1. Giriş: Nesneye Dayalı Programlama (OOP)

1.1. Nesneye Dayalı Programlama Nedir?

Nesneye Dayalı Programlama (NDP), gerçek dünyanın bilgisayar ortamında modellenmesini temel alan bir yazılım geliştirme yaklaşımıdır. Bu yaklaşımda, çözülmek istenen problem bir işlevler bütünü olarak değil, sistemi oluşturan nesnelerin etkileşimi olarak ele alınır.

Geleneksel yaklaşımların aksine, tasarım sürecinde "Bu sistem ne iş yapar?" sorusu yerine "Bu sistem hangi nesnelerden oluşur?" sorusu merkeze alınır. Bu modellemede nesnelerin yapısı iki temel bileşenden oluşmaktadır:

1. **Nitelikler (Attributes):** Nesnenin özelliklerini veya durum bilgilerini ifade eder.
2. **Davranışlar (Behaviors):** Nesnenin yeteneklerini ve sorumluluklarını temsil eder.

Örneğin, bir personel takip sistemi tasarlanırken sistemin işlevi yerine; "memur", "işçi", "müdür" gibi varlıklar birer nesne olarak modellenir.

1.2. Neden Nesneye Dayalı Programlama Kullanılır?

Yazılım sektöründe OOP'nin bir standart haline gelmesi ve tercih edilmesinin temel nedenleri, önceki programlama yaklaşımlarının (Doğrusal ve İşleve Dayalı/Prosedürel) yetersiz kaldığı alanları iyileştirme ihtiyacından doğmuştur.

A. Önceki Yöntemlerin Kısıtlamaları:

- **Karmaşıklık Yönetimi:** İlk dönemlerde kullanılan doğrusal programlama (BASIC vb.) yöntemlerinde, program büyüdükçe kodun okunması ve yönetilmesi "goto" deyimleri nedeniyle zorlaşmaktaydı.
- **Veri Güvenliği Sorunu:** Daha sonra geliştirilen İşleve Dayalı (Procedural) yaklaşımda (Pascal, C vb.) problemler fonksiyonlara bölünerek çözülse de, veri gizleme (data hiding) olanağının kısıtlı olması, verilerin bozulma riskini artırmaktaydı.
- **Gerçek Hayat Uyumu:** Gerçek dünyadaki sistemler sadece fonksiyonlardan oluşmadığı için, prosedürel yöntemlerle gerçeğe yakın bir model oluşturmak zordu.

B. OOP'nin Sağladığı Avantajlar: Sağlanan kaynakta belirtildiği üzere, nesneye dayalı yaklaşımın ve bu yaklaşımla birlikte kullanılan UML (Birleşik Modelleme Dili) standartlarının sağladığı başlıca faydalar şunlardır:

- **Hata Minimizasyonu:** Programdaki mantıksal hataların (bug) en aza indirilmesini sağlar.
- **Yeniden Kullanılabilirlik (Reusability):** Tasarım aşamasının düzgün yapılması durumunda, yazılan kodlar tekrar tekrar kullanılabilir; bu da geliştirme maliyetini büyük ölçüde düşürür.
- **Bakım ve Güncelleme Kolaylığı:** Sisteme yeni öğeler eklemek veya güncellemek, eski fonksiyonel yapıya göre çok daha kolaydır. Dokümantasyonu yapılmış kodların düzenlenmesi daha az zaman alır ve programın kararlılığını artırır.
- **Ekip Çalışması:** Büyük projelerde sistemi parçalara ayırmak ve parçalar arasında ilişki kurmak, programcıların iletişimini ve ortak çalışmasını kolaylaştırır.

