Nesne Yönelimli Programlama | Object Oriented| OOP

ademcatamak.medium.com/nesne-yönelimli-programlama-oop-2425e8de1a

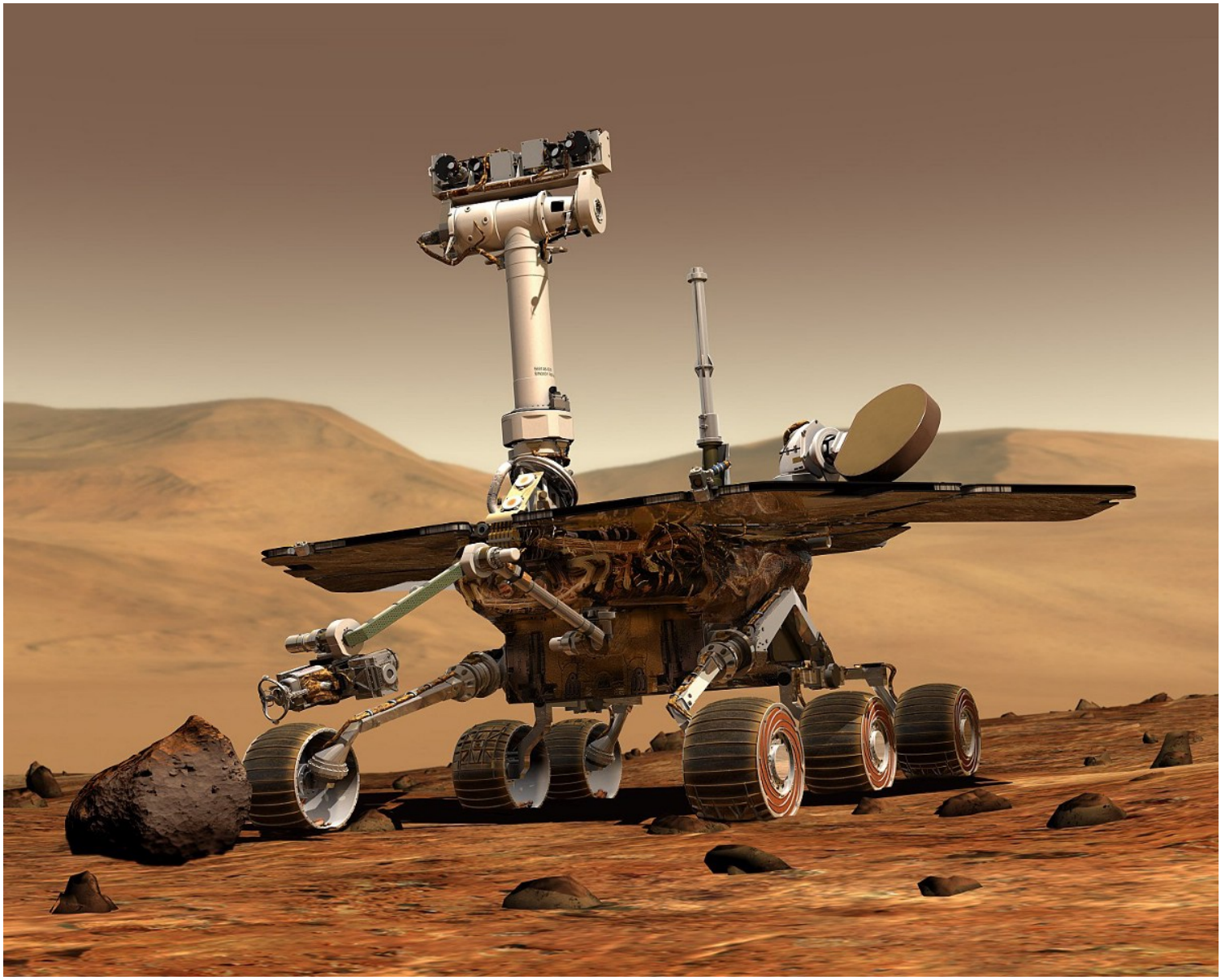
3 Ağustos 2020

Nesne Yönelimli Programlama - Object Oriented Programming (OOP)

Yazılım geliştirmenin altında yatan gerekçe, gerçek hayatta var olan sorunları birler ve sıfırlar dünyasında modelleyerek çözmektir. Modellemek için de farklı yöntemler kullanılabilir. Bunlardan biri nesne yönelimli programlamadır (obje yönelimli programlama) (OOP — Object Oriented Programming).

Nesne yönelimli programlama yaklaşımının altında yatan mantık, gerçek hayatta gördüğünüz varlıkları birer nesneye karşılık gelecek şekilde bilgisayar simülasyonuna aktarmaktır. Bir nesnenin yapabildiklerini veya özelliklerini bilgisayara tanıtmak için OOP üzerinde kullandığımız terimlerden en önemli iki tanesi; sınıf (class) ve arayüz (interface) terimleridir.

Bu makalede yer alan kod örnekleri, nesne yönelimli program geliştirmemize uygun olan C# dilinde olacaktır. Bu örneklerin alındığı projenin amacı ise MarsRover sorununa bir çözüm getirmektir.



