YAZILIM
MODELLEME VE TASARIMI

Yrd.Doç.Dr. Feza BUZLUCA
İstanbul Teknik Üniversitesi
Bilgisayar Mühendisliği Bölümü

http://www.akademi.itu.edu.tr/buzluca
http://www.buzluca.info

Konuyu Anlamak Ve Derste Başarılı Olmak İçin

- Bu dersten yararlanabilmeniz için nesneye dayalı programlama (*object oriented programming*) kavramlarını bilmeniz gerekir.
  Özellikle kalıtım (*inheritance*) ve çok şekillilik (*polymorphism*) konularına hakim olmalısınız.
- Eğer nesneye dayalı programlama konusunda bilgi eksikliğiniz varsa Object-oriented programming ders notlarını mutlaka gözden geçiriniz.
  http://www.ninova.itu.edu.tr/tr/dersler/bilgisayar-bilisim-fakultesi/21/blg-252e/ekkaynaklar/
- Ders notları dersin izlenmesini kolaylaştırmak için hazırlanmıştır.
  Ders notları tek başına konuyu tam olarak öğrenmek için **yeterli değildir**.
  Yüksek lisans / doktora öğrencisi olarak kendiniz de bu konuda **araştırma** yapmalı ve başka kaynaklardan da (kitap, bilimsel yayınlar, Internet) yararlanmalısınız.
- Dersin resmi web sitesi Ninova e-öğrenim sisteminde yer almaktadır. Derse kayıtlı olan öğrenciler sisteme (**http://ninova.itu.edu.tr**) İTÜ şifreleri ile girmeliler.
  Dersle ilgili tüm **duyuruları** ve **ödevleri** şifrenizle girerek takip ediniz.
  Sisteme şifresiz giren misafirler sadece ders notlarına erişebilirler.
- Derste edindiğiniz bilgileri uygulamanız için bir dönem projesi verilecektir.
  Dersler ilerledikçe proje üzerinde de çalışmalısınız.
  Projenin teslim tarihi ertelenmeyecektir.

GİRİŞ

Yazılım Geliştirmenin Özellikleri ve Dersin Hedefi:

- Bu derste "endüstriyel boyutlu" yazılımlar ele alınacaktır.
  Bu tür yazılımlar bir çok işlevsel yeteneğe sahiptirler ve bir çok birimden oluşurlar.
- Gerçek dünyanın karmaşıklığı yazılımlara yansıyor. Bu nedenle günümüz yazılımları en az diğer mühendislik ürünleri (bina, köprü, taşıt yapımı) kadar **karmaşıktır**.
- Günümüzün modern yazılımları çok sayıda kişinin yer aldığı takımlar halinde yazılıyor. Yazılımlar bir çok modülden (sınıf, nesne, hizmet) oluşuyor.
  Bu da iyi bir **iletişim** altyapısı ve modüller arası **uyum** gerektirir.
- Yazılımlar sürekli gelişirler (**değişirler!**).
  Bu gelişme ve değişim hem yazılımın geliştirilmesi sürecinde hem de yazılım tamamlandıktan sonra olur.
  Bu nedenle **esneklik** çok önemlidir.

  Anahtar sözcükler: Karmaşıklık, iletişim (işbirliği), uyum, esneklik (değişim)

Yazılım Geliştirmenin Özellikleri ve Dersin Hedefi (devamı):

- Bir programlama dilini iyi bilmek kaliteli bir yazlım geliştirmek için yeterli değildir.
- **Programlama** (kodlama) zevkli bir konudur ama **kaliteli bir yazılım sistemi oluşturmak** daha karmaşık ve zor bir iştir. (*Philippe Kruchten*)
- İyi bir yazılım oluşturabilmek için uygun yazılım geliştirme tekniklerini de bilmek ve uygulamak gerekiyor.

