Liskov Substitution Principle (LSP)
Liskov'un Yerine Gecme Prensibi

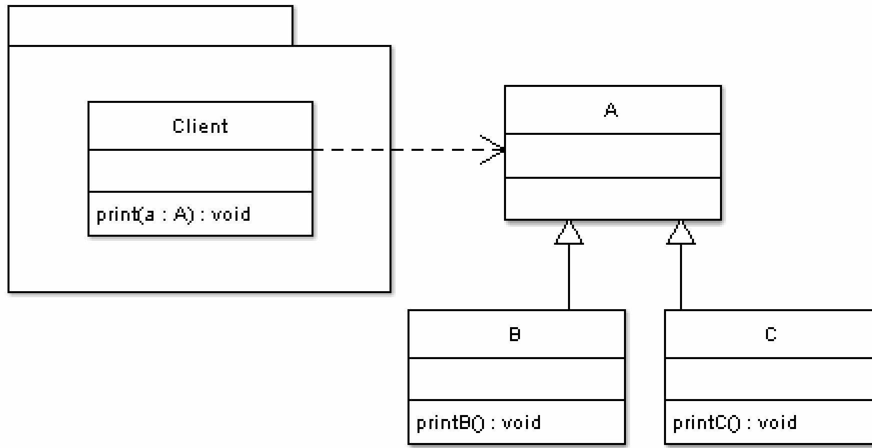
KurumsalJava.com

Özcan Acar
Bilgisayar Mühendisi
http://www.ozcanacar.com

Barbara Liskov [1] tarafından geliştirilen bu prensip kısaca şöyle açıklanabilir:

"Alt sınıflardan oluşturulan nesneler üst sınıfların nesneleriyle yer değiştirdiklerinde aynı davranışı göstermek zorundadırlar."

LSP'ye göre herhangi bir sınıf kullanıcısı, bu sınıfın alt sınıfları kullanmak için özel bir efor sarf etmek zorunda kalmamalıdır. Onun bakış açısından üst sınıf ve alt sınıf arasında farklılık yoktur. Üst sınıf nesnelerinin kullanıldığı metotlar içinde alt sınıftan olan nesneler aynı davranışı sergilemek zorundadır, çünkü oluşturulan metotlar üst sınıf davranışları örnek alınarak programlanmıştır. Alt sınıflarda meydana gelen davranış değişiklikleri, bu metotların hatalı çalışmasına sebep verebilir. Özellikte bu metotlarda *instanceof* gibi nesnelerin tipleri arasında kıyaslama yapılmak zorunda kalındığı zaman, LSP prensibi çiğnenmiş olur ki, bu alt sınıfların varlığından haberdar olunduğu anlamına gelir. Kullanıcı sınıflar ideal durumda alt sınıfların varlığından haberdar bile olmamalıdır.



Resim 1 Client sınıfında bulunan print metodu A tipinde nesneler üzerinde işlem yapmaktadır

Resim 1 de yer alan Client sınıfındaki *print()* metodunun nasıl LSP prensibine ters düştüğünü yakından inceleyelim. Bu metot A sınıfından olan nesneler üzerinde işlem yapmaktadır. A sınıfı B ve C sınıfları tarafından genişletilmiştir, yani A sınıfından olan bir parametre nesnesi aynı zamanda B ve C sınıfında da olabilir.

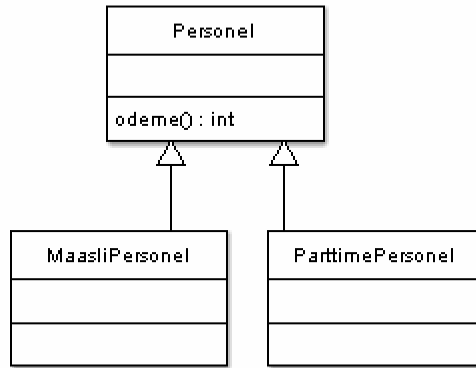
Kod 1 Client.java

```
package shop;

public class Client
{
	public void print(A a)
	{
		if(a instanceof B)
		{
			((B)a).printB();
		}
		else if(a instanceof C)
```

