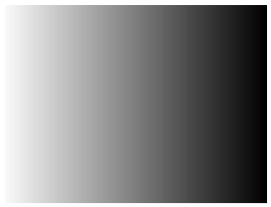


Java Interface ve Abstract Sınıflar

Özcan Acar
Bilgisayar Mühendisi

http://www.KurumsalJava.com
http://www.XPTurk.org

Bu makale Özcan Acar tarafından yazılmış olan Java Tasarım Şablonları ve Yazılım Mimarileri isimli kitaptan alıntıdır.

Detaylı bilgiyi http://www.pusula.com adresinden edinebilirsiniz.

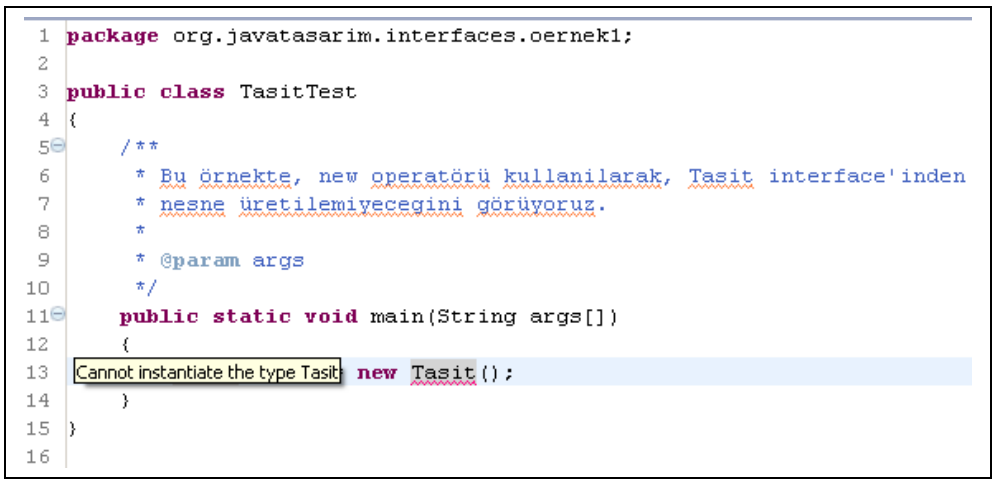
Interface Nedir?

Java dilinde interface adını taşıyan, tasarım şablonlarında ve modellemede kullanılan sınıflar tanımlamak mümkündür. Bir interface normal bir Java sınıfından farksız bir şekilde tanımlanır. Sınıf tanımlanırken class yerine *interface* terimi kullanılır.

Nesneye yönelik programlama dillerinde (örneğin C++) tamamen soyut (abstract) sınıflar tanımlanarak oluşturulabilecek interface benzeri sınıflar için Java dilinde *interface* ve *implements* gibi özel kelimeler (keywords) mevcuttur. Bu Java diline esnek yazılımlar yapılabilmesi adına büyük zenginlik katmaktadır. Bunun nasıl yapıldığını görmeden önce, bir interface sınıfın ne olduğunu görelim.

```
package org.javatasarim.interfaces.oernek1;

public interface Tasit
{
      public String getMarka();

}
```

Java'da tanımlanmış bir interface sınıftan, normal bir sınıfta olduğu gibi new() operatörü ile bir nesne **oluşturulamaz**!



Resim 1

Resimde görüldüğü gibi, Java compiler "**Cannot instantiate the type Tasit**" hata mesajı ile, Taşıt isminde bir nesne oluşturulamıyacağına işaret ediyor. Bu neden mümkün degildir?

Bunu gerçek hayattan bir örnek vererek açıklamaya çalışalım.

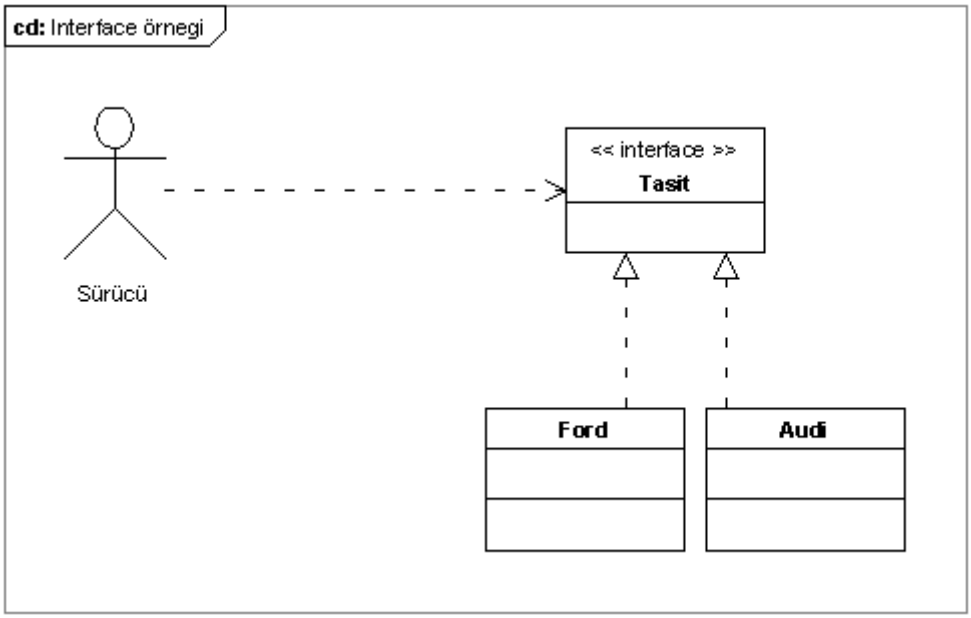
Çalıştığım firma tarafından bana, mesai saatlarinde kullanılmak üzere bir araç tahsis edildi. Bu aracı kullanarak, müşteri görüşmelerine gidiyor ve firmanın diğer işlerini takip ediyorum. Araç bana verilmeden önce, ehliyetimin olduğunu firmaya bildirdim. Şimdi geriye dönelim ve seneler önce sürücü kursunda aracı kullanmak için neler öğrendiğimi gözden geçirelim.

Sürücü kursunda ilk ögrendiğim bir aracı kullanmak için fren, debriyaj, vites vb. Parçaların mevcut olduğuydu. Aracı hareket ettirmek için vitesi 1 e takıp, hafif gaz vererek, yavaş yavaş sol ayağımı debriyajdan çekmem gerektiğini öğrendim. Araç hereket ettikten ve belirli bir hıza ulaştıktan sonra, ayağımı gazdan çekip, sol ayağımla debrijaya basarak, ikinci vitese geçilmesi gerektiğini öğrendim. Aracı durdurmak için frene basmam gerektiği gösterildi. Zaman içinde aracın nasıl hareket ettirilip, durdurulacağını iyice anladım ve sürücü sınavını kazanarak, ehliyetimi aldim.

Ehliyet sahibi olduğumda bildiğim birşey vardı, oda vitesli araçların hepsinin aynı şekilde kullanıldığı! Hangi araç olursa olsun, vitesi, freni, debriyaji, gaz pedalı olduğu sürece kullanabilirdim, çünkü her araç üreticisi mutlaka ve mutlaka, ürettiği her vitesli araçta bu parçaları benim bildiğim şekilde yerleştirecekti. Arabanın markasının benim için bir önemi yoktu, Ford'da olabilirdi, Fiat'da.

Firmam tarafından bana verilen ilk araç Ford marka bir binek oto idi. Hiç zorluk çekmeden bu aracı kullanarak, firmanın işlerini takip ettim. Kısa bir zaman sonra Audi A4 tipinde bir araca geçtim. Ford ve Audi arasında, kullanım açısından bir farklılık olmadığından, zorluk çekmeden Audi'yi kullanmaya başladım.

Bu örnekte dikkatimizi çeken bir nokta bulunuyor. Bunu daha sonra Java interface sınıfları ile ilişkilendirecegiz. Ben araç sürülmesi için gerekli işlemleri ve kullanılacak metodları biliyorum. Her tip vitesli binek otoyu kullanabilirim. Sürücü kursunda öğrendiklerim her vitesli binek oto icin geçerli. Buradan şu sonucu çikarıyoruz: binek oto kullananlar ve üreten firmalar arasında, bir nevi görünmez ama bilinen bir anlaşma yapılmış. Bu anlaşmaya göre, sürücü kursunda, ehliyet edinmek isteyenlere, aracın nasıl kullanılacağı öğretiliyor. Sürücü kursunda, sürücü adayına dolaylı verilen bir söz var: „Bu aracı kullanmayı öğrendiğin taktirde, diğer araçları da bu şekilde, başka birşey öğrenmene gerek kalmadan kullanabileceksin!"