Yazılım Dünyasındaki sorun (The software crisis)

- Yazılımın zamanında tamamlanamaması
- Bütçenin aşılması, bakım maliyetlerinin yüksek olması,
- Bir çok hata çıkması ve bu hataların giderilememesi,
- Yazılımın yeni gereksinimlere göre uyarlanamaması,
- Eski projelerde hazırlanan yazılım modüllerinin yeni projelerde kullanılamaması

Bu **dersin amacı**, yukarıda kısaca sıralanan sorunları gidererek kaliteli yazılımlar geliştirmeyi sağlayan Nesneye Dayalı Çözümleme ve Tasarım (*Object-Oriented Analysis and Design – OOA/D*) yöntemlerini tanıtmaktır.

(1)

(2)

Temel amaç

karmaşıklığı çözebilmek, değişimlerle baş edebilmek, ***esnek*** ve ***tekrar kullanılabilir*** yazılım sistemleri geliştirerek, maliyetleri düşürmek.

(1) http://www.photoeverywhere.co.uk
(2) http://www.cafepress.co.uk/

Dersin Kapsamı:

• Dersin ilk bölümlerinde istekleri ve problemi (gerçeklenecek olan sistemi) **anlamak** için yapılması gereken çözümleme (*analysis*) üzerinde durulacaktır.

Ardından sistemin, işbirliği yapan nesneler halinde nesneye dayalı olarak nasıl **tasarlanacağı** (*design*) açıklanacaktır.

• Analiz ve tasarımların ifade edilmesinde tümleşik modelleme dili (*The Unified Modeling Language* – UML) kullanılacaktır.

• UML'i ayrıntılı olarak öğretmek dersin hedefleri arasında yer almamaktadır, ancak UML diyagramları kullanılırken aynı zamanda bu dilin ders kapsamında kullanılan özellikleri tanıtılacaktır.

• Nesneye dayalı tasarım yapılırken yıllar içinde oluşan deneyimlerin yöntem haline dönüştürülmesi ile oluşturulan tasarım kalıplarından (*design patterns*) yararlanılır.

Bu derste de tasarım aşamasında GRASP kalıpları ve yaygın biçimde kabul gören GoF kalıpları tanıtılacaktır.

• Son olarak yazılım kalitesinin ölçülmesinde ve değerlendirilmesinde kullanılan yazılım metrikleri ele alınacaktır.

• Öğrencilerin bu dersten yararlanabilmeleri için nesneye dayalı programlama (OOP) yöntemini ve bu yöntemi destekleyen dillerden birini (C++, Java, C#) bilmeleri gerekmektedir.

Nesneye Dayalı Analiz ve Tasarım NEDEN Gerekli?
Yazılım Dünyasındaki Problemler:

- Donanım maliyetleri azalırken yazılım maliyetleri artmaktadır.
- Yazılımların boyutları ve karmaşıklığı artmaktadır.
- Yazılımların bakım maliyetleri çok yüksektir.
- Donanım problemleri ile çok az karşılaşılırken yazılım hataları sıklaşmaktadır.

Yazılım Geliştirme Aşamalarının Maliyetleri:

İsteklerin Çözümlenmesi (Requirements):	%3
Tasarım:	%8
Kodlama (Programlama):	%7
Sınama:	%15
Bakım:	**%67 (Maliyeti çok yüksek, neden?)**

Hataların %85'i isteklerin çözümlenmesi ve tasarım aşamalarında oluşmaktadır.

Hataların Giderilme Maliyetleri (Belirlendikleri aşamaya göre):

İsteklerin Çözümlenmesi (Requirements):	1 Birim
Tasarım:	1.5 - 2 Birim
Kodlama (Programlama):	5 – 10 Birim
Sınama:	10 – 15 Birim
Bakım:	15 – 100 Birim

Çözüm:

- Yazılım geliştirme: hem bir bilim dalı hem de bir sanat.
- Kolay ve kesin bir reçete yok. Sezgiler ve deneyim önemli.
- Aşağıdaki unsurlar doğru şekilde kullanıldıklarında işler kolaylaşıyor, başarı olasılığı yükseliyor.

