NESNE YÖNELİMLİ PROGRAMLAMA

Nesne Yönelimli Programlama Paradigması

Yapısal programlamada talimatlar (statement) art arda koda yazılarak programlama yapılır ve programların neler yaptığı bu talimatlar izlenerek anlaşılabilir. Talimatların zincirin halkaları gibi birbirinin peşi sıra yazılarak yapılan programlamaya paradigması adı verilir. Bu paradigmaya sahip diller, yazılımı yapılacak sürece ilişkin nelerin yapılacağını değil, işin nasıl yapılacağını belirtirler

Emreden programlama (imperative programming) bir örneği olan yapısal programlamada talimatlar (statement) art arda koda yazılarak programlama yapılır. 1980'li yıllara yazılım ihtiyaçları kadar ihtiyaçları yapısal programlama ile çözülmüştür. Kodlar büyüdükçe sorunlar artmış ve *Nesne Yönelimli Programlama İhtiyacı* başlığı altında anlatılan problemler ortaya çıkmıştır. Bu yıllarda Simula ve Smalltalk yaklaşımı yazılımcıların imdadına yetişmiştir.

Bu dillerde nesneler (object) programın temel yapıtaşlarıdır. Nesneler; durum (state) ve davranışlara (behavior) sahiptir. Programlama, nesnelerin birbirlerine ileti göndermesiyle (message-passing) yapılır.

1979 senesinde bir Danimarkalı bilgisayar bilim adamı olan *Bjarne Stroustrup*, sonradan C++ olarak bilinecek olan "C with Classes" üzerinde çalışmaya başladı ve 1985 yılında ilk sürümünü yayınladı. C++ Programlama Dili;

- C Dilinden türetilmiştir.
- Düşük düzey programa yapılabilir
- İcra hızı yüksektir.
- C dilinden daha fazla kütüphaneye sahiptir.
- Nesne yönelimli programlamayı destekler.
- Diğer dillere göre daha etkin hafıza yönetimi sağlar
- Standart Template Library -STL ile jenerik (generic) programlamayı destekler.
- Birçok kavramı bir anda özümsemek gerektiğinden öğrenme eğrisi diktir.

Emreden paradigma (imprative programming) aksine nesnelerin birbirlerine ileti göndermesi bakış açısıyla yapılan programlamaya nesne yönelimli programlama (object oriented programming) paradigması adı verilir.

Sınıf ve Nesne Nasıl Tanımlanır?

Nasıl ki dünyamızda evler bir mimari plan (blueprint) üzerinden inşa ediliyor, nesneler (object) de bir plan üzerinden inşa edilir. Nesnelere ilişkin verilerin tutulduğu durumların (state) ve nesnelerin göstereceği davranışlarının (behavior) tanımlandığı bu plana sınıf (class) adı verilir. Durumlar (state), nesnelere ilişkin verilerin tutulduğu alanlardır (field).

Bir mimari plandan birçok ev yapılabileceği gibi, bir sınıftan birçok nesne imal edilebilir. İmal edilen her nesne (object) sınıfın bir örneğidir (instance). Sınıflar, aşağıdaki şekilde tanımlanır;

```
class sınıf-kimliği {
    veri-tipi alan-değişkeni-kimliği;
    veri-tipi davranış-kimliği();
} [örnek-değişkeni-1][, örnek-değişkeni-2];
```

Sınıflar, temelde yapı (struct) ve birlik (union) gibi tanımlanır ancak yapı gibi üyeleri sadece veriler olmaz, fonksiyonlar da sınıfların üyeleri olabilir. Sınıf üyesi olan fonksiyonlar, örnek değişkenlerin davranışlarını (behavior) gösterdiği yöntemlerdir (method).