[1] Bakınız: http://www.pmg.csail.mit.edu/~liskov/

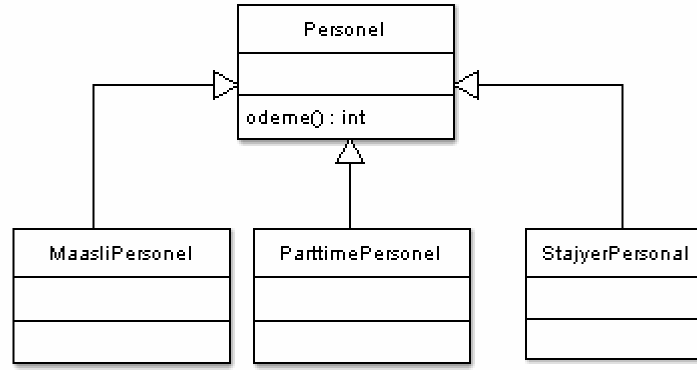
```
            {
                  ((C)a).printC();
            }
      }

      public static void main(String[] args)
      {
            Client client = new Client();
            B b = new B();
            C c = new C();

            client.print(b);
            client.print(c);
      }
}
```

Kod 1 de yer alan *print()* metodu LSP ile uyumlu değildir. Bu metodun A sınıfına bağımlılığı vardır ve bu sınıfın nesneleri üzerinde işlem yapacak şekilde implemente edilmiş olması gerekir. Lakin *print()* metodu *instanceof* komutuyla parametre olarak verilen nesnenin hangi tipte olduğunu tespit etmeye çalışmakta ve tipe bağımlı olarak nesne üzerinde işlemi gerçekleştirmektedir. Bu durum hem OCP'ye (Open Closed Principle) hemde LSP'ye ters düşmektedir. OCP uyumlu değildir, çünkü *print()* metodu A'nin her yeni alt sınıfı için değişikliğe uğramak zorundadır. LSP'ye uyumlu değildir, çünkü B ya da C sınıfından olan nesneler A sınıfından olan bir nesnenin yerini alamadığı için *print()* metodu *instanceof* ile nesnenin tipini tespit etmek zorunda bırakılmaktadır. Buradan genel bir sonuç çıkartabiliriz: LSP'ye ters düşen bir implementasyon aynı zamanda OCP'ye de ters düşer.



Resim 2 Maaşlı ve parttime çalışan işçi modeli

LSP'nin nasıl uygulanabileceğini diğer bir örnekte görelim. Resim 2 de bir firma çalışanları Personel sınıfını genişleten MaasliPersonel ve ParttimePersonel sınıfları ile temsil edilmektedirler. Personel sınıfı soyuttur (abstract). Çalışanların maaşlarının ödenebilmesi için Personel sınıfında bulunan soyut *odeme()* metodunun alt sınıflarca implemente edilmesi gerekmektedir. Firmaya yeni stajyerler alındığı için, modelin resim 3 deki gibi adapte edildiğini farz edelim.



Resim 3 Maaşlı, parttime ve stajyer işçi modeli

Staj yapan elemanlar için maaş ödenmez, bu yüzden *odeme()* metodu StajyerPersonel sınıfında boş implemente edilmesi gerekir. İmplementasyon şu şekilde olabilir:

Kod 2 StajyerPersonel.java - odeme() metodu

```
public int odeme()
{
      return 0;
}
```

Sıfır değerine geri vererek, *odeme()* metodunun geçerli ve kullanılabilir bir metot olduğunu ifade etmiş oluyoruz. Bu yanlıştır! Bu metodun StajyerPersonel sınıfında implemente edilmemesi gerekir, çünkü staj yapanlara maaş ödenmez.