Origin:

Arayüz (Interface)

Arayüz bir nesnenin hangi davranışları sergileyip sergilemediğini tanımlamak için kullanılır. Arayüzleri bir taslak gibi düşünebiliriz.

Yukarıdaki arayüzü implemente eden sınıfların üzerinde *GoForward* metodu olacaktır. Metodun içerisinde neler yapılacağı, her implemente eden sınıf tarafından kendine has kurallarla yazılacaktır. **Rover** sınıfı **IMovable** arayüzünü uygulayıp içerisinde 'tekerleri döndür' gibi bir emir yürütürken, **Human** sınıfı 'adım at' gibi bir emir yürütebilir. Nihayetinde, iki nesne de ileri gidebilme yeteneğine sahiptir.

Sınıf (Class)

Sınıf bir nesnenin ait olduğu tip olarak anlatılabilir. Örneğin **Human** (insan) isimli bir sınıf düşünürsek, **Adem** olarak Human sınıfının bir nesnesi olurum. Bir diğer örnek ise **Rover** sınıfı olabilir. Mars yüzeyinde gezen birçok robot olabilir. Robot nesnelerinin her biri Rover sınıfının örnekleridir. İniş yaptığımız yüzey ise Mars üzerinde var olabilecek birçok platodan biri olarak **Plateau** sınıfına bağlı bir nesnedir.

Sınıf Tipleri

Sınıflar soyut (abstract) ve soyut olmayan sınıflar olarak iki kategoriden oluşur. Soyut sınıflar çalıştığınız sorun üzerinde yer alan, daha genel olan yapılardır. MarsRover sorunu, uzak bir perspektiften bakarak yüzeyde var olan araçlarla alakalı bir sorun olarak özetlenebilir. Bu durumda sorunu çözmeye başlamak için **Surface** ve **Vehicle** tipinde soyut tipler tanımlanabilir. Çözüm için soruna biraz daha yaklaştığımız zaman elimizdeki aracın Vehicle sınıfının bir alt kırılımı olan Rover sınıfından olduğu görülür ve Rover sınıfı tasarlanmaya başlanabilir.

Soyut sınıflar soyut olmayan sınıflar gibi doğrudan kendilerine ait nesneler oluşturamazlar. Soyut sınıflardan türeyen soyut olmayan sınıflar tasarlanır ve bu sınıflar üzerinden örnekler elde edilir. Bu durum yukarıdaki örnek üzerinden açıklandığı zaman daha anlamlı olabilir. Örneğin, NASA'da çalışmaya başladınız. Mars yüzeyine gönderilmek üzere size bir araç talebi gelirse, sanırım aklınıza gelen ilk soru 'Nasıl bir araç?' sorusu olur. Bilgisayarda da bu durum geçerlidir. Bir araç oluşturulmak istendiği zaman, Vehicle sınıfının hangi alt kırılımına ait nesne üretilmesi gerektiğine dair muğlak bir durum oluşacağı için soyut sınıflar üzerinden nesne oluşturulamaz.

Soyut sınıfları kullanmamızın bir amacı kod tekrarından kaçınmaktır. Mars yüzeyinden örnek toplayabilecek bir araç tasarlamanız gerekirse Vehicle sınıfından yeni bir sınıf türetmeniz ve TakeSample metodunu yazmanız yeterli olacaktır. Bu şekilde Turn ve GoForward metotlarını tekrar yazmanıza gerek kalmayacaktır. Ayrıca Turn ve GoForward metotlarında bir değişiklik yapılmak istenmesi halinde bu değişikliğin tek bir yerden yapılabiliyor olması, sürdürülebilirlik açısından uygulamanıza pozitif etki yapacaktır.

Soyut sınıfları kullanmamızın bir diğer amacı ise ortak bir yapı ortaya koyabilmektir. Bunu yaparken kullanabileceğimiz özellik ise soyut metotlardır. Soyut metot şu anlama gelmektedir; sınıf bir kabiliyete sahiptir ama bu kabiliyeti nasıl ortaya koyduğu her alt kırılımda farklılık gösterecektir. Her yiğit yoğurt yer ama her yiğidin ayrı bir yoğurt yiyişi vardır.

Bir noktanın yüzey alanı üzerinde iz düşüme sahip olup olmadığı Surface nesnesine sorulabilir. Her yüzey alanı kendine ait bazı kuralları işleterek bu soruya cevap verebilmektedir. MarsRover örneğimizde Platoları dikdörtgen olarak tasarladık. Yüzey alanımız dikdörtgen olduğu için bir noktanın X ve Y değerinin belirlenen minimum ve maksimum değerin arasında olduğunu kontrol etmesi yeterli olacaktır. Surface sınıfının bir alt kırılımı olarak **Lowland** (ova) sınıfını tasarlamış olalım. Ovaların da üçgen olduğunu varsayalım. Ova sınıfı `*bool Contains(Point point)*` metodunu kendi kurallarına göre uygulayacaktır ama dışarıdan bakıldığı zaman aynı metot imzasına ova üzerinde de rastlanmış olacaktır.