Resim 2

Binek oto üreticileri, sürücülere verilen bu sözü bildiklerinden (bu bir nevi anlaşma), üretilen araçlar üzerinde durup, dururken bir değişiklik yapıp, örneğin vitesi yada fren pedalını bagaja koymuyorlar :) Bu böyle olsa idi, ehliyeti olan bir şahıs bu aracı nasıl kullanırdı? Kullanabilse, trafik emniyeti açısından bu iyi olur muydu? Araç yapısı değistirildiğinde bunun gibi sorulacak birçok soru var. Hiçbir binek oto üreten firma böyle bir değişiklik yapmayacağından, sürücüye verilen söz (bu interface sınıfının metodlarıdır) de hiçbir zaman bozulmamış oluyor.

Bu açıklamanın ardından tekrar başa dönelim ve neden bir interface sınıfından new operatörü ile bir nesne oluşturulamıyacağına göz atalım. Bir interface sınıfında sadece metotlar deklare edilir. Bu metotların gövdeleri boştur. Alt sınıflar bu metotların gövdeleri için gerekli kodu oluştururlar (implemente ederler). Metot gövdesi olmayan bir interface sınıftan nesne oluşturulamaz, çünkü sadece metot deklarasyonlarına sahip bir nesnenin hiçbir işlevsel görevi olamaz!

Bu örnekten sonra bir interface sınıfın nasıl kullanıldığına göz atalım. UML1 diagramında (resim 2) görüldüğü gibi sürücü sadece Tasit sınıfını tanıyor. Tasit interface olarak tanımlanmış.

```
package org.javatasarim.interfaces.oernek1;

/**
 * Tasit interface
 *
 * @author Oezcan Acar
 *
 */
public interface Tasit
{
      public String getMarka();
}
```

Ford ve Audi isminde iki sınıf bulunuyor. Bu sınıflar Tasit interface sınıfını implemente ediyorlar. Bu iki implementasyon sınıfı bünyelerinde getMarka() isminde bir metod oluşturup, bu metodu implemente etmek (metot gövdesi için gerekli kodu oluşturmak) zorundalar.

```
package org.javatasarim.interfaces.oernek1;

public class Audi implements Tasit
{

      public String getMarka()
      {
            return "Audi A4";
      }
}
```

Herhangi bir Java sınıfı **implements** direktifini kullanarak, bir interface sınıfının sahip olduğu metotları implemente edebilir. İmplementasyon ile, interface sınıfının tanımladığı (örneğin getMarka()) metot gövdelerinin, alt sınıfta (yukardaki örnekte Audi sınıfı) kodlanmasını

kastediyoruz. Buna göre interface sınıflarında sadece metodlar deklare edilebilir ve implemente edilemez. İmplementasyonu altsınıflar üstlenir.

```
package org.javatasarim.interfaces.oernek1;

/**
 * TasitTest2 sinifi
 *
 * @author Oezcan Acar
 *
 */
public class TasitTest2
{
	/**
	 * Bu örnekte, new operatörü kullanilarak, bir Audi marka
	 * tasit ürettik. Tasit interface'ini kullanan main metodu
	 * Tasit veritipinde bir nesne tanimliyor. Bu nesne
	 * Tasit interface'ini implemente eden herhanigi bir sinif
	 * olabilir.
	 *
	 * @param args
	 */
	public static void main(String args[])
	{
		Tasit tasit = new Audi();
		System.out.println(tasit.getMarka());
	}
}
```

TasitTest2 bünyesinde Tasit.getMarka() metodu kullanılıyor. Bu metot, Audi ve Ford sınıflarında implemente edildiği için , new Audi() yada new Ford() ile TastTest2 sınıfına, Tasit interface sınıfının verdiği sözü tutan bir sınıftan elde edilmiş bir nesneyi verebiliriz. Burada „verdiği sözü tutan“ kelimelerinin altını çizmek istiyorum. Bir interface aslında dış dünyaya verilen bir söz metnidir. Bir interface sınıf bünyesinde metotlar tanımlar. Interface sınıfı kullanan diğer sınıflar ise bu metodları kullanarak, gerekli işlemleri yaparlar. İşlemlerin nasıl gerçekleştirildiği, bunun için hangi implementasyon sınıflarının kullanıldığı, kullanıcı için önemli değildir. Analog bir şekilde sürücü kursu ve firma aracı örneğine dönecek olursak, interface ve bu örnek arasındaki paralellikleri görmüş oluruz. Ben firmama, „ehliyetim var, binek oto kullanabilirim“ dedim. Bu şu anlama geliyor: ben Tasit interface sınıfında yeralan vitesDegistir(), gazVer(), frenle() gibi metodları tanıyorum ve kullanıyorum. Firmam bana soyut olan Taşıt veremez, 4 tekerleği, vitesi, gaz ve fren pedalı olan bir binek oto vermek zorunda. Bana firmam tarafından verilen araç Ford marka bir binek oto. Ford firması Taşıt interface sınıfını tanıdığı için, içinde yeralan vitesDegistir(), gazVer(), frenle() gibi metodların yazılımını yapıyor, yani Tasit interface sınıfının dış dünyaya vermiş olduğu sözleri (bunlar sahip olduğu metotlar) yerine getiriyor. O halde benim ihtiyaç duyduğum metodlar Ford marka binek otoda da var olduğu için (çünkü Tasit interface sınıfını implemente etmiştir) , bu aracı kullanabiliyorum.

Ford sınıfının, Tasit interface sınıfı tarafından verilen tüm sözleri tutabilmesi için, Tasit interface sınıfında bulunan tüm metodları impelemente etmesi gerekiyor. İşte bu sebepten dolayı tasit.getMarka() metodunu kullanarak, aracın markasını ögreniyorum. Bu araç bir Audi oldugu icin „Audi A4“ seklinde aracın markası bana bildiriliyor. Audi sınıfının, Tasit interface tarafından sunulan getMarka() metodunu implemente etmesi, yani metod gövdesini oluşturması gerekiyor. Aksi taktirde Audi bir taşıt olamaz.

Java interface sınıflarının, kullanıcı sınıflar için bir nevi sözleşme metni oluşturduğunu gördük. Bir interface sadece dış dünyaya sunulacak hizmetleri tanımlar. Bu hizmetlerin nasıl yerine getirileceğine, interface sınıfını implemente eden altsınıflar karar verir. Interface kullanıcısı genelde hangi altsınıf üzerinden gereken hizmeti aldığını bilmez, bilmek zorunda değildir. Bu sayede „loose coupling“ olarak tabir edilen, servis saglayıcı ve kullanıcı arasında gevşek bir bağımlılık oluşturulmuş olur. Bu yazılım mimarisi açısından büyük bir avantaj sağlamaktadır. Interface sınıflar kullanılarak, sisteme esnek ve ilerde servis kullanıcılarını etkilemeden eklemeler yapilabilir. Firma aracı örneğine dönecek olursak: Firmanın Ford ve Audi marka iki binek otosu bulunmaktadır. Kısa bir zaman sonra Fiat marka bir araç daha satın alınır. Burada yapılmasi gereken, sadece Fiat isminde bir alt sınıf oluşturup, Tasit interface sınıfını implemente etmektir. Bu şekilde sistemi, diğer bölümleri etkilemeden genişletmiş oluyoruz. Sürücüler, sadece Tasit inteface sınıfını tanıdıklarından (ehliyet sahibi olduklarından), onlara firma tarafından hangi aracın verildiği önemli değildir.

Kısaca interface sınıflar, servis kullanıcılarından, kompleks yapıdaki alt sınıfları saklamak ve servis sunucusu ve saglayıcı arasındaki bağımlılığı azaltmak için kullanılır.

