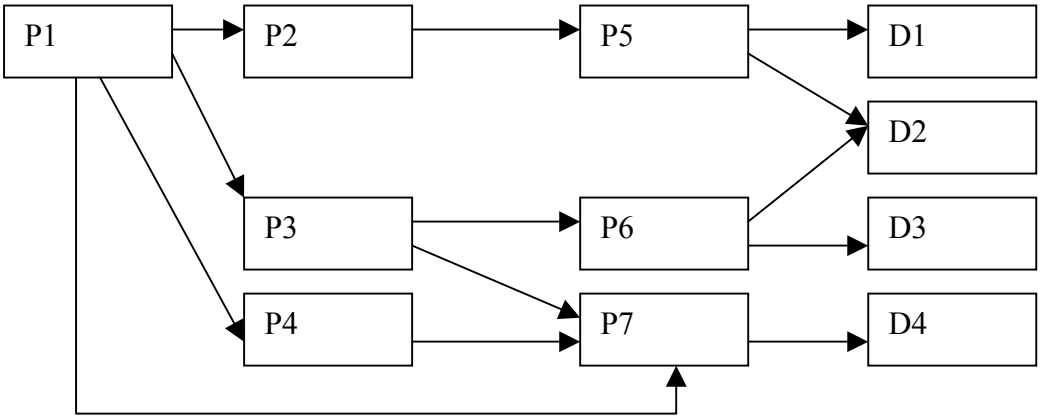
# Neden Nesneye Yönelik (object-oriented) Programlama?

Nesneye yönelik programlamanın sunduğu olanakların yeterli şekilde değerlendirilebilmesi için önce geleneksel yordamsal (procedural) programlama yaklaşımından kaynaklanan sorunların ele alınması gerekir.

**Yordamsal (Procedural) Programlama**

Yordamsal programlamada yazılımlar birbirini çağıran bir dizi yordam (procedure) ve işlevler topluluğu olarak geliştirilir. Her yordam ve işlev kendi yerel verisini, yine kendi yerel değişkenlerinde tutar. Paylaşılması gereken veriler yordam çağırma komutlarında parametre olarak yordamdan yordama geçirilir. Parametrelere sığmayacak büyük veriler ise genel (global) değişkenler içerisinde herkesin kullanımına açılır. Yordamsal programlama yaklaşımının ilke çizimi aşağıdadır:




Şekil 1. Yordamsal Programlama Yaklaşımı

Şekil 1'de dört adet genel kullanıma açılmış veri (D1-D4) ve bunları doğrudan ya da dolaylı olarak kullanan 7 adet yordam (P1-P7) yer almaktadır.

**Yordamsal Programlamanın Zayıf Yanları**

- Genel kullanıma açılan veriler tümüyle korumasız kalır.
- Verinin kullanım amacı veri üzerinde yapılabilecek işlemleri hiçbir biçimde sınırlamaz.
- Veriyi tutan değişken genel kullanıma açıldıktan sonra, o değişken türünün desteklediği her türlü işlem veriye uygulanabilir.
- Amaç dışı kullanımdan kaynaklanan yanlışlıklar ortaya çıktıktan sonra, yanlışlığa neden olan program kesiminin saptanması zordur. Bunun için sözkonusu veriye erişen tüm yordamların tek tek incelenmesi gerekir.

Örnek:

Bir banka hesabının bakiyesini tutan sayısal türde bir değişkeni ele alalım ve bu değişkenin arada bir hiç beklenmedik biçimde değer değiştirdiğini saptadığımızı varsayalım. Bu genel değişkeni kullanan tüm yordamlar incelendikten sonra programcının yanlışlıkla çok benzer adlı bir değişken yerine bakiye değişkeni tutan değişkendeki değerin karekökünü aldığı belirlenmiş olsun. Bu tür bir yanlışlığın önlenmesi için aşağıdaki önlemlerin programcı tarafından alınması gerekecektir:

- Hesap bakiyeleri üzerinde sayısal değişkenlere sadece toplama ve çıkarma işlemlerinin uygulanması anlamlıdır.
- Ek olarak bakiye değişkeni üzerinde toplama ve çıkarma işlemlerinde de sınırlamalar sözkonusudur. Bakiyeyi eksi değere düşüren çıakrma işlemlerine izin verilmemelidir.
- Bankacılık uygulamasında hesap bakiyesini değiştiren her işlem hesap özetinde görülen izleme kayıtları yaratır. İzleme kaydı yaratmaksızın yapılan toplama ve çıkarma işlemleri de engellenmelidir.