Bu sorunu kod 3 de yer aldığı gibi bir Exception oluşturarak çözebiliriz. Eğer bir stajyer için *odeme()* metodu kullanılacak olursa, oluşan PersonelException, bir stajyer için maaş ödenemeyeceğini metot kullanıcısına bildirir. Exception nesneleri olağan olmayan durumları ifade etmek için kullanılır.

Kod 3 StajyerPersonel.java - odeme() metodu

```
public int odeme()throws PersonelException
{
      throw new PersonelException("Stajyer maasli calismaz!");
}
```

Bu implementasyon ilk örnekte olduğu gibi LSP ile uyumlu değildir. Neden uyumlu olmadığını inceleyelim. Kod 4 de çalışanlara ödenen toplam maaş miktarı hesaplanmaktadır.

Kod 4 Toplam maaş miktarı hesaplanıyor

```
List<Personel> personel = getPersonelList();
int total = 0;
```

```java
for(int i=0; i < personel.size(); i++)
{
	total+=personel.get(i).odeme();
}
```

StajyerPersonal sınıfının sisteme eklenmesiyle kod 4 de yer alan kodun değiştirilmesi gerekmektedir. Ya try/catch bloğu kullanılarak oluşabilecek bir PersonelException'in işleme tabi tutulması gerekir ya da metot signatüründe throws kullanılarak PersonelException'in bir üst katmana iletilmesi gerekir. StajyerPersonel ismini taşıyan yeni bir sınıf oluştuğu için, mevcut Personel sınıfı kullanıcıları (client) yapısal değişikliğe uğramıştır.

Kod 5 try/catch...

```java
try
{
	List<Personel> personel = getPersonelList();
	int total = 0;
	for(int i=0; i < personel.size(); i++)
	{
		total+=personel.get(i).odeme();
	}
}
catch (Exception e)
{
	// handle exception
}
```

Kod 6 instanceof

```java
List<Personel> personel = getPersonelList();
int total = 0;
for(int i=0; i < personel.size(); i++)
{
	if(! (personel.get(i) instanceof StajyerPersonal))
	{
		total+=personel.get(i).odeme();
	}
}
```

Try/catch blokları komplike yapılardır ve kodun okunulabilirliğini azaltırlar. Try/catch bloğundan kurtulmak için kod 6 de yer alan implementasyon düşünülebilir. Burada *instanceof* ile sırada hangi tip bir nesnenin olduğunu tespit edebilir ve StajyerPersonel tipi nesneleri işlem dışı bırakabiliriz. Daha öncede gördüğümüz gibi bu implementasyon LSP ile uyumlu değildir, çünkü üst sınıf ile çalışan bir metot *instanceof* ile alt sınıfları tanımak zorunda bırakılmaktadır. Bunun tek sebebi StajyerPersonel sınıfından olan bir nesnenin, Personel sınıfından bir nesne ile yer değiştiremez durumda olmasıdır. Personel sınıfını kullanan diğer sınıflar alt sınıf olan StajyerPersonel sınıfı sisteme eklendikten sonra bu değişiklikten etkilenmiştir. LSP'ye göre bu olmaması gereken bir durumdur. Alt sınıfların nesneleri, üst sınıflardan olan nesnelerin kullanıldığı metotlar içinde üst sınıf nesneleriyle aynı davranışı göstermek zorundadırlar, aksi taktirde kullanıcı sınıflar bu durumdan etkilenirler.

Sorun staj yapan bir elemanın Personel sınıf hiyerarşisinde yer alan bir sınıf aracılığıyla modellenmiş olmasında yatmaktadır. Staj yapan bir eleman maaş almadığı için personele ait değildir, bu yüzden StajyerPersonel sınıfının Personel sınıfını genişletmesi doğru değildir. Sorunu çözmek ve LSP konform hale gelmek için StajyerPersonel sınıfının Personel sınıf hiyerarşisinden ayrılması gerekmektedir.

EOF (End Of Fun)
Özcan Acar