Özet olarak sınıflar, kodumuzda imal edilecek nesnelere ilişkin ortak davranış ve özellikleri tasnif eden bir şablon bileşenidir. Aşağıda ortak davranış ve özelliklerin tanımlandığı bir `Ogrenci` sınıfı örneği verilmiştir.

```
#include <iostream>
using namespace std;

class Ogrenci {
public: /* bu sınıftan imal edilecek nesnelerin
    umuma açık (public) davranış ve özellikleri bu kısımda tanımlanır.
    Bir sonraki başlıkta anlatılmıştır.
    */
    unsigned yas; /* "yas"  kimlikli alan(field),
    ogrencinin yaşına ait durumlara(state) ait verileri tutar.
    */
    char cinsiyet; /* "cinsiyet"  kimlikli alan(field),
    ogrencinin cinsiyetine ait durumlara(state) ait verileri tutar.
    */
    void yasiniSoyle() /* yasiniSoyle() yöntemi(method),
    öğrencinin yaşını söyleme davranışını(behavior) tanımlıyor */
    {
        cout << "Yaşım:" << yas << endl;
    }
    void cinsiyetSoyle() /* cinsiyetSoyle() yöntemi(method),
    öğrencinin cinsiyetini söyleme davranışını(behavior) tanımlıyor */
    {
        if (cinsiyet=='E' || cinsiyet=='e')
            cout << "Cinsiyetim: ERKEK" << endl;
        else if (cinsiyet=='K' || cinsiyet=='k')
            cout << "Cinsiyetim: KADIN" << endl;
        else
            cout << "Cinsiyetim: SÖYLEMEK İSTEMİYORUM!" << endl;
    }
} ilhan;  // Ogrenci sınıfından "ilhan" nesnesi imal edildi
int main() {
  ilhan.yas=50; // "ilhan" nesnesinin "yas" özelliği değiştirildi.
  ilhan.yasiniSoyle(); /* "ilhan" nesnesine yasiniSoyle()
  iletisi gönderildi (message passing). */
  ilhan.cinsiyet='E';
  ilhan.cinsiyetSoyle();  /* "ilhan" nesnesine cinsiyetSoyle()
  iletisi gönderildi (message passing).*/

  Ogrenci gullu; // Ogrenci sınıfından "gullu" nesnesi imal edildi
  gullu.yas=30; // "gullu" nesnesinin "yas" özelliği değiştirildi.
  gullu.cinsiyet='K';
  gullu.yasiniSoyle(); /* "gullu" nesnesine yasiniSoyle() iletisi
  gönderildi (message passing). */
  gullu.cinsiyetSoyle();  /* "gullu" nesnesine cinsiyetSoyle()
  iletisi gönderildi (message passing). */
}
/* Program çalıştırıldığında:
Yaşım:50
Cinsiyetim: ERKEK
Yaşım:30
Cinsiyetim: KADIN

...Program finished with exit code 0
*/
```

Program çalıştırıldığında evrensel olarak `ilhan` nesnesi oluşturulacaktır. Ana fonksiyon icra edilmeye başlandığında da `gullu` nesnesi gibi nesne imal edilebilir. Burada alanların yani imal edilen nesnelere ilişkin durumların davranışları etkilediği gözden kaçırılmamalıdır. Bu durumlara ilişkin veriler alanlarda (field) saklanmaktadır.

Sınıf Üyelerine Erişim ve Erişim Değiştiricileri

Sınıftan imal edilen nesnelerin üyelerine başka nesne ve programların nasıl erişeceği erişim değiştiricileri (access modifier) ile belirlenir. Görünürlük (visibility) sıfatları olarak da adlandırılan bu değiştirticiler kısaca sınıf üyelerinin sınıf içinden ve dışından erişimi belirleyen sıfatlardır. Bunlar;

- `public`: İmal edilecek nesnenin diğer nesneler tarafından ulaşılabilen yani umuma açık davranış ve özellikleri sınıf içinde bu belirleyici altında toplanır.
- `private`: İmal edilen nesnelerin mahrem/şahsi/kişisel davranış ve özellikleri sınıf içinde bu belirleyici altında toplanır. Ön tanımlı (default) erişim değiştiricidir. Bir sınıfta hiçbir erişim değiştirici yoksa tanımlanan durum ve davranışlar mahrem (private) olarak kabul edilir.
- `protected`: İmal edilen nesneler ile "bu sınıftan miras alan sınıflardan imal edilecek" nesnelerin ulaşabileceği korumalı (protected) davranış ve özellikleri bu belirleyici altında toplanır. Bu belirleyici ileride Kalıtım başlığında incelenecektir.

Aşağıda erişim belirleyicilerin uygulandığı basit bir `Kisi` sınıfı örnek verilmiştir. Bu sınıftan imal edilen nesnelerin `yas` özelliği diğer tüm nesneler tarafından erişilip değiştirilebilir, ancak `sir` özelliğine ise hiçbir nesne tarafından erişilemez. `maas` özelliğine ise `Kisi` sınıfından türemiş sınıflardan imal edilen nesneler tarafından bu özelliğe erişilebilir. İleride göreceğiz.

```
#include <iostream>
using namespace std;

class Kisi {
public: /* bu erişim değiştirici sonrası tanımlanacak üyeler;
    sınıftan imal edilecek nesnelerin umuma açık (public)
    davranış ve özellikleridir.
    */
    unsigned yas; /* "yas"  kimlikli alan(field),
    ogrencinin yaşına ait durumlara(state) ilişkin verileri tutar.
    imal edilecek nesnelerda umuma açık bir özelliktir.
    */
private:/* bu erişim değiştirici sonrası tanımlanacak üyeler;
    sınıftan imal edilecek nesnelerin mahrem (private)
    davranış ve özellikleridir.
    */

    int sir; /* "sir"  kimlikli alan(field),
    ogrencinin sırrına ait durumlara(state) ilişkin verileri tutar.
    imal edilecek nesnelerde mahrem bir özelliktir.
    diğer nesneler/programlar tarafından bu özelliğe erişilemez.
    */
protected: /* bu erişim değiştirici sonrası tanımlanacak üyeler;
    sınıftan imal edilecek nesneler ile bu sınıftan türetilmiş
    sınıflardan imal edilece nesnelerin erişebileceği korumalı (protected)
    davranış ve özellikleridir.
    */
    float maas;/* "maas"  kimlikli alan(field),
    ogrencinin maaşına ait durumlara(state) ilişkin verileri tutar.
    imal edilecek nesnelerde mahrem bir özelliktir.
    Ogrenci sınıfından türemiş sınıflar (ileride göreceğiz) dışında
    diğer nesneler/programlar tarafından bu özelliğe erişilemez.
    */
};
int main() {
    Kisi ilhan;  //Kisi sınıfından "ilhan" nesnesi imal edildi
    ilhan.yas=50; // "ilhan" nesnesinin "yas" özelliği değiştirildi.
    //ilhan.sir=-1; //HATA: Ana program ilhan nesnesinin sir özelliğine ulaşamaz.
    //ilhan.maas=10000.0; //HATA: Ana program ilhan nesnesinin maas özelliğine ulaşamaz.
}
```