2. Prosedürel ve OOP Yaklaşımlarının Karşılaştırılması

1. Veriye Erişim ve Okunabilirlik

- **Prosedürel Kodumda:** Verileri listelerde tuttuğum için, bir kitabın adına veya yazarına ulaşmak istediğimde sürekli indeks sayılarını hatırlamak zorundaydım. Örneğin, kodun içinde kitap[0] yazdığımda bunun kitap adı mı yoksa yazar mı olduğunu anlamak için listenin tanımlandığı yere geri dönüp bakmam gerekiyordu. Bu durum hata yapma riskimi artırdı.
- **OOP Kodumda:** Verileri nesneler içinde paketlediğim için kitap.ad veya kitap.yazar şeklinde isimleriyle çağırdım. Bu sayede kodum adeta düz bir cümle gibi okunaklı hale geldi, indeks ezberlemekten kurtuldum.

2. Veri Güvenliği (Kapsülleme)

- **Prosedürel Kodumda:** Kütüphane listesi (kutuphane) herkese açıktı. Kodun herhangi bir yerinde yanlışlıkla kitap[4] = "Yırtıldı" yazsam program bunu kabul ederdi ama sistemin mantığı bozulurdu (çünkü sadece "Mevcut" veya "Ödünç Verildi" olmalıydı).
- **OOP Kodumda:** Book sınıfında self.__durum değişkenini gizleyerek (private yaparak) dışarıdan erişimi kapattım. Değişiklik yapmak için yazdığım set_durum() metoduna kontrol ekledim. Böylece artık istesem de geçersiz bir durum giremiyorum, kodum beni koruyor.

3. Kod Tekrarının Önlenmesi (Kalıtım)

- **Prosedürel Kodumda:** Eğer sisteme "Personel" diye yeni bir kullanıcı türü eklemek isteseydim, isim ve ID alma işlemlerini tekrar baştan yazmam gerekecekti.
- **OOP Kodumda:** User adında bir ana sınıf oluşturdum ve Member sınıfını ondan türettim (Inheritance). super().__init__ komutu sayesinde, ortak olan isim ve ID kodlarını tekrar yazmadan doğrudan miras aldım. Bu sayede kodum daha temiz ve kısa oldu.

4. İşlem Mantığı (Çok Biçimlilik)

- **Prosedürel Kodumda:** Ekrana yazdırma işlemleri için listele fonksiyonunu tek bir kalıpta yazdım. Özelleştirmek zordu.
- **OOP Kodumda:** bilgi_goster() metodunu her sınıf için (User, Member, Book) ayrı ayrı tanımladım. uye.bilgi_goster() dediğimde, program o nesnenin bir "Üye" olduğunu anlayıp, normal kullanıcıdan farklı olarak ödünç aldığı kitapları da listeledi. Kodum nesneye göre şekil değiştirebilir (Polymorphism) hale geldi.

3. Kapsülleme, Kalıtım ve Çok Biçimlilik Kavramlarının Uygulanışı

A. Kapsülleme (Encapsulation) Kullanımım

Kapsüllemeyi, projemdeki veri güvenliğini sağlamak ve dışarıdan yapılacak hatalı müdahaleleri engellemek amacıyla Book sınıfında kullandım.

1. **Veriyi Gizleme:** Kitapların "Mevcut" ya da "Ödünç Verildi" bilgisini tutan durum değişkenini herkesin değiştirmesini istemedim. Bu yüzden __init__ metodumda bu değişkeni self.durum yerine self.__durum (başına iki alt çizgi koyarak) tanımladım. Böylece bu değişkeni "Private" (Gizli) hale getirdim; yani sınıf dışından doğrudan erişime kapattım.

2. **Kontrollü Erişim (Getter ve Setter):** Gizlediğim bu veriye erişilmesi gerektiğinde (örneğin kitabı listelerken durumunu görmek için) get_durum(self) adında bir metot yazdım. Bu metot sadece veriyi okumaya izin veriyor, değiştirmeye değil.
3. **Veri Doğrulama:** Durumun değiştirilmesi gerektiğinde (kitap ödünç verilirken) ise set_durum(self, yenidurum) metodunu kullandım. Ancak buraya kritik bir kontrol (if bloğu) ekledim. Kodum, gelen yeni durum bilgisinin sadece ["Mevcut", "Ödünç Verildi"] listesinde olup olmadığını kontrol ediyor. Eğer geçerli bir değerse değişikliği yapıyor, değilse reddediyor. Böylece kitabın durumunun "Kayboldu", "Yırtıldı" gibi tanımsız bir hale getirilmesini engelledim.