- Uygun yazılım geliştirme süreçleri:
  - Yinelemeli (*iterative*) ve evrimsel (*evolutionary*) yöntemler
  - Tümleştirilmiş geliştirme süreci (*The Unified Process* – UP)
- Programlama ve modelleme yöntemleri
  - Nesneye Dayalı Yöntem
- Yardımcı araçlar
  - UML (*The Unified Modeling Language*)
  - Yazılım Geliştirme Programları
- Nesneye Dayalı Prensipler (Örneğin bağımlılığı sınırlayın)
- Tasarım kalıpları (*Design Patterns*)

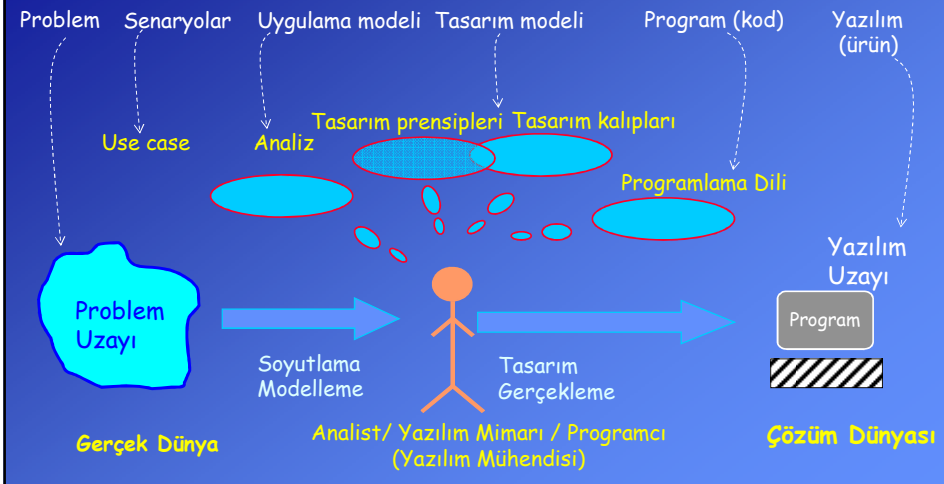
Temel Kavramlar:

Yazılım geliştirme aşamaları:

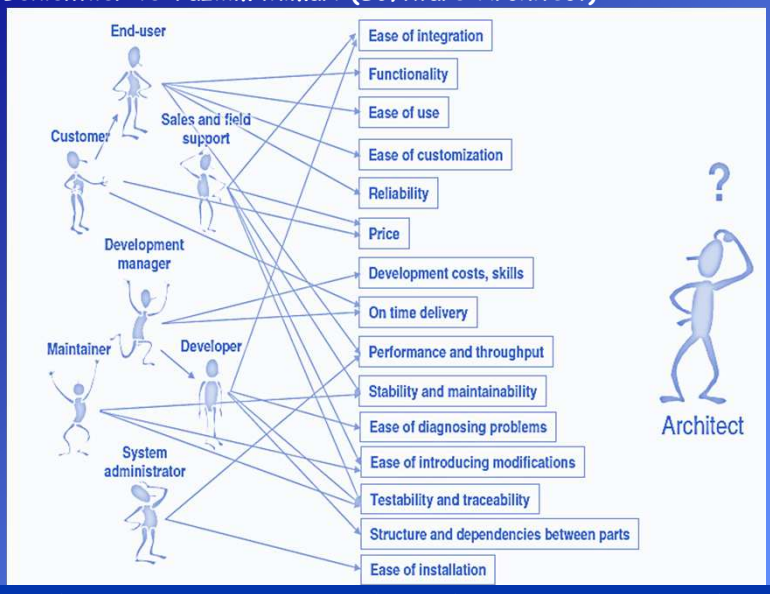
- İstekler (Requirements)
  Müşterilerin isteklerinin anlaşılması. Kullanım durumlarının (*use case*) yazılması.
- Problemin analizi (*Domain analysis*)
  Sistemin (problemin) anlaşılması. Sistem ne yapacak?
- Tasarım (*Design*)
  Sistemin işbirliği yapan nesneler şeklinde tasarlanması.
  Sorumlulukların sınıflara atanması.
- Gerçekleme (*Implementation*)
  Kodlama (*Coding*), programlama (*programming*)
- Değerlendirme (*Evaluation*)
  Sınama (*testing*), performans ölçümü ve değerlendirmesi, bakım

Bu dersin ana konusu sorumlulukların sınıflara uygun şekilde atanmasıdır (tasarım). Ayrıca isteklerin anlaşılması ve analiz konusuna da değinilecektir.