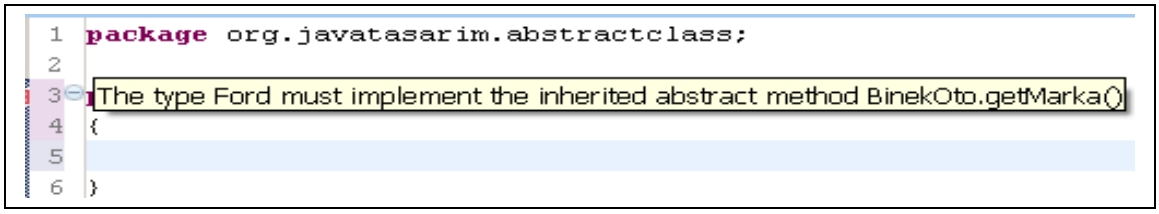
Abstract (Soyut) Sınıf Nedir?

Ortak özellikleri olan nesneleri modellemek için Java dilinde soyut sınflar kullanılır. Soyut sınıflardan interface sınıflarında olduğu gibi somut nesneler oluşturulamaz.

```
package org.javatasarim.abstractclass;

/**
 * BinekOto sinifi
 *
 * @author Oezcan Acar
 *
 */
public abstract class BinekOto
{
	private String marka;
	private int ueretimYili;
	private int vitesAdedi;

	/**
	 * Altsiniflar tarafindan implemente
	 * edilmek zorunda olan bir metod.
	 * @return marka String
	 */
	public abstract String getMarka();

	/**
	 * Altsiniflar tarafindan kullanilabilir
	 * yada tekrar re-implemente edilebilir
	 * bir metod.
	 */
	public int getUeretimYili()
	{
		return this.ueretimYili;
	}
}
```

Interface sınıflarının aksine, soyut sınıflarda istenilen sınıf metodları normal bir Java sınıfında olduğu gibi implemente edilebilir.

BinekOto.java sınıfı bir soyut sınıftır. getMarka() isminde bir soyut metodu ve getUeretimYili() isminde normal bir metodu bulunmaktadır. getMarka() soyut olduğundan, alt sınıfların bu metodu mutlaka implemente etmeleri gerekmektedir. Aksi taktirde Java compiler (derleyici) resim 3 de görüldüğü şekilde hata mesajı verecektir.



Resim 3

Ford isminde, BinekOto sınıfındaki özellikleri devralan bir alt sınıf tanımlıyoruz. Bunun için Java dilinde **extend** direktifi kullanılır.

```
package org.javatasarim.abstractclass;

/**
 * Ford marka binek otomobil
 *
 * @author Oezcan Acar
 *
 */
public class Ford extends BinekOto
{
	public String getMarka()
	{
		return "Ford Focus";
	}

	/**
	 * getUeretimYili() metodu üst sinifta
	 * tanimlanmistir ve alt sinifta degistirilerek
	 * tekrar re-implemente edilebilir yada
	 * degistirilmeden kullanilabilir.
	 */
	public int getUeretimYili()
	{
		System.out.println(super.getUeretimYili());
		return super.getUeretimYili();
	}
}
```

BinekOto sınıfında getMarka() soyut olarak tanımlandığı için, alt sınıflarda mutlaka getMarka() isminde implementasyonu yapılmıış bir metodun bulunması gerekmektedir. Ford sınıfı, tanımlamak zorunda olmadan, BinekOto sınıfında yeralan getUeretimYili() metodunu

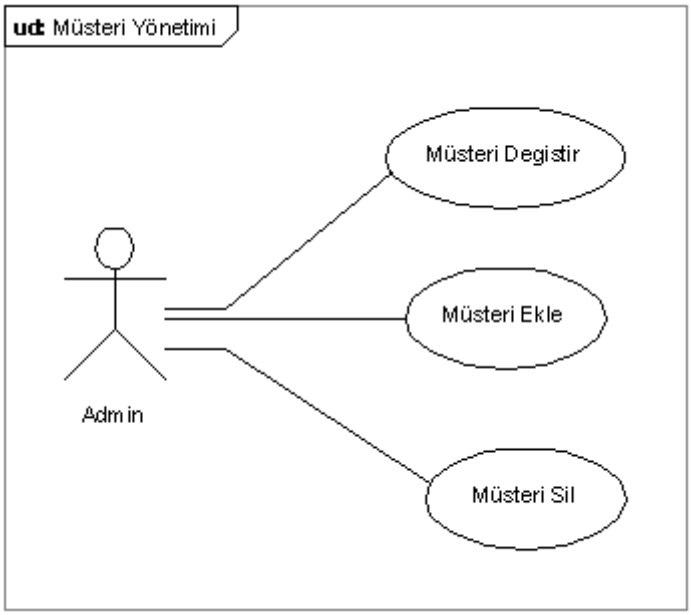
ve sınıf degiskenlerini kullanabilir. Java gibi nesneye yönelik programlama (OOP = object oriented programming) dillerinde, ortak değerler bir ana sınıfta toplanır ve alt sınıflarda, sadece alt sınıf için gerekli ekleme ve değişiklikler yapılarak kullanılır. Alt sınıflar, istedikleri taktirde, üst sınıfta tanımlanmış bir metodu, kendi bünyelerinde değiştirebilirler (reimplementasyon).

Soyut sınıflar, interface sınıfların aksine tecrübeli Java programcıları tarafından yazılımda tercih edilmezler. Kompleks sınıf hiyerarşileri, kodun bakımını ve genişletilmesini engeller. Bu problemi çözmek için çesitli tasarım şablonları üretilmiştir. Bunlardan bir tanesi Composite tasarım şablonudur. Composite tasarım şablonu ile, soyut sınıflar ve bu sınıfların alt sınıflarını oluşturan impelentasyonları birbirinden ayırmak ve bu şekilde sınıflar arası bağımlılığı azaltmak mümkündür.

Interface Örneği

Özellikle yazılım kitaplarında, verilen teorik bilgilerin kod örnekleri ile açıklanması okuyucu için büyük önem taşımaktadır. Kendim programcı olduğum için, sadece teorilerin yeraldığı bir kitabı okuyup, anlamanın zor olduğunu ve kitabı, ilk bölümü okuduktan sonra okumaya devam etmenin ne kadar güç olduğunu biliyorum. Bu nedenle, bu kitapta okuyucuya anlatmak istediğim konuları örnekler ile açıklamaya çalıştım.

Interface sınıflar hakkında okuduklarımızı bir örnek yazılım ile tekrar gözden geçirelim. Bir firmanın müşterilerini bilgibankasında tutmak istediğimizi düşünelim. Yapılması gereken işlemler, aşağıda yeralan UML Usecase[1] diagramında gösterilmiştir. Programımız yeni bir müşteriyi bilgibankasına ekler, mevcut bir müşteriyi silebilir veya müsteri bilgilerini değiştirebilir.



Resim 4

[1] Bakınız: http://www.agilemodeling.com/artifacts/useCaseDiagram.htm

Programımızı, ilerde oluşabilecek teknolojik değişikliklerden etkilenmeyecek bir şekilde tasarlamak istiyoruz. Müsteri isminde bir sınıf tanımladık. Müşteri için gerekli tüm bilgiler bu sınıf içinde tanımlandı.

```
package org.javatasarim.interfaces.muesteriyoenetimi;

/**
 * Müsteri sinifi
 *
 * @author Oezcan Acar
 *
 */
public class Muesteri
{
	private String isim;
	private String soyad;
	private String adres;

	public String getAdres()
	{
		return adres;
	}
	public void setAdres(String adres)
	{
		this.adres = adres;
	}
	public String getIsim()
	{
		return isim;
	}
	public void setIsim(String isim)
	{
		this.isim = isim;
	}
	public String getSoyad()
	{
		return soyad;
	}
	public void setSoyad(String soyad)
	{
		this.soyad = soyad;
	}
}
```