Nesne Yapıcısı

Bir mimari plandan birçok ev yapılabileceği gibi, bir sınıftan birçok nesne imal edilebilir. İmal edilen her nesne (object) sınıfın bir örneğidir (instance). İmal edilecek nesnelerin varsayılan (default) olarak belirlenen durumlarla (state) imal edilmesi gerekiyorsa nesne yapıcısı (constructor) kullanılır.

Yapıcıların görevi bir sınıftan nesneleri, varsayılan durumlara sahip olarak imal etmektir. Nesnelere ait veriler, alanlarda (field) tutulurlar ve nesnenin durumumu (state) bu veriler belirler. Çünkü durumlar nesnenin davranışını etkiler.

Nesne yapıcıları sınıf kimliğiyle aynı olup bir değer geri döndürmeyen fonksiyonlarmış gibi tanımlanır. Amacı kısaca imal edilecek nesnelerin durumlarına ilk değer vermektir.

```
#include <iostream>
using namespace std;
class Ogrenci {
public: /* bu erişim değiştirici sonrası tanımlanacak üyeler;
    sınıftan imal edilecek nesnelerin umuma açık (public)
    davranış ve özellikleridir.
    */
    Ogrenci() { // Ogrenci yapıcısı
        // Yapıcı içerisinde imal edilecek nesnelerine ilk değer verilir.
        yas=6;
        cinsiyet='E';
        // Yapıcı içerisinde return talimatı bulunmaz!
    }
    unsigned yas; /* "yas"  kimlikli alan(field),
    ogrencinin yaşına ait durumlara(state) ait verileri tutar.
    */
    char cinsiyet; /* "cinsiyet"  kimlikli alan(field),
    ogrencinin cinsiyetine ait durumlara(state) ait verileri tutar.
    */
    void yasiniSoyle() /* yasiniSoyle() yöntemi(method),
    öğrencinin yaşını söyleme davranışını(behavior) tanımlıyor */
    {
        cout << "Yaşım:" << yas << endl;
    }
    void cinsiyetSoyle() /* cinsiyetSoyle() yöntemi(method),
    öğrencinin cinsiyetini söyleme davranışını(behavior) tanımlıyor */
    {
        if (cinsiyet=='E' || cinsiyet=='e')
            cout << "Cinsiyetim: ERKEK" << endl;
        else if (cinsiyet=='K' || cinsiyet=='k')
            cout << "Cinsiyetim: KADIN" << endl;
        else
            cout << "Cinsiyetim: SÖYLEMEK İSTEMİYORUM!" << endl;
    }
};
int main() {
    Ogrenci ilhan; /* Ogrenci sınıfından "ilhan" nesnesi; Ogrenci() yapıcısı
    kullanılarak yası 6 ve cinsiyeti E olarak imal edildi. */

    ilhan.yasiniSoyle(); /* "ilhan" nesnesine yasiniSoyle()
    iletisi gönderildi (message passing). */
    ilhan.cinsiyetSoyle();  /* "ilhan" nesnesine cinsiyetSoyle()
    iletisi gönderildi (message passing).*/

    ilhan.yas=50; // "ilhan" nesnesinin "yas" özelliği değiştirildi.
    ilhan.yasiniSoyle(); /* "ilhan" nesnesine yasiniSoyle() iletisi
    gönderildi (message passing). */
}
```

Örnekte `Ogrenci` sınıfından imal edilecek her nesnenin yaşı 6, ve cinsiyeti 'E' olarak üretilecektir. Hiçbir parametresi olamayan bu yapıcıya ön tanımlı yapıcı (default constructor) adı verilir.

Dinamik Nesne İmalatı

Nesneleri çalıştırma anında (run time) da imal edebiliriz. Nesneleri çalışma zamanında yapıcılar kullanarak imal ederiz ve buna dinamik başlatma (dynamic initialization) adı verilir. Yapıcısı çağrılırken veri tutan alanlara (field) ilk değer değerler için mantık içeriyorsa,

```
SınıfKimliği* imal-edilen-nesne-göstericisi = new SınıfKimliği(yapıcı-argümanları);
```

Dinamik olarak `new` işleci (new operator) ile imal edilecek nesneye öbek bellekte (heap segment) yer tahsis edilir. Ardından nesnenin yapıcısı ile veri tutan alanlarına (field) ilk değer değerler için mantık çalıştırılır ve nesne göstericisi geri döner.