Yazılım Mühendisinin Dünyası



Beklentiler ve Yazılım Mimarı (*Software Architect*)



Kaynak: D. Falessi, G. Cantone, R. Kazman, and P. Kruchten, "Decision-making techniques for software architecture design," *ACM Computing Surveys*, vol. 43, pp. 1-28, Oct. 2011.

Nesneye Dayalı Çözümleme (*Analysis*):

Yazılım mühendislerinin çok farklı alanlarda çalışması gerekebilir.

Bu nedenle sadece yazılım konusunda (*software domain*) bilgili olmaları yetmez çalıştıkları alanı da (*problem domain*) tanımaları gerekir.

Analiz aşamasında problem (uygulama) uzayındaki, yani gerçek dünyadaki sınıflar veya nesneler (kavramlar) belirlenip tanımlanır.

Bu aşamada amaç problemi çözmek değil, **anlamaktır**.

Örnek: Kütüphane otomasyonundaki kavramlar: Kütüphane, Kitap, Üye, Görevli vs.

Nesneye Dayalı Tasarım (*Design*):

Yazılım (çözüm) uzayındaki, yani bilgisayardaki sınıflar (ve nesneler) oluşturulur. Burada sınıfların içerikleri (özellik ve davranış) ve sınıflar arası ilişkiler tam olarak tanımlanır.

Tasarım prensipleri ve kalıpları kullanılarak sorumluluklar uygun sınıflara atanır.

Uygulama uzayı

Kitap
baslik

Uygulama uzayı modeli

Kitap
-baslik: string
+sayfa_git(int)

Yazılım uzayı modeli

```
class Kitap{
private:
  string baslik;
public:
  void sayfa_git (int);
}
```

Basit Bir Örnek:

Ayrıntıya girmeden önce temel kavramlar basit bir örnek üzerinde gösterilecektir.

Zar Oyunu: (*C.Larman'ın kitabından alınmıştır.*)

Oyuncu iki zar atar. Zarların üste gelen yüzeylerindeki sayıların toplamı 7 ise oyuncu kazanır, aksi durumda kaybeder.

1. İsteklerin (*Requirements*) Belirlenmesi, Kullanım Senaryolarının (*Use Case*) Yazılması

İstekleri belirlemek için kullanılan en geçerli yöntem, kullanım senaryoları (*use case*) yöntemidir.

Bu yöntemde tasarımı yapılan sistem ile kullanıcıları arasında gerçekleşebilecek tüm olaylar numaralandırılarak adım adım yazılır.

Örnek:

Ana senaryo:

1. Oyuncu iki zarı yuvarlar.
2. Sistem zarların üstündeki değerleri ve toplamlarını gösterir.
3. Oyun sona erer.

Alternatif akışlar:

2.a. Üste gelen değerlerin toplamı 7'dir. Sistem oyuncuya kazandığını bildirir.

2.b. Üste gelen değerlerin toplamı 7'den farklıdır. Sistem oyuncuya kaybettiğini bildirir.

2. Uygulama Uzayında Modelleme (Analiz)

Uygulamayı (gerçeklenecek olan sistemi) oluşturan kavramlar (sınıflar), bunların özellikleri ve aralarındaki ilişkiler belirlenir.