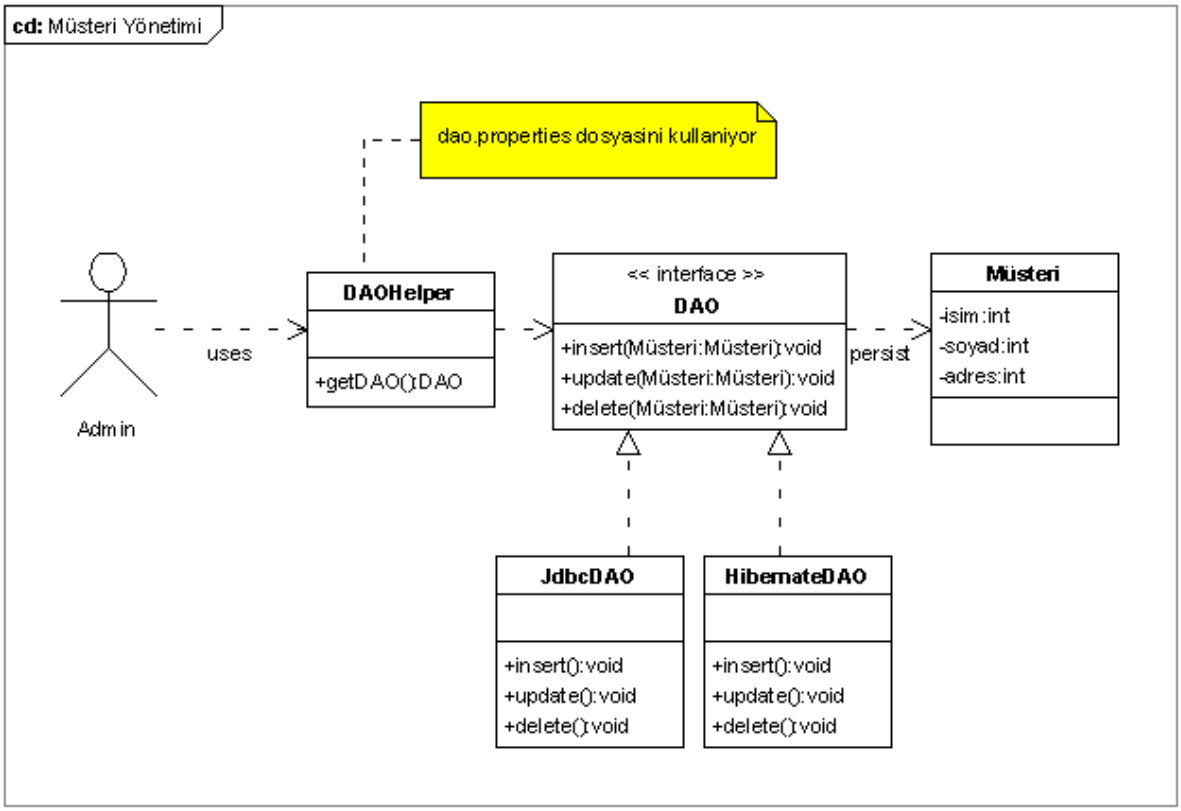
Bu sınıf içinde yeralan bilgiler bilgibankasındaki Müsteri tablosuna eklenecektir. Müşteri bilgilerini bilgibankasına kayıt edebilmek için kullanabileceğimiz 2 yöntem mevcuttur:

- JDBC – Direk bilgibankası sürücü (JDBC Driver) üzerinden bilgibankasına bağlanıp insert, delete, update SQL komutlarını kullanabiliriz
- Hibernate[2] ya da IBatis[3] gibi bir framework kullanarak, JDBC kodu yazmaya gerek kalmadan, Musteri sınıfıii bilgibankasına ekleyebiliriz.

Şu an için Hibernate hakkında yeterli bilgimiz olmadığından dolayı, JDBC yöntemini kullanmayı tercih ediyoruz. Proje menajerimiz esnek bir mimari tasarlamamızı istiyor.

[2] Bakınız: http://www.hibernate.org
[3] Bakınız: http://www.ibatis.org

Tasarımını yapacağımız müşteri yönetimi programı, ilerde sorun çıkmadan, JDBC yerine Hibernate ve ya IBatis kullanabilmelidir. Buradan yola çıkarak aşağıda yeralan tasarımı oluşturuyoruz.



Resim 5

UML sınıf diagramında, DAO, Müsteri, DAOHelper, JdbcDAO ve HibernateDAO isimlerinde sınıflar tanımladık. Bu sınıfların görevlerinin ne olduğuna bir göz atalım:

- **Müsteri:** Bu sınıf bünyesinde, bir müşteri için gerekli bilgiler yeralır (isim, soyad, adres).
- **DAO (Data Access Object):** Bu interface ile, müşteri nesneleri tarafından kullanılacak bilgibankası işlemleri tanımlanır (insert, update, delete).
- **DAOHelper:** DAOHelper dao.properties dosyasında tanımlanmış DAO sınıfı implementasyonunu edinmemizi sağlar.
- **JdbcDAO:** Bir müşteri nesnesini JDBC teknolojisi ile bilgibankasına ekler, siler, değiştirir.
- **HibernateDAO:** Müsteri sınıfını Hibernate frameworkunu kullanarak bilgibankasına ekler, siler, değiştir.
- **dao.properties:** Program tarafından kullanılacak DAO implementasyonunu tanımlar.

DAO interface sınıfını tanımlayarak, bir müşteri nesnesinin oluşumu ve program içinde kullanımı ile bu nesnenin bilgibankasına eklenmesi işlemini birbirinden ayırmış olduk. Programımız bir müşteri nesnesini bilgibankasına eklemek istediğinde, DAOHelper sınıfı üzerinden bir DAO implementasyonu alarak, gerekli işlemi yapacaktır.

Peki böyle bir mimarinin esnekliği nereden kaynaklanıyor? Bu noktaya açıklık getirelim. Programın burada tanıması gereken sadece DAO interface sınıfıdır. DAO interface sınıfı direk kullanılamıyacağı icin (çünkü bir interface), implementasyon sınıflarını oluşturmamız gerekiyor. Bunlar ilk etapta JdbcDAO ve HibernateDAO sınıflarıdır. dao.properties dosyası içinde kullandığımız DAO implementasyon sınıfını belirterek, istediğimiz esnekliğe kavuşuyoruz. Programı tekrar derlemek zorunda kalmadan, dao.properties dosyasını değiştirerek HibernateDAO sınıfını kullanabiliriz yada başka bir DAO sınıfı oluşturabiliriz (örnegin FileDAO). dao.properties dosyasının yapısını ve nasıl kullanıldığını biraz sonra göreceğiz.

```
package org.javatasarim.interfaces.muesteriyoenetimi;

/**
 * Test sinifi
 *
 * @author Oezcan Acar
 *
 */
public class Test
{
	public static void main(String args[])
	{
		Muesteri muesteri = new Muesteri();
		muesteri.setIsim("Ahmet");
		muesteri.setSoyad("Yildirim");
		muesteri.setAdres("Sisli / Istanbul");

		/**
		 * Müsteri nesnesi bilgibankasina
		 * insert edilir.
		 */
		DAOHelper.getDAO().insert(muesteri);
	}
}
```

Yukarda yeralan Test sınıfında bir müşteri nesnesinin oluşturulması ve DAOHelper üzerinden bilgibankasına eklenmesi işlemini görüyoruz. DAOHelper sınıfı hangi DAO implementasyon sınıfının kullanıldığını nereden biliyor?

```
package org.javatasarim.interfaces.muesteriyoenetimi;

/**
 * DAOHelper sinifi
 *
 * @author Oezcan Acar
 *
 */
public class DAOHelper
{
	public static DAO getDAO()
	{
		try
		{
			/**
```

```
                     * dao.properties icinde yeralan
                     * dao.impl anahtarinin degerini
                     * okuyarak, DAO interface
                     * sinifini implemente etmis bir nesne
                     * olusturur.
                     */
                    return ((DAO)Class.forName(PropertyHandler.
                                    getProperty("dao.impl")).newInstance());
              }
              catch (Exception e)
              {
                    e.printStackTrace();
              }
              return null;
        }
}
```

DAOHelper.getDAO() statik metodu ile, dao.properties içinde yeralan DAO implementasyon sınıfına ulaşıyoruz. Bunun için PropertyHandler isminde bir sınıftan yararlanıyoruz.

```
package org.javatasarim.interfaces.muesteriyoenetimi;

import java.util.ResourceBundle;

/**
 * PropertyHandler  sinifi.
 *
 * @author Oezcan Acar
 *
 */
public class PropertyHandler
{
      public  static String getProperty(String key)
      {
            return ResourceBundle.getBundle("dao").getString(key);
      }
}
```

PropertyHandler sınıfı, ResourceBundle sınıfını kullanarak, classpath içinde dao.properties isminde bir dosyayı arıyor. Bu dosyanın içeriği bir sonraki tabloda yeralmaktadır.

```
## DAO implementasyon sınıfı
dao.impl = org.javatasarim.interfaces.muesteriyoenetimi.JdbcDAO
```

dao.properties dosyasındaki satırlar anahtar / değer (key/value) değerlerinden oluşmaktadır. *dao.impl* anahtarı için kullanmak istediğimiz DAO implementasyon sınıfı olan JdbcDAO sınıfının paket ismini yazıyoruz: **org.javatasarim.interfaces.muesteriyoenetimi.JdbcDAO**

DAOHelper.getDAO() metodu içinde kullanmak istediğimiz anahtarı belirterek (dao.impl), Class.forName() ile yeni bir JdbcDAO nesnesi oluşturuyoruz (newInstance()).