B. Kalıtım (Inheritance) Kullanımım

Kalıtımı, kod tekrarını önlemek ve mantıksal bir hiyerarşi kurmak için User ve Member sınıfları arasında uyguladım.

1. **Ana Sınıfın Tasarımı:** Önce, tüm kullanıcı türlerinde (ister personel olsun ister üye) ortak olan özellikleri (isim ve user_id) barındıran User adında bir ana sınıf (Base Class) tasarladım.
2. **Miras Alma İşlemi:** Kütüphane üyelerini tanımladığım Member sınıfını oluştururken, isim ve id tanımlamalarını tekrar yazmak yerine class Member(User): diyerek User sınıfından miras aldım.
3. **Super() Fonksiyonu:** Member sınıfının yapıcı metodunda (__init__), super().__init__(isim, user_id) komutunu kullandım. Bununla şunu amaçladım: "İsim ve ID atama işini benim yerime üst sınıfım olan User yapsın." Böylece ben Member sınıfında sadece üyeye özel olan self.odunc_alinanlar listesini tanımladım. Bu sayede kodumda gereksiz tekrarlardan kaçındım.

C. Çok Biçimlilik (Polymorphism) Kullanımım

Çok biçimliliği, aynı isme sahip bir metodun farklı nesneler üzerinde farklı davranışlar sergilemesini sağlamak için bilgi_goster() metodu üzerinden uyguladım.

1. **Metotların Farklılaşması (Overriding):**
    - **Book Sınıfında:** bilgi_goster() metodunu, kitabın adı, yazarı ve durumunu yazdıracak şekilde kodladım.
    - **User Sınıfında:** Aynı isimli metodu, sadece Kullanıcı ID ve İsim bilgisini ekrana basacak şekilde tasarladım.
    - **Member Sınıfında:** Burası en kritik noktaydı. Member sınıfı User'dan miras almasına rağmen, User'ın bilgi gösterme yöntemi bana yetmedi. Çünkü üyenin elindeki kitapları da göstermem gerekiyordu. Bu yüzden Member sınıfı içinde bilgi_goster() metodunu ezdim (Override ettim).
2. **Sonuç:** Kodun çalışma anında uye.bilgi_goster() komutunu verdiğimde; Python arka planda bu nesnenin bir Member olduğunu anladı ve User sınıfındaki basit versiyonu değil, benim Member için yazdığım detaylı (ödünç alınan kitapları da listeleyen) versiyonu çalıştırdı. Böylece tek bir komutla her nesne kendi doğasına uygun çıktıyı üretmiş oldu.

4. Sonuç: Oop Kullanımının Avantajlarına Dair Gözlemlerim

Bu projede aynı kütüphane senaryosunu önce prosedürel (fonksiyonel) yöntemle, ardından Nesneye Yönelik Programlama (OOP) ile kodlayarak iki yaklaşımı kıyaslama fırsatı buldum. Kodlama süreci ve elde ettiğim nihai ürün üzerinden OOP kullanmanın sağladığı avantajları şu başlıklar altında gözlemledim:

1. Hata Yönetimi ve Veri Bütünlüğü

Prosedürel versiyonu yazarken en büyük risk, verilerin merkezi bir kontrol mekanizması olmamasıydı. Kitap bilgilerini bir liste içinde tuttuğum için, kitabın durumunu güncellemek istediğimde doğrudan kitap[4] = "..." ataması yapmam gerekiyordu. Bu durum, yanlışlıkla "Mevcut" yerine "Bilinmiyor" gibi sisteme yabancı bir veri girilmesine veya yanlış indeks kullanımına (örneğin yıl bilgisi olan kitap[3]'ü değiştirmek gibi) çok açıktı.