Kalıtım (Inheritance)

Soyut sınıfları ve arayüzleri anlatırken bu konudan bahsetmiş olduk aslında. Kalıtım; sınıf ve arayüzlerin arasındaki ilişkinin modellenmesidir. Bir sınıfın veya arayüzün başka bir sınıf veya arayüzü üstlenerek aynı özellik ve davranışları sergilemesidir.

Bir sınıf bir başka sınıfı kimlik endişesiyle kalıtım alır. Örneğin, Human (insan) sınıfı Organism (organizma) sınıfını kendi kimliği gereği üstüne alır. Bir diğer deyişle "insan bir organizmadır" sözcüğünün yazılım dilindeki karşılığı, bir sınıfın başka bir sınıfı kalıtım almasıdır.

Arayüzler ise sergilenen yetenekler gereği üstlenilir. Human sınıfı GoForward veya Think gibi metotları IMovable veya IAbleToThink arayüzleri aracılığıyla elde eder. “İnsan koşabilir” veya “İnsan düşünebilir” demenin yolu bir sınıfın arayüzleri kalıtım almasıdır.

Bir sınıf C# dilinde yalnızca bir sınıfı kalıtım alırken birden fazla arayüze ait davranışı sergileyebilir. Ayrıca bir sınıf hem başka bir sınıfı kalıtım alıp hem arayüz implementasyonu yapabilir. “İnsan düşünebilen, hareket edebilen bir organizmadır” demenin yolu aşağıdaki kod parçasıdır:

Çok Biçimlilik (Polymorphism)

Çok biçimlilik, kalıtım ile birlikte ortaya çıkan bir özelliktir. Aynı metot imzası ile farklı işlemlerin tetiklenmesi çok biçimliliğin somut halidir. Yukarıda soyut sınıflarla alakalı özelliklerden bahsederken, Surface sınıfı ve alt kırılımı olarak Plateau ve Lowland sınıflarından bahsetmiştik. İki sınıfa ait nesnelere de bir noktanın iz düşümüne sahip olup olmadıkları sorusunu sorabiliriz fakat yürütülen işlemler birbirinden farklı olacaktır. Bir diğer örnek ise GoForward metodunu Human ve Rover için çağırabilmemiz fakat bu iki nesnenin farklı aksiyonlar almasıdır.

Aşağıdaki kod parçasında görüleceği gibi aynı metota çağrı yapıyormuşuz gibi görünüyor fakat iki farklı aksiyon yürütülüyor.

```
foreach (var movable in new IMovable[]{new Rover(), new Human()})
{
  movable.GoForward();
}
```

Çıktı şu şekilde olacaktır:

```
Tekerlek döndürülüyor
Adım atılıyor
```

Kapsülleme (Encapsulation)

Bir nesne içerisinde var olan verilerin dış dünyadan saklanması, erişilmesine kısıt getirilmesi veya üzerinde yapılan değişikliklerin belli kurallara tabi tutulması için yürütülen aksiyonlara verilen isimlerdir.

Rover sınıfına bakacak olursak, bu sınıfa ait bir örnek oluşturulurken nesnenin ilk anda bulunduğu nokta ve baktığı yön bilgisi alınmaktadır. Ardından nesneye GoForward (ileri git) veya Turn (dön) emirleri metotlar aracılığı ile iletilmektedir. Bu metotlar aracılığı ile Rover sınıfının bulunduğu noktanın değeri değiştirilmektedir. Kullanıcı matematik işlemleri ile uğraşmak zorunda kalmaz. Kullanım senaryosu aşağıdaki gibi basit olur, buna karşın karmaşıklık Rover sınıfının içerisinde gizlenir.

Karmaşıklığın gizlenmesi dışında, özelliklere erişimin kısıtlanması da kapsülleme olarak isimlendirilir. Kısıtlama işlemleri için kullanılan anahtar kelimelere erişim belirteci (access modifiers) denir.

- *public*: bu etiket ile etiketlenen metot, alan veya özelliğe herkesin erişebileceğini anlatır.
- *protected*: bu etiket ile etiketlenen metot, alan veya özelliğe sınıfın kendisinin ve bu sınıftan türemiş sınıfların erişebileceğini anlatır.
- *private*: bu etiket ile etiketlenen metot, alan veya özelliğe sınıfın kendisinden başka kimsenin erişemeyeceğini anlatır.

- *internal*: bu etiket ile etiketlenen metot, alan veya özelliğe aynı assembly içerişindeki sınıflar tarafından erişilebileceğini anlatır.

Rover sınıfının içerisinde `public Point CurrentPoint { get; private set; }` şeklinde bir satır görmekteyiz. CurrentPoint özelliğine erişimlerin public get ve private set oluşu şu anlama gelmektedir: bir aracın (rover) o anki konumunu okuyabilirsiniz ama değiştiremezsiniz. Değiştirmek için GoForward metodunu kullanmamız gerekecektir.

```
var rover = new Rover();
rover.Turn(RelativeDirections.Left);
Console.Writeline(rover.CurrentPoint);
Console.Writeline(rover.Facade);
```

Rover sınıfı

More From Medium