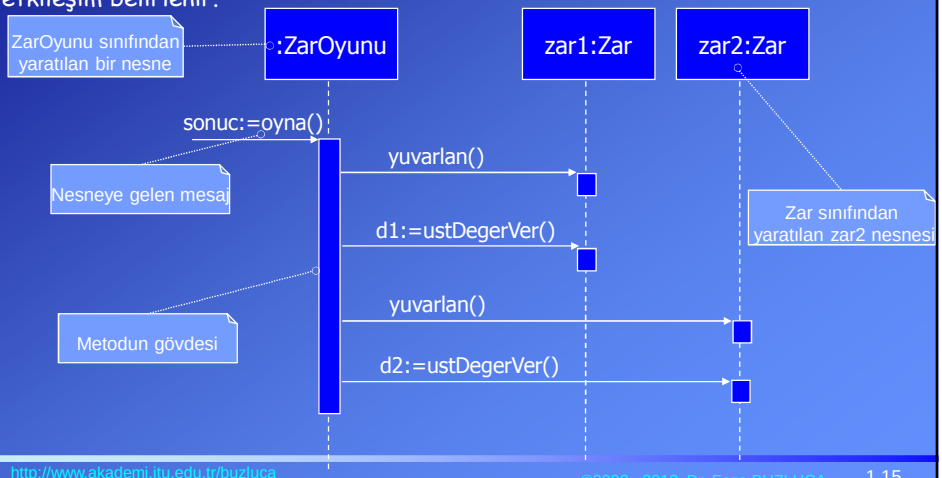
Bu aşamada elde edilen sınıflar gerçek dünyadaki sınıflardır. Oluşturulan modelin amacı problemi çözmek değil, **anlamak**tır.

3. Yazılım (Çözüm) Uzayında Modelleme (Tasarım)

Bu aşamada tasarım yapılarak sistemin sorumlulukları tasarım kalıplarının yardımıyla uygun yazılım sınıflarına atanır.

Tasarım sonucu iki farklı diyagram ile ifade edilir.

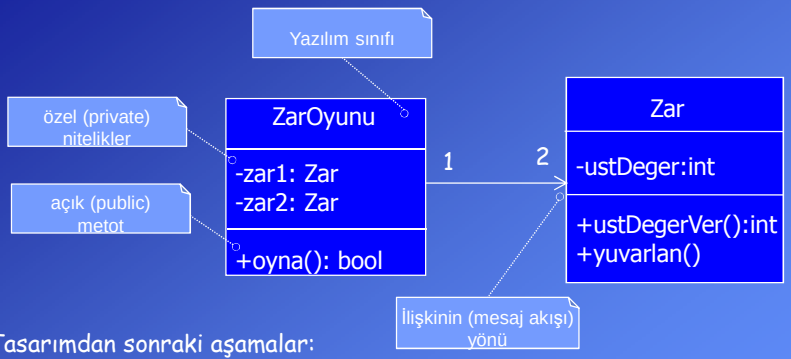
a. Etkileşim Diyagramı: Sınıfların (nesnelerin) davranışları ve aralarındaki etkileşim belirlenir.



b. Sınıf Diyagramı

Sınıfların sahip olacakları üyeler (özellikler ve davranışlar) programda yer alacağı şekilde belirlenir.

Bu aşamadaki model problemin çözümünü oluşturur.



Tasarımdan sonraki aşamalar:

4. Kodlama

5. Sınama

Bu örnek ile kısaca açıklanan konular dersin ilerleyen bölümlerinde ayrıntılı olarak anlatılacaktır.

Bir Yazılımın Kalitesi

- Bir yazılım istenen işi doğru olarak yapmalıdır (*useful*).
- Program kolay kullanılabilir olmalıdır (*usable*).
- Program uygulamanın gerektirdiği kadar hızlı çalışmalıdır.
- Program, sistem kaynaklarını (işlemci, bellek, disk, iletişim ağı vb.) gerektiğinden daha fazla kullanmamalıdır.
- Yazılım sağlam olmalıdır.
- Yazılımı güncellemek kolay olmalıdır.

kullanıcı

- Yazılımın bakımı (değişen isteklere uyum ve hata giderme) kolay olmalıdır.
- Yazılım projesi yeterli bir süre içinde tamamlanmalıdır.
- Yazılımın modülleri yeni projelerde tekrar kullanılabilmeli (*reusable*).
- Yazılımı geliştirme maliyeti düşük tutulabilmelidir.