OOP tarafında ise bu sorunu iki aşamalı çözdüm:

1. **Kapsülleme ile Koruma:** self.__durum değişkenini gizleyerek dışarıdan kontrolsüz erişimi kapattım.
2. **Mantıksal Doğrulama (Validation):** set_durum(self, yenidurum) metodunda yazdığım if yenidurum in ["Mevcut", "Ödünç Verildi"] kontrolü sayesinde, sisteme sadece önceden tanımladığım geçerli durumların girilmesini zorunlu kıldım.

Ayrıca, hata ayıklama (debugging) sürecinde __str__ metodunun avantajını gördüm. Prosedürel kodda bir nesneyi yazdırmak istediğimde sadece bellek adresini veya ham listeyi görüyordum. Ancak Book sınıfına eklediğim __str__ metodu sayesinde, print(kitap) dediğimde terminalde doğrudan {self.ad} - {self.yazar} ({self.yil}) formatında anlamlı bir çıktı aldım.

2. Kodun Yönetilebilirliği ve Modülerlik Prosedürel kodda tüm fonksiyonlar ve global değişkenler iç içe geçmiş durumdaydı; bir yerde yaptığım küçük bir değişiklik, zincirleme olarak diğer fonksiyonları bozabiliyordu. OOP yapısında ise her sınıfın kendi işinden sorumlu bağımsız bir modül olduğunu fark ettim. Örneğin Book sınıfının __str__ metodunda yaptığım bir görsel güncelleme, User sınıfının çalışmasını veya Library sınıfındaki ödünç verme mantığını hiç etkilemedi. Bu durum, hata ayıklama (debugging) sürecimi ciddi oranda hızlandırdı.

3. Geliştirme Kolaylığı ve Esneklik Sisteme "Üye" eklerken OOP'nin gücünü net olarak hissettim. Prosedürel yapıda bunu yapmak için sıfırdan "üye_ekle", "üye_sil" gibi fonksiyonlar yazmam gerekirken; OOP kodumda User sınıfından miras alarak (class Member(User)) dakikalar içinde yeni yapıyı kurabildim.

4. Bilişsel Yükün Azalması Kod yazarken zihnimde tutmam gereken detay sayısı OOP ile azaldı. Prosedürel kodda "Hangi veri hangi listenin kaçıncı sırasındaydı?" diye sürekli kodun başına dönüp bakmak zorundayken; OOP kodunda nesneler ve metodlar (örn: uye.kitap_al(kitap)) gerçek hayattaki cümleler gibi aktığı için sadece işin mantığına odaklanabildim.

5. Kendi Yorumum ve Değerlendirme

Bu ödev çalışması bana "kod yazmak" ile "sistem tasarlamak" arasındaki farkı öğretti. Prosedürel ve OOP yaklaşımlarını şu iki noktada değerlendiriyorum:

- **Okunabilirlik ve Bilişsel Yük:** Prosedürel kod büyüdükçe, "kitap[2] sayfaydı, kitap[4] durumdu" gibi teknik detayları zihinde tutma zorunluluğu "bilişsel yükü" artırarak asıl

mantığa odaklanmamı engelledi. OOP kodumda ise nesneler gerçek hayattaki karşılıklarıyla (uye.kitap_al(), kitap.get_durum()) ifade edilebildi.

- **Genişletilebilirlik ve Sürdürülebilirlik:** Prosedürel kodda yeni bir materyal eklemek, tüm indeks ve if-else yapılarının riskli bir şekilde güncellenmesini gerektirir. OOP'de ise Kalıtım (Inheritance) sayesinde mevcut yapıyı bozmadan, yeni sınıflar türeterek sistemi güvenle genişlettim.

Sonuç olarak; prosedürel yapı basit işler için yeterli olsa da, sürdürülebilir ve profesyonel sistemler için OOP kaçınılmaz bir mühendislik standardıdır.