Nesne göstericileri üzerinden sınıf üyelerine dolaylı işleç (indirection operator) yani (`->`) ile erişilir. Bu nokta işleci ile aynı önceliklidir.

Dinamik olarak imal edilmiş nesne kullanıldıktan sonra `delete` işleci (delete operator) ile bellekten kaldırılabilir. Çeşitli yapıcılara sahip dinamik olarak nesne imal edilen bir kod örneği aşağıda verilmiştir;

```
#include <iostream>
using namespace std;
class Karmasik {
public:
    float gercek, sanal;
    Karmasik() // varsayılan yapıcı (default constructor)
    {
        gercek = 0.0;
        sanal = 0.0;
        cout<< "Varsayılan (default) yapıcı çağrıldı" << endl;
    }
    Karmasik(float pGercek, float pSanal): gercek(pGercek),sanal(pSanal) {
        // parametreli yapıcı
        cout<< "Parametreli yapıcı çağrıldı" << endl;
    }
    Karmasik(float pGercek): Karmasik(pGercek,0.0) {
        //delegating constructor
        cout<< "Öncesinde parametreli yapıcı çağrılacak -> "
            << "Tek parametreli yapıcı çağrıldı" << endl;
    }
    void yaz() { //yaz yöntemi
        cout<< gercek << "+" << sanal << +"i" << endl;
    }
};
int main(void) {
    Karmasik* pc1=new Karmasik(); /* hafıza tahsisi: allocate memory
     Bu durumda faıza ayrıldıktan sonra alanlara ilk değerler varsayılan yapıcı
     (default constructor) ile veriliyor
     */
    pc1->yaz();
    Karmasik* pc2=new Karmasik(2.0,3.0); /* hafıza tahsisi: allocate memory
     Bu durumda hafıza ayrıldıktan sonra alanlara ilk değerler parametreli
     yapıcı (default constructor) ile veriliyor */
    pc2->yaz();
    Karmasik* pc3=new Karmasik(4.0); /* hafıza tahsisi: allocate memory
     Bu durumda faıza ayrıldıktan sonra alanlara ilk değerler devredilen yapıcı
     (delegating constructor) ile veriliyor
    */
    pc3->yaz();
    delete pc3; // hafızayı serbest bırakma:deallocate memory
    delete pc2; // hafızayı serbest bırakma:deallocate memory
```

```
    delete pc1; // hafızayı serbest bırakma:deallocate memory
}
/* Program Çıkısı:
Varsayılan (default) yapıcı çağrıldı
0+0i
Parametreli yapıcı çağrıldı
2+3i
Parametreli yapıcı çağrıldı
Öncesinde parametreli yapıcı çağrılacak -> Tek parametreli yapıcı çağrıldı
4+0i

...Program finished with exit code 0
*/
```

Soyut, Somut ve Bilgi Gizleme

Soyut (abstract) kavramı kelime itibariyle detay içermeyen, özet anlamına gelir. Somut (concrete) ise en ince detayına kadar bilinen, elle tutulur anlamına gelir. Bir şey hakkında soyutlama yapıldığında, o şeyin detayıyla ilgili olan tüm bilgiler gizlenir (information hiding).

Bir sınıftan imal edilen nesneler tüm durum (state) veya davranışlarını (behavior) diğer nesnelere şeffaf olarak gösterip diğer nesnelerin değiştirmesine izin verilmemesi gerekir. Bu durumda nesnenin durum ve davranışları tutarsız hale gelir. Bu nedenle bir sınıftan imal edilen nesnelerin durum ve davranışlarının diğer nesnelere veya dış dünyaya kontrollü bir şekilde açılması gerekir.

Soyutlama (abstraction), kullanıcıya gerekli bilgileri gösterme ve kullanıcıya göstermek istemediği veya belirli bir kullanıcıyla ilgisi olmayan ayrıntıları gizleme işlemidir. Bir sınıfta soyutlama iki türlü yapılır;

- Veri soyutlaması (data abstraction): nesnenin durumları üzerinden yapılan soyutlama.
- Kontrol soyutlaması (kontrol abstraction): nesnenin davranışları üzerinde yapılan soyutlama.

Sarmalama

Yukarıda verilen `Ogrenci` örneğinden devam edecek olursak imal edilecek nesnenin `yas` durumuna bir başka nesne veya program `-1` gibi bir değer atayabilir. Bu durumda `yasiniSoyle()` davranışında istenmeyen bir çıktı verilmesine sebep olur. Bunu gidermek için veri soyutlaması yapılması gerekir. Bu durumda yas alanına bir değer koymayı ve yas alanından bir değer almayı kontrol altına almalıyız. Benzer durum `cinsiyet` alanı için de geçerlidir.