Yazılım geliştiren

Programlama dilini bilmek yeterli değil. Programlama dilinin desteklediği yöntemleri iyi kullanabilmek gerekir.

"Çekiç sahibi olmak kişiyi mimar yapmaz."

Nesneye Dayalı Programlama Kavramlarını Kısaca Anımsatma

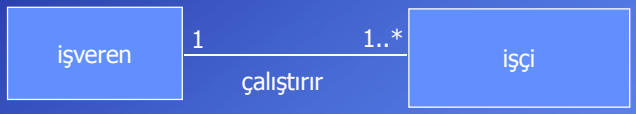
Sınıflar (Nesneler) Arasındaki İlişkiler (*Relations*)

Bir uygulamanın yapısal analizi büyük ölçüde gerekli sınıfların ve sınıflar arasındaki ilişkilerin belirlenmesine dayanır.

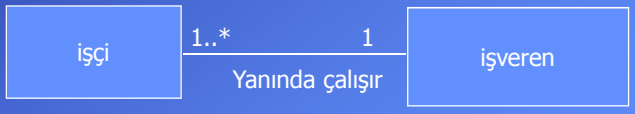
Nesneye dayalı dünyada sınıflar arasında; bağlantı (*association*), sahip olma (*aggregation*), -dan oluşma (*composition*), kalıtım (*inheritance*) gibi ilişkiler olur.

Bağlantı (*Association*)

Sınıflar arasında hizmet alma/vermeye dayalı ilişkidir.



Çoğunlukla çift yönlüdür. Tek yönlü olduğu durumlarda ok kullanılır.



Nesneler arasındaki bağlantıya bağ (link) denir.

Sahip Olma (*has a*) İlişkisi (*Aggregation / Compositon*)

Bir sınıfın üyeleri sadece hazır veri tipleri (char, int, double....) olmak zorunda değildir. Bir sınıfın üyeleri başka sınıftan tanımlanmış nesneler de olabilirler.

Sahip olma ilişkisinin (*has a relation*) iki ayrı türü vardır:

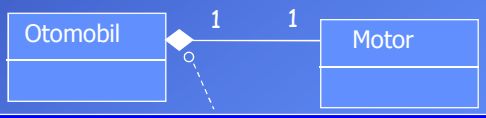
• **Toplama, bir araya getirme (*Aggregation*):** İçerilen nesneler (alt parçalar) kendi başlarına da kullanılırlar. Sadece o nesneye ait parçalar değillerdir.

Örneğin havaalanında uçaklar vardır.

UML sınıf diyagramında bu ilişki ifade edilirken kutucuğun içi boş bırakılır.

• **Meydana gelme (*Composition*):** Alt parçalar o nesneyi meydana getirmek için oluşturulmuşlardır; kendi başlarına kullanılmazlar.

Örneğin otomobilin motoru vardır.



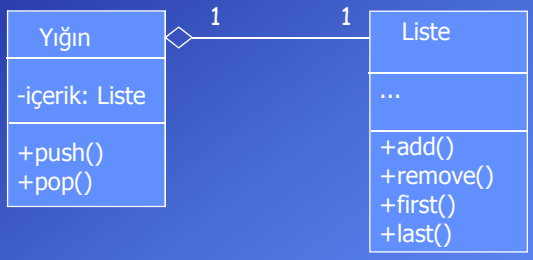
UML sınıf diyagramında bu ilişki ifade edilirken kutucuğun içi doldurulur.

Temsil (*Delegation*)

Nesneye dayalı programlamanın önemli kavramlarından biri de temsildir (*delegation*).

Bir sınıf/nesne bazı hizmetleri yerine getirmek için başka varlıkları (sınıf/nesne) görevlendirebilir. Diğer bir deyişle kendisini temsil etmesini sağlar.

Örnek: Bir yığın sınıfı tasarlanırken bu sınıfın içine daha önceden varolan bir liste sınıfı yerleştirilebilir.



