmıştır.

Çok çeşitli uygulamalarda yararlı olabilmesi için, bir sınıf, özellikler ve metotlar aracılığıyla özelleştirme için çeşitli yollar sağlamalıdır.

Örneğin, String sınıfı, çeşitli uygulamalar için yararlı olan 40'tan fazla metot içerir.

6. Instance vs. Static (Örnek vs. Statik)

Sınıfın belirli bir örneğine (instance, nesne) bağımlı olan bir değişken veya metot, bir örnek (instance, nesne) değişkeni veya metot olmalıdır.

Bir sınıfın tüm örnekleri tarafından paylaşılan bir değişken static olarak bildirilmelidir.

Sayici sınıfı içindeki sayac değişkeni,
Sayici sınıfının tüm nesneleri tarafından paylaşılır ve bu nedenle static olarak bildirilir.

```
1  public class Sayici {
2      static int sayac;
3
4      Sayici(){
5          sayac++;
6          System.out.println(sayac);
7      }
8  }
```

6. Instance vs. Static (Örnek vs. Statik)

Belirli bir örneğe(instance, nesne) bağlı olmayan bir metot, static bir metot olarak tanımlanmalıdır.

Hesapla'daki kupAl() ve kareAl() metotları belirli bir örneğe bağlı değildir

ve bu nedenle static bir metotlar olarak tanımlanır.

```
1  public class Hesapla{
2      static int kupAl(int a){
3          return a*a*a;
4      }
5      static int kareAl(int a){
6          return a*a;
7      }
8  }
```

Okunabilirliği artırmak ve hataları önlemek için her zaman bir sınıf adından (bir referans değişkeni yerine) statik değişkenlere ve metotlara başvurunuz.

6. Instance vs. Static (Örnek vs. Statik)

Statik bir veri alanını başlatmak için yapıcıdan parametre iletmeyiniz.

Statik veri alanını değiştirmek için bir ayarlayıcı (set) metodu kullanmak daha iyidir.
Bu nedenle, aşağıda (a)'daki belirtilen sınıfın (b) ile değiştirilmesi daha iyidir.

```
public class SomeThing {
  private int t1;
  private static int t2;

  public SomeThing(int t1, int t2) {
    ...
  }
}
```

(a)

```
public class SomeThing {
  private int t1;
  private static int t2;

  public SomeThing(int t1) {
    ...
  }

  public static void setT2(int t2) {
    SomeThing.t2 = t2;
  }
}
```

(b)

6. Instance vs. Static (Örnek vs. Statik)

Örnek ve statik, nesne yönelimli programlamanın ayrılmaz parçalarıdır.

Bir veri alanı veya metot ya örnektir veya statiktir.

Statik veri alanlarını veya metotlarını yanlışlıkla gözden kaçırmayınız.

Statik olması gereken metodu bir örnek metodu olarak tanımlamak yaygın bir tasarım hatasıdır.

Örneğin, n'nin faktöriyelini hesaplamak için faktoriyel(int n) metodu, herhangi bir özel durumdan bağımsız olduğu için statik olarak tanımlanmalıdır.

6. Instance vs. Static (Örnek vs. Statik)

Bir yapıcı her zaman örnektir (instance),

çünkü belirli bir örnek (instance) oluşturmak için kullanılır.

Statik bir değişken veya metot, bir örnek metottan çağrılabilir,

ancak bir örnek (instance) değişkeni veya metot, statik bir metottan

çağrılamaz.

7. Inheritance vs. Aggregation
(Kalıtım vs. Toplama)

Kalıtım ve toplama arasındaki fark, is-a ve has-a ilişkisi arasındaki farktır.

Örneğin, elma bir meyvedir; bu nedenle, Elma ve Meyve sınıfları arasındaki ilişkiyi modellemek için kalıtımı kullanırız.

Bir kişinin bir adı vardır; bu nedenle, Kisi ve Ad sınıfları arasındaki ilişkiyi modellemek için toplamayı kullanırız.

8. Interfaces vs. Abstract Classes
(Arayüzler vs. Soyut Sınıflar)

Hem arayüzler hem de soyut sınıflar, nesneler için ortak davranışı belirtmek için kullanılabilir.

Arayüz mü yoksa sınıf mı kullanacağımıza nasıl karar veriyoruz?

Genel olarak, ebeveyn-çocuk (parent-child) ilişkisini açıkça tanımlayan güçlü bir (is-a) ilişki, sınıflar kullanılarak modellenmelidir.

Örneğin, portakal bir meyve olduğu için, ilişkileri sınıf kalıtımı kullanılarak modellenmelidir.

8. Interfaces vs. Abstract Classes
(Arayüzler vs. Soyut Sınıflar)

Zayıf bir (is-a) ilişki, aynı zamanda bir tür (is-kind-of) ilişki olarak da bilinir, bir nesnenin belirli bir özelliğe sahip olduğunu gösterir.

Zayıf bir is-a ilişkisi, arayüzler kullanılarak modellenebilir.

Örneğin, tüm dizeler (string) karşılaştırılabilir, bu nedenle String sınıfı Comparable arayüzünü uygular (implements).

Cember veya dikdörtgen geometrik bir nesnedir, bu nedenle Cember, GeometrikSekil'in bir alt sınıfı olarak tasarlanabilir.

Cemberler, farklıdır ve yarıçaplarına göre karşılaştırılabilirdir, bu nedenle Cember, Comparable arayüzü uygulayabilir (implements).

8. Interfaces vs. Abstract Classes
(Arayüzler vs. Soyut Sınıflar)

Arayüzler, soyut sınıflardan daha esnektir, çünkü bir alt sınıf yalnızca bir üst sınıfı genişletebilir, ancak herhangi bir sayıda arayüz uygulayabilir.

Ancak arayüzler somut yöntemler içeremez.

Arayüzlerin ve soyut sınıfların erdemleri, onu uygulayan soyut bir sınıfla bir arayüz oluşturularak birleştirilebilir.

Ardından, hangisi uygunsa, arayüzü veya soyut sınıfı kullanabilirsiniz.