DAOHelper sınıfı içinde aşağıdaki örnekte yeraldığı gibi alternatif bir yazılımda yapılabilirdi. Bu yöntemin bir dezavantajı var: Başka bir DAO implementasyonu kullanmak istediğimizde, DAOHelper.getDAO() metodunu değiştirip, tekrar derlememiz gerekecektir. Oysaki, biz yukarda yeralan örnekte olduğu gibi, yapılması gereken değişikliği dao.properties dosyasında tanımlıyarak, programı tekrar derleme ihtiyacı olmadan, başka bir DAO implementasyon sınıfı kullanabiliriz. Yapmamız gereken sadece yeni DAO sınıfını dao.properties dosyasına eklemektir.

```
package org.javatasarim.interfaces.muesteriyoenetimi;

/**
 * DAOHelper2 sinifi
 *
 * @author Oezcan Acar
 *
 */
public class DAOHelper2
{
	public static DAO getDAO()
	{
		try
		{
			return new JdbcDAO();
		}
		catch (Exception e)
		{
			e.printStackTrace();
		}
		return null;
	}
}
```

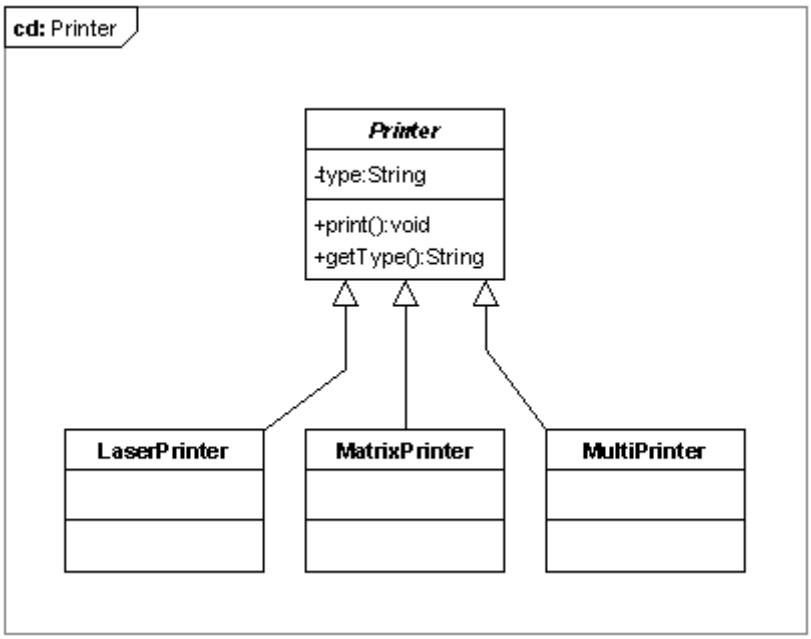
Programın ilk versiyonunda JdbcDAO sınıfını kullanma kararı veriyoruz. İstediğimiz zaman yeni bir DAO implementasyonu oluşturarak, programın diğer bölümlerini değiştirmek zorunda kalmadan kullanabiliriz.

```
package org.javatasarim.interfaces.muesteriyoenetimi;

import java.sql.Connection;
import java.sql.PreparedStatement;

public class JdbcDAO implements DAO
{
	public void insert(Muesteri muesteri)
	{
		Connection con = null;
		PreparedStatement pstmt = null;
		try
		{
			con = getConnection();
			pstmt = con.prepareStatement("insert into muesteri " +
					"(isim,soyad, adres) values(?,?,?)");
			pstmt.setString(1, muesteri.getIsim());
			pstmt.setString(2, muesteri.getSoyad());
			pstmt.setString(3, muesteri.getAdres());
			pstmt.executeUpdate();
		}
```

```
            catch (Exception e)
            {
            }
      }

      public void update(Muesteri muesteri)
      {
            //TODO
      }

      public void delete(Muesteri muesteri)
      {
            //TODO
      }

      public Connection getConnection()
      {
            //TODO
            return null;
      }
}
```

Neden Interface ve Abstract Sınıflar Yeterli Değil?

Önce soyut sınıfların yazılım yaparken sebep oldukları sorunlara bir göz atalım.



Resim 6

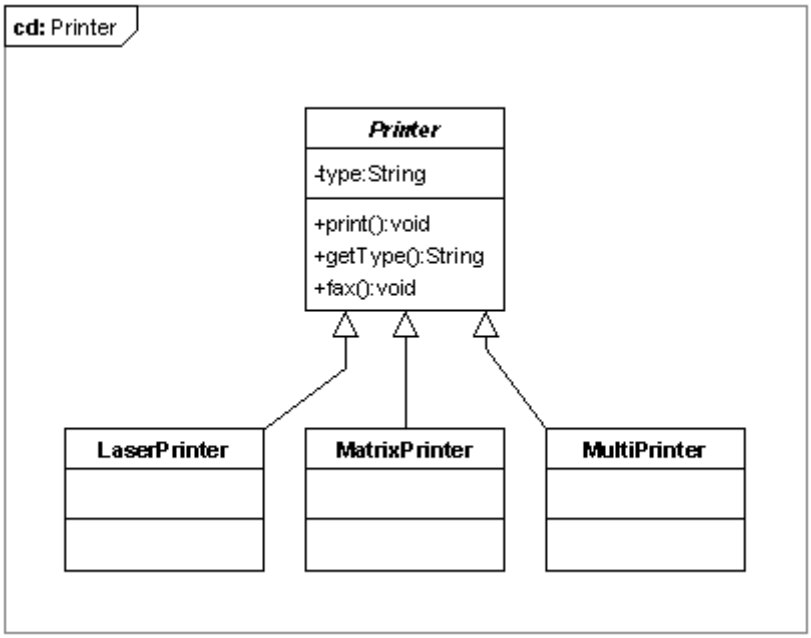
Printer isminde genel anlamda bir yazıcıyı modelleyen bir soyut sınıf oluştuyoruz. Amacımız değişik tipteki yazıcılar için bir üstsınıf oluşturmak ve ortak olan özellikleri bu sınıfın bünyesinde toplamak.

```java
package org.javatasarim.abstractclass.printer;

/**
 * Printer sinifi
 *
 * @author Oezcan Acar
 *
 */
public abstract class Printer
{
	private String type;


	public Printer()
	{
	}

	public Printer(String _type)
	{
		setType(type);
	}

	public void print()
	{
		System.out.println("Printed with "+getType());
	}

	public void fax()
	{
		System.out.println("");
	}

	public String getType()
	{
		return type;
	}

	public void setType(String type)
	{
		this.type = type;
	}
}
```

Tüm yazıcıların ortak yanı baskı işlemidir. Bunun icin print() isminde bir metod tanımlıyoruz. Oluşturulacak olan tüm alt sınıflar Printer sınıfının sahip olduğu tüm değişken ve metotlara sahip olacaktır ve böylece print() metodunu ortak kullanacaklardır.

Yazılım yaparken amacımız, yazdığımız kodun ilerde bakımının kolay ve genişletilebilir halde olmasını sağlamaktır. Zamanla değişik yapıda ve modelde yazıcı sistemlerinin piyasaya sürüleceği bir gerçektir. Peki yaptığımız model bu gelişmeleri bünyesine katabilecek yapıda mıdır? Bunu deneyelim ve görelim.

Modelimizde MultiPrinter isminde, faks gönderme fonksiyonuna sahip bir yazıcı bulunmaktadır. Faks gönderebilmek icin fax() isminde bir metodun oluşturulması gerekiyor.



Resim 7

Alt sınıflara tek tek fax() metodunu eklemek ve implemente etmek istemediğimiz için Printer sınıfına fax() isminde bir metod ekliyoruz. İmplemente ettigimiz fax() metodunu tüm alt sınıflar kullanabilir.