Yığın sınıfı push işlemi için Liste sınıfının add hizmetini kullanacaktır.

Temsil görevli nesne bulununcaya kadar birden fazla aşama sürebilir.

Kalıtım (*Inhertiance*), *Generalization / Specialization*

Varolan genel bir sınıftan daha özel (ek niteliklere sahip) sınıflar türetilebilir.

Bu türetimde üst sınıfın ayrıntılarının bilinmesine ve kaynak kodunun elde edilmesine gerek yoktur.

Yararları:

1. Tekrar kullanılabilirlik (*reusability*).

Sistemin genel kısımları önce kodlanır. Daha özel kısımlar genel kısımlardan türetilir. Ortak özelliklerin yeniden yazılmasına gerek kalmaz.

2. Çok şekillilik (*polymorphism*) ile birlikte kullanıldığında, aynı sınıftan türeyen farklı varlıkların onlara mesaj gönderen nesnelere aynı varlıklar gibi görünmesini sağlar.

Yandaki örnek diyagramda Müdür bir öğretmendir (*is a relation*) .

Müdür öğretmenin tüm özelliklerine sahiptir, ayrıca ek özelliklere sahiptir.

Üst sınıfın (öğretmen) istenen özellikleri alt sınıfta (müdür) değiştirilebilir (*overriding*).

Çok Şekillilik (*Polymorphism*)

Aynı mesaja farklı sınıflardan yaratılmış olan nesneler farklı tepkiler verirler.

Mesajı gönderen taraf bu mesajı hangi sınıftan bir nesneye gönderdiğini bilmek zorunda değildir.

```
class Teacher{                                  // Base class
    string name;
    int numOfStudents;
  public:
    Teacher(const string &, int);               // Constructor of base
     virtual void print() const
       {  cout << "Name: "<< name << endl;
          cout << " Num of Students:"<< numOfStudents << endl;}
};

class Principal : public Teacher{               // Derived class
    string SchoolName;
  public:
    Principal(const string &,  int , const string &);
    void print() const
       {  Teacher::print();
          cout << " Name of School:"<< SchoolName << endl;}
};
```

```
void show (const Teacher * tp)
{
  tp->print();      // hangi print
}

// Test amaçlı main
int main()
{
  Teacher t1("Teacher 1",50);
  Principal p1("Principal 1",40,"School");
  Teacher *ptr;
  char c;
  cout << "Teacher or Principal "; cin >> c;
  if (c=='t') ptr=&t1;
      else    ptr=&p1;
  show(ptr);              // hangi print ?
    :
```

Eğer print fonksiyonu C++ dilinde sanal (*virtual*) olarak tanımlanmazsa show fonksiyonunda her zaman Teacher sınıfındaki fonksiyon çağırılır.

print fonksiyonu sanal olarak tanımlandığında çok şekillilik özelliği kazanır. Bu durumda tp işaretçisi hangi tipte bir nesneye işaret ediyorsa ona ait sınıftaki print fonksiyonu canlanır.

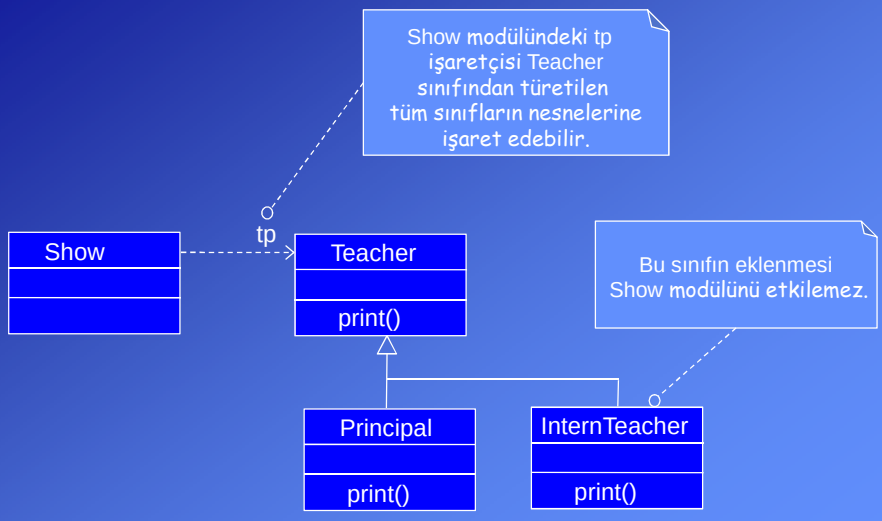
Bu durumda hangi metodun çağırılacağına program çalışırken (*run-time*) karar verilmiş olur.