Veri soyutlamasını yapmak için `yas` ve `cinsiyet` alanlarını mahrem (private) olarak belirleyip bu alanlara değer atama ve değer okuma işlevlerini yerine getirecek `get` ve `set` yöntemlerini umuma açık olarak tanımlamalıyız;

```
#include <iostream>
using namespace std;

class Ogrenci {
private: /* bu erişim değiştirici sonrası tanımlanacak üyeler;
    sınıftan imal edilecek nesnelerin mahrem (private)
    davranış ve özellikleridir.
    */
    unsigned yas; /* "yas"  kimlikli alan(field),
    ogrencinin yaşına ait durumlara(state) ait verileri tutar.
    */
    char cinsiyet; /* "cinsiyet"  kimlikli alan(field),
     ogrencinin cinsiyetine ait durumlara(state) ait verileri tutar.
    */
public: /* bu erişim değiştirici sonrası tanımlanacak üyeler;
     sınıftan imal edilecek nesnelerin umuma açık (public)
```

```
     davranış ve özellikleridir.
    */
    Ogrenci() { // Ogrenci yapıcısı
        // Yapıcı içarisinde imal edilecek nesnelerine ilk değer verilir.
        yas=6;
        cinsiyet='E';
        // Yapıcı içerisinde return talimatı bulunmaz!
    }
    void setYas(unsigned yeniYas) {
        if (yeniYas <3)
            cout << "Öğrenci 3 Yaşından Küçük Olamaz." << endl;
        else
            yas=yeniYas;
    }
    unsigned getYas() {
        return yas;
    }
    void setCinsiyet(char yeniCinsiyet) {
        if (yeniCinsiyet == 'E' ||
            yeniCinsiyet == 'E' ||
            yeniCinsiyet == 'K' ||
            yeniCinsiyet == 'k' ||
            yeniCinsiyet == 'B' ||
            yeniCinsiyet == 'b')
            cinsiyet=yeniCinsiyet;
        else
            cout << "Cinsiyet E,e,K,k,B,b Dışında Bir Değer Olamaz." << endl;
    }
    char getCinsiyet() {
        return cinsiyet;
    }
    void yasiniSoyle() /* yasiniSoyle() yöntemi(method),
    öğrencinin yaşını söyleme davranışını(behavior) tanımlıyor */
    {
        cout << "Yaşım:" << yas << endl;
    }
    void cinsiyetSoyle() /* cinsiyetSoyle() yöntemi(method),
    öğrencinin cinsiyetini söyleme davranışını(behavior) tanımlıyor */
    {
        if (cinsiyet=='E' || cinsiyet=='e')
            cout << "Cinsiyetim: ERKEK" << endl;
        else if (cinsiyet=='K' || cinsiyet=='k')
            cout << "Cinsiyetim: KADIN" << endl;
        else
            cout << "Cinsiyetim: SÖYLEMEK İSTEMİYORUM!" << endl;
    }
};
int main() {
    Ogrenci ilhan; /* Ogrenci sınıfından "ilhan" nesnesi;
    Ogrenci() yapıcısı kullanılarak yası 6 ve cinsiyeti E olarak imal edildi. */
    ilhan.yasiniSoyle(); /* "ilhan" nesnesine yasiniSoyle()
    iletisi gönderildi (message passing). */
    ilhan.cinsiyetSoyle();  /* "ilhan" nesnesine cinsiyetSoyle()
    iletisi gönderildi (message passing).*/

    ilhan.setYas(0); // "ilhan" nesnesinin "yas" özelliği değiştirildi.
    ilhan.yasiniSoyle(); /* "ilhan" nesnesine yasiniSoyle() iletisi
    gönderildi (message passing). */
    ilhan.setCinsiyet('H');
    ilhan.cinsiyetSoyle();
}
```

İşte bir durumun (state) veri soyutlaması yapılmış haline özellik (property) adı veririz. Çünkü artık duruma dışarıdan doğrudan müdahale edilemez. Nesnenin durum üzerinde kontrol hakları vardır.

Benzer şekilde davranışlar (behavior) da nesne dışına kontrollü olarak açılabilir. Bu durumda da kontrol soyutlaması (control abstraction) yapılır. Aşağıda faktöriyel hesaplayan bir hesap makinesinin ekranı yenileme mahrem davranışı buna örnek verilebilir;

```
#include <iostream>
using namespace std;
class HesapMakinesi {
private:
    int hesaplanan;
    void ekraniYenile() {
        cout << "Hesaplanan:" << hesaplanan << endl;
    }
public:
    HesapMakinesi() { //Yapıcı
        hesaplanan=0;
        ekraniYenile();
    }
    void setHesapMakinesi(int pSayi) {
        hesaplanan=pSayi;
        ekraniYenile();
    }
    int getHesapMakinesi() {
        return hesaplanan;
    }
    void makineyiSifirla() {
        hesaplanan=0;
        ekraniYenile();
    }
    void FaktoriyelHesapla() {
        unsigned temp=1;
        for (int sayac = hesaplanan; sayac > 0; sayac--)
            temp =  temp*sayac;
        hesaplanan=temp;
        ekraniYenile();
    }
};
int main ()
{
  HesapMakinesi hesaplayici; // hesaplayici nesnesi imal edildi
  hesaplayici.setHesapMakinesi(4);
  hesaplayici.FaktoriyelHesapla();
  hesaplayici.setHesapMakinesi(6);
  hesaplayici.FaktoriyelHesapla();
  hesaplayici.makineyiSifirla();
}
```