Bu noktadan itibaren bir sorunla karşı karşıyayız! Printer sınıfına eklenen bir metod tüm alt sınıflar tarafından kullanılır. Bu durumda faks gönderme özelliğine sahip olmayan LaserPrinter yada MatrixPrinter fax() gönderme metoduna sahip olarak, faks gönderme özelliğine kavuşuyorlar. Lakin fiziksel olarak bu yazıcılar faks gönderemezler, yani farkında olmadan bazı altsınıflar için gereksiz davranışlar (bir nesnenin davranışı sahip olduğu bir metotdur.) oluşturduk. Bu şartlar altında faks gönderme fonksiyonu olmayan bir yazıcıyı faks göndermek icin kullanabiliriz ve bu bir hatadır!

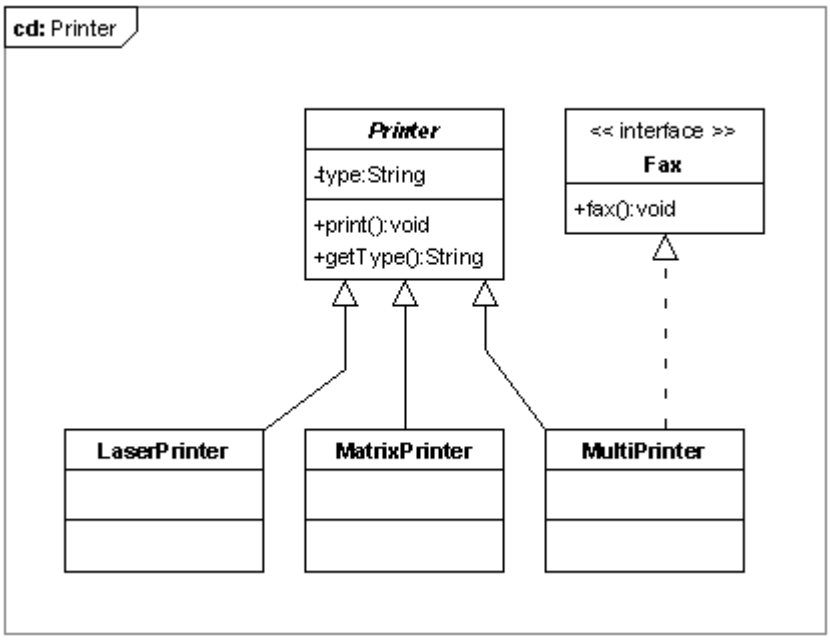
Bu sorunu, fax() metodunu altsınıflarda, yazıcının yapısına göre tekrar implemente ederek çözebiliriz:

```
package org.javatasarim.abstractclass.printer;

/**
 * LaserPrinter sinifi
 *
 * @author Oezcan Acar
 *
 */
public class LaserPrinter extends Printer
{
	@Override
	public void fax()
	{
		// LaserPrinter fax gönderemdigi icin bu metod
		// bos kalmak zorundadir.
	}
}
```

LaserPrinter faks gönderme özelliğine sahip olmadığı için fax() metodunun bu sınıf bünyesinde boş kalması gerekiyor.

```
package org.javatasarim.abstractclass.printer;

/**
 * MultiPrinter sinifi
 *
 * @author Oezcan Acar
 *
 */
public class MultiPrinter extends Printer
{
	@Override
	public void fax()
	{
		System.out.println("Fax sent with " + getType());
	}
}
```

Soyut bir sınıf kullanıldığı taktirde, program yapısının yapılacak değişikliklere pek açık olmadığını gördük. Peki bu sorunu bir interface sınıf kullanarak çözebilir miyiz? Aşağıdaki gibi bir çözüm düşünülebilir:



Resim 8

fax() metodu sadece bazı yazıcılar tarafından desteklendiği için Fax isminde bir interface sınıfı oluşturarak, faks gönderebilecek yazıcıların bu sınıfı implemente etmesini sağlıyoruz. Böylece gerçekten faks gönderebilen yazıcılar bu özelliğe kavuşuyor.

```
package org.javatasarim.abstractclass.printer;

/**
 * Fax interface sinifi
 *
 * @author Oezcan Acar
 *
 */
public interface Fax
{
      void fax();
}
```

```
package org.javatasarim.abstractclass.printer;

/**
 * MultiPrinter sinifi
 *
 * @author Oezcan Acar
 *
 */
public class MultiPrinter extends Printer implements Fax
{
      public void fax()
      {
```

```
            System.out.println("Fax sent with " + getType());
        }
}
```

Peki sizce bu iyi bir çözüm müdür? Bizi bekleyen sorunları kestirebiliyor musunuz? Faks gönderme işlemini değiştirmek zorunda kaldığımızı düşünelim, bu durumda sistemde ne gibi değişiklikler yapılması gerekiyor?

Interface sınıfı kullandığımız zaman karşılaştığımız ilk sorun, her altsınıfın interface sınıfında yeralan metodları implemente etmesi gerçeğidir. Elliye yakın altsınıfın bulunduğu bir sistem düşünün. Fax() metodunu değiştirmek zorunda kaldığımızda, bu elli sınıfın fax() metodunu değiştirmek zorunda kalacağız. Gereksiz yere aynı kodu çoğaltıp, program içinde kullanmış oluyoruz. Bu sistemin bakımını çok zorlaştırır ve mutlaka ve mutlaka sakınılması gereken bir durumdur.

Modelimizi tekrar gözden geçirdigimiz zaman, soyut sınıfların kullanılmasının sorunumuzu çözmekte yeterli olmadığı görüyoruz. Aynı şekilde interface sınıfları kullanarak sorunumuzu çözemiyoruz, çünkü Java'da bulunan interface sınıflarında kod implementasyonu yapmak mümkün değildir. Bu sebepten dolayı her interface metodunun alt sınıflarca implemente edilmesi gerekmektedir ve buda aynı kodların tekrar tekrar başka sınıflarda kullanılmasına ve yapı itibariyle bakımı zor kalıpların oluşmasına sebebiyet vermektedir. Kısaca interface sınıfları hakkında şunu diyebiliriz: "Interface sınıfları kodun tekrar kullanılmasını sağlayamaz". Ama bu bizim amacımız olmalı: "Mümkün mertebe yazılan bir sınıf yada metot kodu sistemin başka bir yerinde kullanılabilmelidir. Bu en önemli nesneye yönelik prograglama presiplerinden biridir.

Tasarim Prensipleri

İyi bir tasarım oluşturabilmek için bazı prensiplerin bilinmesi ve uygulanması gerekmektedir. Bunları yakından inceliyelim.

Değişken Bölümleri Tespit Et!

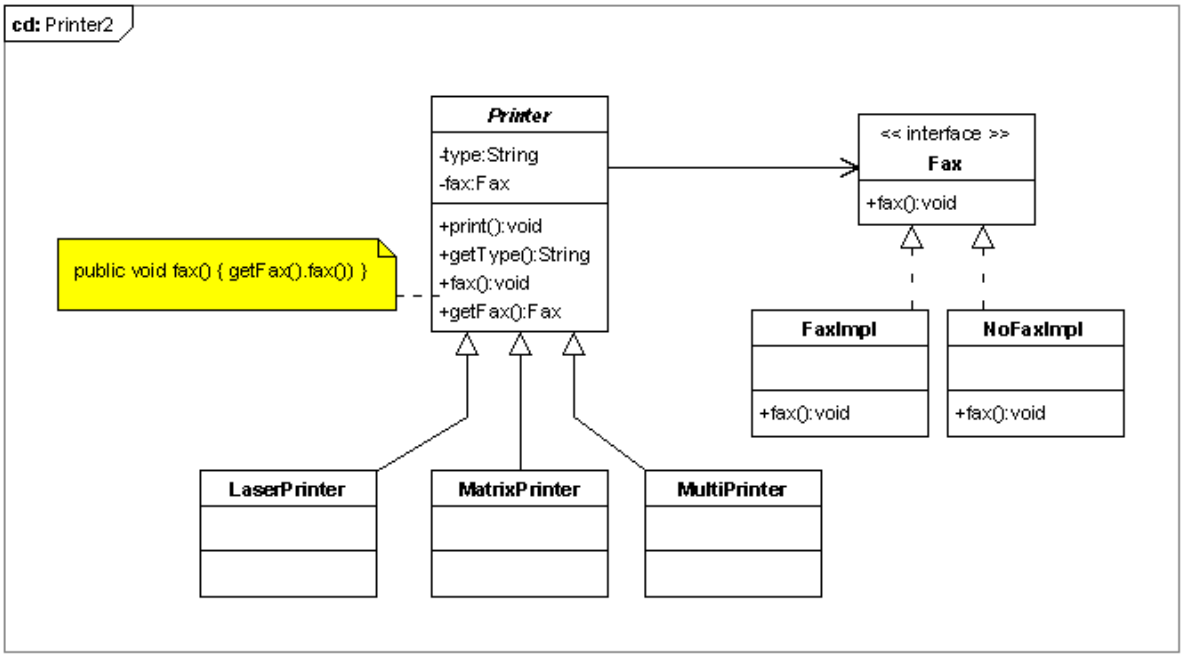
Bununla, sınıf içinde kullanılan değişkenleri (variable) değil, daha ziyade program içinde değişikliğe ugrayabilecek bölümleri kastediyoruz. Yapı itibariyle değişikliğe ugrayabilecek bölümler, programın kırılgan bir hale gelmesine sebep verebilir. Bu sebeple bu gibi bölümlerin, değişikliğe ugramayan bölümlerden ayrılması gerekmektedir. Sabit kalan bölümler sağlam yapıdaki programların temelini oluşturur.