Çok şekillilik esneklik sağlar.

Yukarıdaki örnekte show fonksiyonu Teacher sınıfından türeyen tüm sınıfların nesneleri üzerinde işlem yapabilir.

Örnek olarak ileride Teacher sınıfından türetilerek stajyer öğretmenleri tanımlamak üzere InternTeacher adlı yeni bir sınıf sisteme katılsa show fonksiyonunda bir değişiklik yapmaya gerek olmaz.

Örnek tasarımın UML diyagramı:



C++ dilinde işaretçiler yerine kullanımı daha basit olan referans tipi de (&) kullanılabilir.

Aşağıdaki örnekte show fonksiyonun giriş parametresi referans tipindedir.

```
// Show is a system that operates on Teachers and Principals
void show (const Teacher & tp)
{
  tp.print();      // which print
}
// Only to test the show function
int main()
{
  Teacher t1("Teacher 1", 50);
  Principal p1("Principal 1", 40, "School");
  char c;
  cout << "Teacher or Principal " ; cin >> c;
  if (c == 't') show(t1);
       else    show(p1);
  .
```

Eğer nesneye dayalı programlama konusunda bilgi eksikliğiniz varsa Object-oriented programming ders notlarını mutlaka gözden geçiriniz.
http://www.ninova.itu.edu.tr/tr/dersler/bilgisayar-bilisim-fakultesi/21/blg-252e/ekkaynaklar/

Modellemenin Önemi ve Yararı:

- **Uygulama (analiz) modeli** gerçek dünyadaki problemi ve üzerinde çalışacağımız sistemi doğru şekilde **anlamamızı** sağlar.
- **Tasarım modeli** sistemin tüm gerekli işlevlerinin (sorumluluklarının) sağlanıp sağlanmadığının görülmesini sağlar.

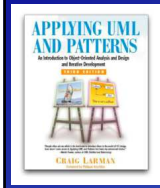
  Bu model ayrıca tasarımımızı güvenlik, esneklik gibi ölçütlere göre sınamamızı ve değerlendirmemizi de sağlar.
- Model tasarım ile ilgili kararlarımızı daha kolay sunmamızı ve açıklamamızı sağlar.

  Bu durum takım içi iletişimi ve çalışmayı kolaylaştırır.
- Modeller gerekli düzeltme ve değişikliklerin yazılım geliştirmenin erken aşamalarında yapılmasını sağlarlar. Bu da maliyeti düşürür.
- Örneğin uçaklar üretilmeden önce tasarımlarını fiberglastan modeli yapılır ve bu model rüzgar tünelinde sınanır.

"Progress is possible only if we train ourselves to think about programs without thinking of them as pieces of executable code."
Edsger W. Djikstra (1930-2002)

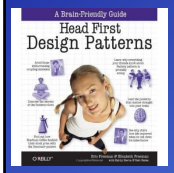
Ana kaynak:

Kaynak Kitaplar:

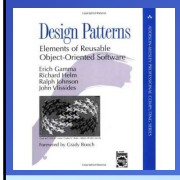


Craig Larman, Applying UML and Patterns , An Introduction to OOA/D and Iterative Development, 3/e, 2005.

Diğer Kaynaklar:



Eric & Elisabeth Freeman: Head First Design Patterns, O'REILLY, 2004.



Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns : Elements of Reusable Object-Oriented Software*, Reading MA, Addison-Wesley, 1995.