Bir sınıf üzerinde veri soyutlaması (data abstraction) veriler üzerine kılıf çekilir, hem de kontrol soyutlaması (control abstraction) ile davranışların üzerine bir kılıf çekilir. Bir sınıf üzerinde her iki işlemin birlikte yapılasına sarmalama (encapsulation) adı verilir. Soyutlamaların gerekli olup olmadığına programcı karar verir.

Sarmalama işlemi örneklerde görüldüğü üzere erişim değiştiricilerle (access modifier) yapılır. Böylece imal edilen nesnelerin verileri ve davranışları güvenli bir şekilde dış dünyaya açılmış olur. Böylece, sınıf ve nesne gibi bileşenlerin içeriğini bilmeden soyut (abstract) olarak kullanmak mümkün olmaktadır. Programcı tasarladığı sınıfın soyut olarak katlanılacağını düşünerek sınıfları tasarlamalıdır.

Özetle sarmalama (encapsulation) tekniğini uygulamak, kullanımı kolay sınıf (class), bileşen (component) , kütüphane (library) ve çerçeve yapılar (framework) elde etmemizi sağlar;

- n adet veri (data) ve n adet davranış (behavior) sarmalama işlemine tabi tutulursa sınıf,
- n adet sınıf sarmalama işlemine tabi tutulursa bileşen,
- n adet bileşen sarmalama işlemine tabi tutulursa kütüphane,
- n adet kütüphane sarmalama işlemine tabi tutulursa çerçeve yapılar oluşur.

Böylece bize projemizde proje süresini ve maliyetini kısaltma üstünlüğünü verir.

Kalıtım

Öğrenci, öğretmen ve müdürün olduğu bir sistemi örnek alarak ele alalım. Bunlardan her birinden nesne imal etmek için her birine ayrı ayrı sınıf tanımlamak gerekir.

```
class Ogrenci {
private:
    unsigned yas;
    char cinsiyet;
    unsigned ogrenciNo;
public:
    Ogrenci() {
        yas=18;
        cinsiyet='E';
    }
    void setYas(unsigned yeniYas) {
        if (yeniYas <18)
            cout << "Yaş 18 den Küçük Olamaz." << endl;
        else
            yas=yeniYas;
    }
    unsigned getYas() {
        return yas;
    }
    void setCinsiyet(char yeniCinsiyet) {
        if (yeniCinsiyet == 'E' ||
            yeniCinsiyet == 'E' ||
            yeniCinsiyet == 'K' ||
            yeniCinsiyet == 'k' ||
            yeniCinsiyet == 'B' ||
            yeniCinsiyet == 'b')
            cinsiyet=yeniCinsiyet;
        else
            cout << "Cinsiyet E,e,K,k,B,b Dışında Bir Değer Olamaz." << endl;
    }
    char getCinsiyet() {
        return cinsiyet;
    }
    void yasiniSoyle()
    {
        cout << "Yaşım:" << yas << endl;
    }
    void cinsiyetSoyle()
    {
        if (cinsiyet=='E' || cinsiyet=='e')
            cout << "Cinsiyetim: ERKEK" << endl;
        else if (cinsiyet=='K' || cinsiyet=='k')
            cout << "Cinsiyetim: KADIN" << endl;
        else
            cout << "Cinsiyetim: SÖYLEMEK İSTEMİYORUM!" << endl;
    }
    void dersCalis() {
        cout << "Ders Çalışıyorum..." << endl;
    }
```

```
};
```

Öğretmen için ayrı bir sınıf;

```
class Ogretmen {
private:
    unsigned yas;
    char cinsiyet;
    unsigned verdigiDersKodu;
public:
    Ogretmen() {
        yas=18;
        cinsiyet='E';
    }
    void setYas(unsigned yeniYas) {
        if (yeniYas <18)
            cout << "Yaş 18 den Küçük Olamaz." << endl;
        else
            yas=yeniYas;
    }
    unsigned getYas() {
        return yas;
    }
    void setCinsiyet(char yeniCinsiyet) {
        if (yeniCinsiyet == 'E' ||
            yeniCinsiyet == 'E' ||
            yeniCinsiyet == 'K' ||
            yeniCinsiyet == 'k' ||
            yeniCinsiyet == 'B' ||
            yeniCinsiyet == 'b')
            cinsiyet=yeniCinsiyet;
        else
            cout << "Cinsiyet E,e,K,k,B,b Dışında Bir Değer Olamaz." << endl;
    }
    char getCinsiyet() {
        return cinsiyet;
    }
    void yasiniSoyle()
    {
        cout << "Yaşım:" << yas << endl;
    }
    void cinsiyetSoyle()
    {
        if (cinsiyet=='E' || cinsiyet=='e')
            cout << "Cinsiyetim: ERKEK" << endl;
        else if (cinsiyet=='K' || cinsiyet=='k')
            cout << "Cinsiyetim: KADIN" << endl;
        else
            cout << "Cinsiyetim: SÖYLEMEK İSTEMİYORUM!" << endl;
    }
    void dersVer() {
        cout << "Ders Veriyorum..." << endl;
    }
};
```