Yordamsal programlamada bu tür denetimler uygulama içerisinde dolaylı yoldan programcı tarafından yapılır.

- Yordamsal programlamada verinin saklanma biçimi, veriye erişim ve verinin işlenme biçimini doğrudan etkiler. Aynı verinin sayısal bir değişken yerine karakter türü bir değişkende tutulmasına karar verildiğinde, tek boyutlu dizi yerine iki boyutlu dizi kullanıldığında vs. Sözkonusu veriye erişen tüm yordamların elden geçirilmesi gerekir. Bu durumda yöntem değişikliğinden etkilenen tüm kod kesimleri eksiksiz saptanmalı ve gereken güncellemeler hatasız olarak yapılmalıdır. Daha sonra programlar yeniden sınanır, derlenir ve kullanıcılara dağıtılır. Bu işlemler sırasında bazı kod kesimlerinin atılması ve yenilerinin eklenmesi de gerekebilir. Sonuç olarak, her aşamada sisteme yeni yanlışların sızması ya da dolaylı ilişkilerden ötürü ancak belli süre sonunda saptanabilecek ve bulunması daha zor hataların ortaya çıkması olasılığı yüksektir.
- Yordamsal programlamada eldeki kodun yeniden kullanımına dönük altyapı zayıftır. Farklı işletmelerdeki belli işlevlerdeki küçük farklılıklar geliştirilen programın özel durumlar için özel kesimler içermesine ya da programın farklı durumlar için farklı kopyalarının üretilmesine neden olur. İlk seçenekte varolan genel amaçlı parametrik programların tasarımı, gerçekleştirilmesi ve sınanması daha zor, pahalı ve zaman alıcıdır. Bunların anlaşılması ve bakımı da zordur. İkinci seçenekte ise bir kopyada saptanan hatanın diğer kopyalarda düzeltilmesi, güncellemelerin yapılması çok zahmetli ve tekrarlıdır. Uygulamada bir kopyada yapılan değişikliğin her kopyaya yansıtılamadığı ve her kopyanın başarısının farklılaştığı görülmektedir.

Sonuç olarak yordamsal programlamadaki zorluklar şöyle özetlenebilir:

- Hataların saptanma ve düzeltilmesindeki zorluk
- Programın kullanıcı gereksinmelerini ve yenilik isteklerini tam karşılayacak biçimde hızla değiştirilebilir olmaması ve her yapılan ek ya da değişikliğin programın daha önce çalışan ve değiştirilmeyen kesimlerinde bile hataların oluşmasına yol açabilmesi
- Eldeki sınanmış kod kesimlerinin ciddi değişiklikler yapılmaksızın yeni gereksinmelerin karşılanmasında kolayca kullanılamaması

**Nesneye Yönelik Programlama**

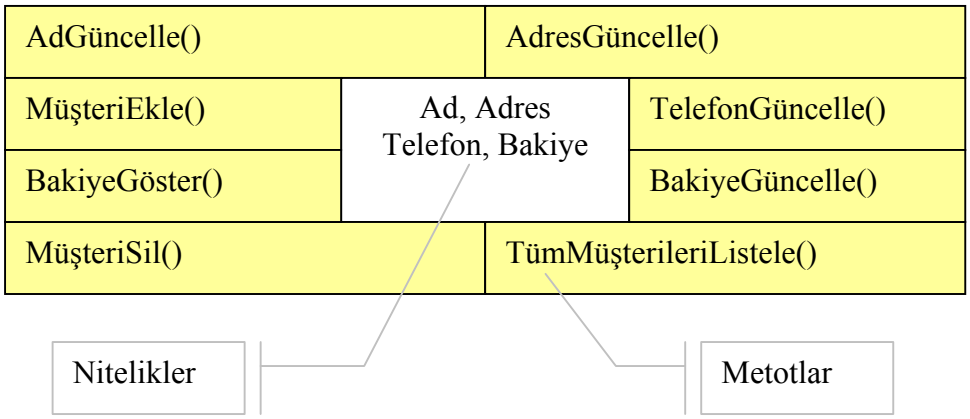
Nesneye yönelik programlamada yazılımlar birbirleriyle iletişim kurabilen nesneler topluluğu olarak tasarlanır ve gerçekleştirilirler. Kod işletimi nesnelerin içinde yapılır ve her nesne bir diğer nesneye ileti göndeerek ondan hizmet alabilir. Bu nedenle nesneye yönelik programlama "nesnelere ileti gönderme yoluyla programlama" olarak da isimlendirilir.