Örnegin X isminde bir sınıf müşterinin istekleri doğrultusunda devamlı değişikliğe uğruyorsa, değişikliğe uğrayan bölümlerin isole edilip, sistemin diğer bölümlerinden ayrılması gerekmektedir. Aksi taktirde X sınıfının ve sahip olduğu metodların kullanıldığı diğer sınıfları, X sınıfının yapısı değiştiği için değistirmek zorunda kalırsınız.

Değişken bölümleri tespit ettikten sonra, gerekli sınıf ve metodları interface sınıfları arkasında saklıyarak, sistemin diğer bölümlerini etkilemeden bu bölümler üzerinde istediğimiz şekilde değişiklik yapabiliriz. Programın diğer bölümleri bu interface sınıflarında yeralan metodları kullandıkları için, değişken bölümler üzerinde yapacağımız değişikliklerden

etkilenmiyeceklerdir. Kitabın diğer bölümlerinde yakından inceleyeceğimiz tasarım şablonlarının temelinde bu prensip yatmaktadır.

Printer örneğinde değişken olan bölüm nedir? Soyut olan Printer sınıfına eklediğimiz fax() metodu, bazı altsınıflarca istemediğimiz davranışlara sebep vermişti. Bu sebepten dolayı her alt sınıf bünyesinde fax() metodunu yeniden implemente etmek zorunda kalmıştık, yani elli alt sınıf için elli değişik yada aynı yapıda fax() implementasyonu yapılmak zorunda. Şimdi degişken olan bölümü görebildiniz mi? Evet, değişen bölüm fax() metodunun implementasyonudur. Elli sefer aynı kodu elli değişik sınıfta kullanmak yerine, fax metodunu isole edip, bir interface sınıfı arkasında saklayabilir ve Printer sınıflarını bu interface sınıfındaki metodları kullanaya zorlayabiliriz. Bunun nasıl olabilecegine bir göz atalım.



Resim 9

İlk olarak Fax isminde bir interface sınıf tanımlıyoruz. Bu sınıf fax() isminde bir metoda sahip. Akabinde FaxImpl isminde, fax() metodunu implemente eden bir altsınıf oluşturuyoruz. Faks işlemi için gerekli yapıyı bir noktada yani FaxImpl.fax() metodunda toplamış oluyoruz. Faks gönderme özelliğine sahip olmayan yazıcılar için NoFaxImpl isminde ikinci bir implementasyon sınıfı oluşturuyoruz. Bu sınıfın fax() metodu, faks gönderilemeyeceğine dair bir mesajı ekranda görüntüler.

İkinci adımda Printer sınıfında Fax tipinde bir sınıf değişkeni tanımlıyoruz:

```
package org.javatasarim.abstractclass.printer2;

/**
 * Printer sinifi
 *
 * @author Oezcan Acar
 *
 */
public abstract class Printer
```

```
{
	private String type;

	private Fax fax;


	public void fax()
	{
		getFax().fax();
	}

	public Fax getFax()
	{
		return fax;
	}

	public void setFax(Fax fax)
	{
		this.fax = fax;
	}

	public Printer()
	{
	}

	public Printer(String _type)
	{
		setType(type);
	}

	public void print()
	{
		System.out.println("Printed with "+getType());
	}


	public String getType()
	{
		return type;
	}

	public void setType(String type)
	{
		this.type = type;
	}
}
```

Dikkatimizi çeken, Fax tipinde yeni bir sınıf değişkeninin (fax) tanımlanmış olması. Burada değişken tipi olarak Fax interface sınıfını kullanıyoruz. Printer sınıfının alt sınıflarından nesneler oluşturulduğunda, fax değişkeni herhangi bir Fax implementasyonu olabilir.

Printer sınıfına fax() isminde yeni bir metod ekliyoruz. Faks gönderme işleminini fax isimli sınıf değişkenine delege ediyoruz. Bu noktadan itibaren Printer sınıfı faks göndermek için koda sahip olmak yani fax metodunu implemente etmek zorunda değildir. Printer sınıfı, Fax interface sınıfını kullanarak, bu görevi Fax interface sınıfına delege etmektedir. Faks gönderme sorumluluğu Fax interface sınıfına aittir ve Printer sınıfından alınmıştır. Artık alt sınıflar tarafından Printer.fax() metodunun ayrı ayrı implemente edilme zorunluluğu ortadan kalkmıştır. Tüm alt sınıflar ortak olarak Printer.fax() metodunu kullanabilir.

Bir sonraki örnekte görüldüğü gibi, altsınıflar gerekli Fax interface implementasyonunu kullanarak faks gönderip, gönderemiyeceklerine kendileri karar verirler.

```
package org.javatasarim.abstractclass.printer2;

import org.javatasarim.abstractclass.printer2.MultiPrinter;

/**
 * Test sinifi
 *
 * @author Oezcan Acar
 *
 */
public class Test
{

	public static void main(String[] args)
	{
		Printer multi = new MultiPrinter();
		multi.setFax(new FaxImpl());
		multi.fax();

		Printer laser = new LaserPrinter();
		laser.setFax(new NoFaxImpl());
		laser.fax();
	}
}
```

Birinci prensibe göre, programın değişken bölümlerini isole etmemiz gerekiyor. İsole edilmiş bölümleri interface sınıfları arkasına saklayarak, sistemin diğer bölümlerince herhangi bir değişikliğe maruz kalmadan kullanılmalarını sağlamamız gerekiyor. Buradan yola çıkarak ikinci tasarım prensibi ile tanışıyoruz: "Her zaman interface sınıflarına karşı program yazılmalı". İkinci prensip uygulandığı taktirde, interface sınıfları arkasında saklı kod üzerinde yapılan değişiklikler, sistemin diğer bölümlerini etkilemez.

Bir önceki bölümde yeralan örnekte faks gönderme işlemini ve gerekli kodu Fax interface sınıfı arkasına sakladık. Kullanıcı (örneğin MultiPrinter) sadece interface sınıfını ve sahip olduğu fax() metodunu tanıyor. Biz istediğimiz bir Fax implementasyonu kullanarak (örneğin FaxImpl) sistemin davranışını dinamik olarak değiştirebiliyoruz. Bu durumda kullanıcı sınıfının faks gönderme işlemi hakkında hiç bir bilgiye ihtiyacı kalmamaktadır. Kullanıcı sınıf için sabit olan Fax.fax() metodu hicbir zaman degişmeyecektir. Değişen, duruma göre Fax implementasyonu olacaktır. Kullanıcı sınıfı etkilemeden istedigimiz şekilde fax() metodu kodunda değişiklik yapabiliriz.

İkinci prensibi uygulayabilmek için, sistem içinde kullanılan değişken tiplerinin interface ya da soyut sınıf türünden olmaları gerekmektedir.

```
Printer multi = new MultiPrinter();
multi.setFax(new FaxImpl());
multi.fax();
```

Multi ismindeki değişkenin tipi Printer soyut sınıfıdır. Program çalışır durumda iken (runtime) multi değişkenine Printer soyut sınıfını implemente eden her hangi bir altsınıf eşitlenebilir. Buna Java'da polimorfi (polymorphy) ismi verilir. Polimorf yapıdakş tüm değişkenlere sahip oldukları tipi implemente eden her hangi bir alt sınıf eşitlenebilir.