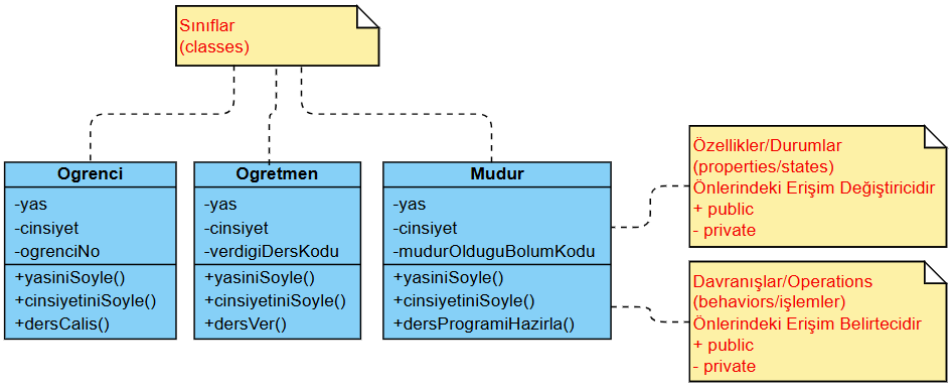
Müdür için ayrı bir sınıf;

```
class Mudur {
private:
    unsigned yas;
    char cinsiyet;
    unsigned muduruOlduguBolumKodu;
```

```
public:
    Mudur() {
        yas=18;
        cinsiyet='E';
    }
    void setYas(unsigned yeniYas) {
        if (yeniYas <18)
            cout << "Yaş 18 den Küçük Olamaz." << endl;
        else
            yas=yeniYas;
    }
    unsigned getYas() {
        return yas;
    }
    void setCinsiyet(char yeniCinsiyet) {
        if (yeniCinsiyet == 'E' ||
            yeniCinsiyet == 'E' ||
            yeniCinsiyet == 'K' ||
            yeniCinsiyet == 'k' ||
            yeniCinsiyet == 'B' ||
            yeniCinsiyet == 'b')
            cinsiyet=yeniCinsiyet;
        else
            cout << "Cinsiyet E,e,K,k,B,b Dışında Bir Değer Olamaz." << endl;
    }
    char getCinsiyet() {
        return cinsiyet;
    }
    void yasiniSoyle()
    {
        cout << "Yaşım:" << yas << endl;
    }
    void cinsiyetSoyle()
    {
        if (cinsiyet=='E' || cinsiyet=='e')
            cout << "Cinsiyetim: ERKEK" << endl;
        else if (cinsiyet=='K' || cinsiyet=='k')
            cout << "Cinsiyetim: KADIN" << endl;
        else
            cout << "Cinsiyetim: SÖYLEMEK İSTEMİYORUM!" << endl;
    }
    void dersProgramiHazırla() {
        cout << "Ders Programı Hazırlıyorum..." << endl;
    }
};
```

Yazılım analizi yapılırken takım içerisinde iletişimi kolaylaştırmak için aşağıdaki sınıf diyagramları (class diagram) kullanılır. Bunlar Unified Modelling Language-UML dilinin bir parçasıdır.

Bu sınıfların her birinde `yas` ve `cinsiyet` gibi ortak özellikler (property) veya `yasiniSoyle()` ve `cinsiyetSoyle()` gibi ortak davranışlar (behavior) bulunur. Her sınıf için bunlar tekrar tekrar tanımlanırlar. Hâlbuki bu gibi ortak özellikler ve davranışlar tek bir sınıf altında toplanabilir. Bütün bu ortak özellik ve davranışlar tek bir sınıfta toplandığı zaman daha yönetilebilir bir kod elde edilir.