Nesneler *metodlar* (methods) ve *nitelikler*'den (attributes) oluşur. Nitelikler, nesnelerin sahip oldukları *verilere*, metodlar ise bunlar üzerinde yapılabilecek *işlemlere* karşılık gelir. Bir başka deyişle nesne, kendisini işleyecek kod kesimini kendisi ile birlikte tanımlayan ve taşıyan ve kendi tanımladığı biçimden daha farklı amaçlarla kullanılamayan veri türü olarak yorumlanabilir. Nesnelerin genel özellikleri aşağıda tanımlanmıştır:

- Nesneler *sınıflar*'dan (class) yaratılırlar. Sınıf, nesnenin yaratılmasını sağlayan bilgileri içeren yapının adıdır. Bu nedenle sınıf, bir nesne şablonu olarak tanımlanabilir. Bir sınıftan türetilen tüm nesneler aynı şekilde davranır; bir diğer deyişle, hepsi *aynı metot çağrılarına cevap verirler*.

- Nesneler *kalıcı* (persistent) ve geçici (transient) olabilirler. Nesneler ana bellekte yaratılan ve ana bellekte kullanılabilen yapılardır. Dolayısıyla, önlem alınmazsa içinde yer aldıkları program sonlandırıldığında kaybolurlar.
- Nesnelerin, niteliklerinde tutulan değerlerle tanımlanan durumları (state) vardır. Bir başka deyişle, aynı sınıfa bağlı nesneleri birbirinden ayıran şey işleyiş biçimleri ya da yetenekleri değil, içlerinde tuttukları verilerin değerleridir.
- Nesnelerin yaşam döngüleri vardır. Bu yaşam döngüsü içinde yaratılırlar, kullanılırlar ve yok edilirler.
- Nesneler dış dünyada arayüzleri (interface) ile bilinirler ve kullanılırlar. Bir nesnenin arayüzü, o nesnenin dış dünyanın kullanımına açtığı metotların listesinden oluşur. Bir diğer deyişle, arayüzler nesnelerin hangi hizmetleri sağladığını belirtir, ancak bunları nasıl yapacağı konusunda bilgi içermezler.

Aşağıda bir müşterinin ad, soyad, telefon ve bakiye bilgilerini tutan bir sınıf (class) tanımı görülmektedir. Müşterinin bilgilerini içeren değişkenlere dış dünyadan erişim olanaklı değildir. Bu bilgilere erişmek, okumak ya da değiştirmek isteyen bir program, bu sınıfın metotlarından birini - *Bakiye göster () ya da Adres Güncelle () gibi* - kullanmak zorundadır.

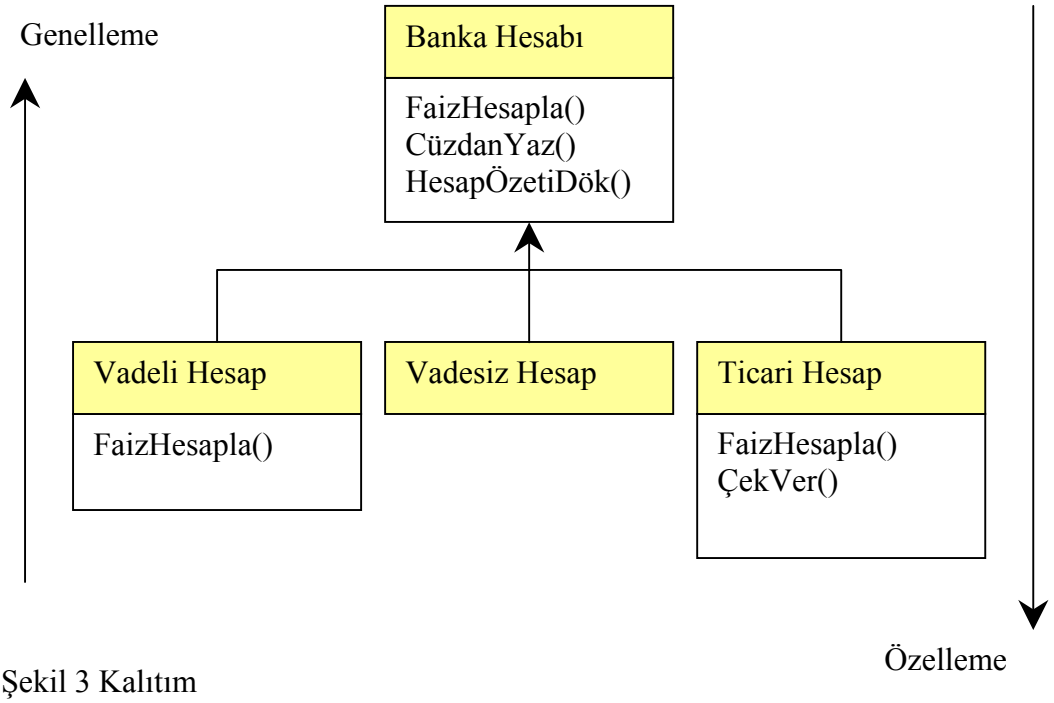



Şekil 2 Sınıf (class) Örneği (Müşteri)