Kalıtım (inheritance) Yerine Kompozisyon Kullan!

Printer sınıfında Fax tipi bir değişken kullandık. İki sınıf bu şekilde bir araya getirildiği zaman kompizisyon oluşur. MultiPrinter gibi alt sınıflar kalıtım yöntemiyle fax metoduna sahip olmak yerine, kompozisyon aracılığıyla faks işlemini başka bir sınıfa delege ederler. Böylece faks gönderme sorumluluğu başka bir sınıfa geçmis olmaktadır.

Kompozisyon bir çok tasarım şablonunda kullanılmaktadır. Kompozisyon aracılığıyla esnek yapıda programlar oluşturabiliriz.

Bu arada ilk tasarım şablonu ile tanışmış olduk: "Strategy (strateji) tasarım şablonu".

Strategy tasarım şablonu ile değişik tipte algoritmalar tanımlanır (örneğin faks gönderme, faks gönderememe, faksı email ile gönderme vs). Bu algoritmalar aynı interface sınıfını (Fax) implemente ettiklerinden, değiş tokuş edilebilirler. Strategy tasarım sablonu ile kullanıcı sınıflar kullanılan algoritmalardan bağımsız bir şekilde programlanabilir. Detaylı bilgiyi Strategy Tasarım Şablonu bölümünde bulabilirsiniz.

Nesneler Arası Esnek Bağ Oluştur!

İki nesne arasındaki bağ esnek ise, bu nesneler birbirleri hakkında fazla bilgiye sahip olmadan beraber çalışabilirler. Esnek bağ prensibine dayanan tasarımlar ile bakımı ve geliştirilmesi kolay sistem yazılımı yapılabilir, çünkü esnek bağ nesneler arasındaki bağımlılık oranını azaltır.

Örneğin nesneler arası esnek bir bağ oluşturmak icin Observer (gözlemci) tasarım şablonu kullanılabilir. Daha sonra detaylı inceliyeceğimiz bu tasarım şablonu ile, bir nesnede meydana gelen değişikliklerden haberdar olmak isteyen nesneler bir araya getirilir. Bu bir nevi abonelik sistemi gibidir ve belirli bir nesnede meydana gelen değişikliklerden haberdar olmak isteyen nesneler bu nesneye abone olurlar. Haber veren ve haberdar olan nesneler arasında Observer tasarım sablonu ile çok esnek bir bağ oluşturulur. Haberdar olmak isteyen nesneler belirli bir interface sınıfını implemente ederek, abone olma özelliğine kavuşurlar. Bunun haricinde haberdar olmak isteyen nesneler üzerinde değişiklik yapılmaz. Abone olmak isteyen nesne adedi sınırlı değildir ve her abone için haber veren nesne üzerinde değişlik yapılmak zorunda kalınmaz. Haber veren ve haberdar olan nesneler birbirlerinde bağımsız bir şekilde her zaman kullanilabilir. Haber veren nesne üzerinde yapılan bir değişiklik haberdar edilen nesneleri etkilemez. Aynı şekilde haberdar olan nesneler üzerinde yapılan değişiklik, abone olunan nesneyi etkilemez. Görüldüğü gibi bu şekilde nesneler arası esnek bağların oluşturulması, sistemin dinamik ve esnek bir yapıda olmasını sağlar.

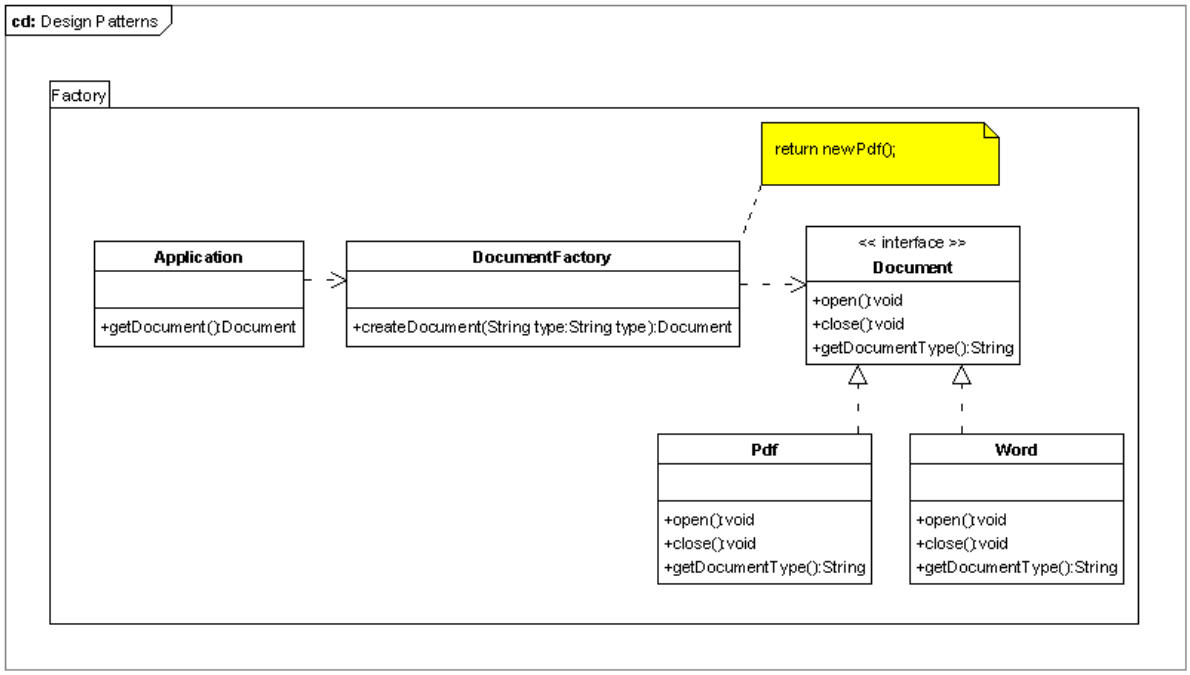
Sınıflar Geliştirmeye Açık, Degiştirilmeye Kapalı Olmalıdir!

Zaman içinde kullanıcıların yazılım sistemden beklentileri değişebilir. Bu durumda yeni kullanıcı isteklerinin sisteme dahil edilmesi gerekecektir. Esnek tasarım metodları kullanılarak hazırlanmış bir sistem, yapılması gereken değişikliklere açık olacaktır. Amacımız yazılan kodu değiştirmek zorunda kalmadan sistemi, yeni fonksiyonlar ekliyerek geliştirmek olmalıdır. Değişik tasarım sablonları kullanılarak, mevcut sınıflar, değiştirilmek zorunda kalmadan yeni fonksiyonlarla genişletilebilir.

Örneğin Decorator (dekoratör) tasarım sablonu kullanılarak mevcut bir sınıf ve bu sınıfın nesnelerine yeni davranış biçimleri (metot) eklenebilir. Bunu elde edebilmek için sınıfın yapısını değiştirmemiz gerekmiyor. Amacımız da bu olmalıdır: “mevcut kodu değiştirmeden sisteme yeni davranış biçimleri eklemek!”

Somut Sinifları Kullanma!

Program içinde somut sınıfların kullanılması kullanıcı sınıf ile somut sınıf arasında yüksek derecede bağımlılık oluşturur. Somut sınıf üzerinde yapılan bir değişiklik kullanıcı sınıfı etkileyebilir. Bu yüzden somut sınıf yerine soyut (abstract, interface) sınıflar kullanılmalıdır.



Resim 10

Daha sonra detaylı olarak inceleyeceğimiz Factory (fabrika) tasarım şablonu ile, kullanıcı sınıfın (UML diagramında Application) somut sınıflara olan bağımlılığı azaltılır. Kullanıcı sınıf new Pdf() ile bir Pdf dokümanı oluşturduğu anda, bu somut sınıfa bağımlı hale gelir. Factory tasarım şablonu bu bağı çözerek, doküman oluşumunu kullanıcı sınıfından saklar ve böylece kullanıcı sınıf ile Pdf sınıfı arasında esnek bir bağ oluşturur. Böyle bir bağı oluşturabilmek icin Document isminde bir interface yada soyut sınıfın olusturulması gerekmektedir. Kullanıcı sınıf, bünyesinde Document tipinde bir değişken barındırarak, istediği tipte bir dokümanı Factory aracılığıyla edinebilir.