Yazılımlar giderek daha fazla kullanıcı tarafından kullanıldıkça, yeni ihtiyaç ve isteklerin ortaya çıkması kaçınılmazdır. Öte yandan her iyileştirme, eklenti ve değişiklik isteminin yazılımın tasarımını az ya da çok etkilediği ve ek çaba gerektirdiği açıktır. Dolayısıyla her yazılımcının değişiklik istemlerinin ana tasarım üzerindeki sarsıcı etkilerini olabildiğince en aza indirgeyerek bu istekleri karşılamasını sağlayacak araçlara ihtiyacı vardır. Nesneye yönelik programlama, olası değişikliklerin etkisini kısıtlamak üzere aşağıdaki mekanizmaların kullanımını önermektedir:

- Kalıtım (inheritence)
- Çok biçimlilik (polymorphism)
- Sarmalama (encapsulation)

Kalıtım, eldeki sınıfları kullanarak yeni sınıflar yaratabilme özelliğidir. Bu işlem sırasında, yeni yaratılan sınıfın tüm özellikleri yerine yalnızca varolan sınıftan hangi noktalarda farklı olduğunun belirtilmesiyle yetinilir. Aşağıda bir bankacılık uygulamasında kullanılan kalıtım mekanizması örneği verilmiştir:




Şekil 3 Kalıtım

Bu örnekte bankanın üç farklı hesap türü kullandığı görülmektedir. Aslında tüm hesaplar genel bir banka hesabı kavramı kapsamına girer. Bir başka deyişle, cüzdan yazdırma ve hesap özeti dökme gibiişlemler türünden bağımsız olarak her banka hesabı için geçerlidir. Örnekte verilen *genel banka hesabı sınıfından* (class) türetilen *vadesiz hesap sınıfı*, genel hesap sınıfında tanımlanan *metotlarda* herhangi bir değişiklik yapılmasını gerektirmemiştir. Vadeli hesap sınıfı ise faiz hesabını farklı yaptığı için faiz hesaplama *metodunu* yeniden tanımlamıştır. Ticari hesap sınıfı ise yalnızca faiz hesaplama metodunu yeniden tanımlamakla kalmamış, diğer hesap sınıflarında bulunmayan çeklerle ilgili yeni bir metot daha tanımlamıştır.

Kalıtım mekanizması, eldeki kodun yeniden kullanımı, değişiklik ve eklemelerin kolayca yapılabilmesi konusunda yordamsal programlama yaklaşımına oranla daha esnek bir yapı sunmaktadır. Örneğin üs-sınıfın (superclass)HesapÖzetiDök() metodunun gerçekleştirimi değiştirilip iyileştirildikçe tüm alt-sınıf (subclass) nesneleri bundan otomatik olarak yararlanır. Yeni bir hesap türünün sisteme eklenmesi ise Banka Hesabı sınıfından yeni bir alt-sınıf türetilmesiyle gerçekleştirilmektedir. Bu yeni hesap sınıfının eklenmesi esnasında diğer alt-sınıf kodlarının bundan etkilenip etkilenmediğinin denetlenmesi gerekli değildir, çünkü eklentinin etkileri yeni alt-sınıf ile sınırlıdır.
Alt-sınıf nesneleri, üst-sınıf nesneleri ile aynı temel özelliklere ve metotlara sahiptir. Bu özellik üst-sınıf nesnelerinin kullanıldığı her yerde alt-sınıf nesnelerinin kullanımına olanak verir ve *çokbiçimlilik* (polymorphizm) olarak isimlendirilir. *Sarmalama* (encapsulation) ise veri ve işlevlerin tek bir nesne biçiminde tanımlanabilmesi olarak açıklanabilir.

## KAYNAK

**Baransel, C., Mumcuoğlu, A.,** Web Tabanlı, Üç Katmanlı Yazılım Mimarileri: UML, EJB ve ORACLE İle Sistem Modelleme, Tasarım ve Gerçekleştirim, SAS Yayınları, 2003.