Şekil 22. Öğrenci, Öğretmen ve Müdür UML Sınıf Diyagramları

Ortak özellik ve davranışların bir sınıfta toplanarak bu sınıftan miras alınmasına kalıtım (inheritance) adı verilir. Yukarıdaki örnekte ortak özellik ve davranışlar `Kisi` sınıfına toplanarak aşağıdaki şekilde kod yeniden düzenlenebilir;

Aşağıda verilen ve Kişi olarak tanımlanan genel sınıf (general class), taban sınıf (base class) veya ebeveyn sınıf (parent class) olarak adlandırılır. Daha sonra tanımlanacak Öğrenci, Öğretmen ve Müdür sınıflarının ortak özellik ve davranışlarını barındırır.

```
class Kisi {
private:
    unsigned yas;
    char cinsiyet;
public:
    Kisi() {
        yas=18;
        cinsiyet='E';
    }
    void setYas(unsigned yeniYas) {
        if (yeniYas <18)
            cout << "Yaş 18 den Küçük Olamaz." << endl;
        else
            yas=yeniYas;
    }
    unsigned getYas() {
        return yas;
    }
    void setCinsiyet(char yeniCinsiyet) {
        if (yeniCinsiyet == 'E' ||
            yeniCinsiyet == 'E' ||
            yeniCinsiyet == 'K' ||
            yeniCinsiyet == 'k' ||
            yeniCinsiyet == 'B' ||
            yeniCinsiyet == 'b')
            cinsiyet=yeniCinsiyet;
        else
            cout << "Cinsiyet E,e,K,k,B,b Dışında Bir Değer Olamaz." << endl;
    }
    char getCinsiyet() {
        return cinsiyet;
    }
    void yasiniSoyle()
    {
        cout << "Yaşım:" << yas << endl;
    }
    void cinsiyetSoyle()
    {
        if (cinsiyet=='E' || cinsiyet=='e')
```

```
            cout << "Cinsiyetim: ERKEK" << endl;
        else if (cinsiyet=='K' || cinsiyet=='k')
            cout << "Cinsiyetim: KADIN" << endl;
        else
            cout << "Cinsiyetim: SÖYLEMEK İSTEMİYORUM!" << endl;
    }
};
```

Aşağıda taban sınıftan türemiş (derived) Öğrenci, Öğretmen ve Müdür sınıfları verilmiştir. Bu sınıflar genel sınıf olan Kişi sınıfının yanında kendine özel özellik ve davranışlara da sahiptir. Özel sınıf (private class), türemiş sınıf (derived class) veya çocuk sınıf (child class) olarak adlandırılan bu sınıflar, mirasçı olup, kendilerinden imal edilmiş nesneler; hem taban sınıfın özellik ve davranışlarını, hem de kendi özellik ve davranışlarını sergilerler.

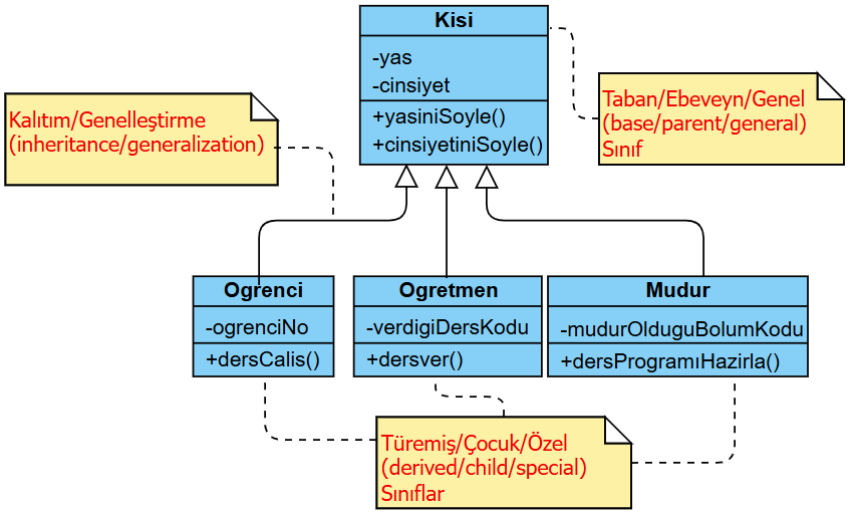
```
class Ogrenci: public Kisi {
private:
    unsigned ogrenciNo;
public:
    Ogrenci() {
    }
    void dersCalis() {
        cout << "Ders Çalışıyorum..." << endl;
    }
};
class Ogretmen:public Kisi {
private:
    unsigned verdigiDersKodu;
public:
    Ogretmen() {
    }
    void dersVer() {
        cout << "Ders Veriyorum..." << endl;
    }
};
class Mudur:public Kisi {
private:
    unsigned muduruOlduguBolumKodu;
public:
    Mudur() {
    }
    void dersProgramiHazırla() {
        cout << "Ders Programı Hazırlıyorum..." << endl;
    }
};
```

Programlamanın yapıldığı ana fonksiyonda bu üç sınıftan da nesne imal edilmiş ve çeşitli davranışlar göstermesi için kendilerine ileti gönderilmiştir (message-passing).

```
int main() {
    Ogrenci ilhan;
    Ogretmen mehmet;
    Mudur abdullah;

    ilhan.yasiniSoyle();
    ilhan.cinsiyetSoyle();
    ilhan.dersCalis();

    mehmet.yasiniSoyle();
    mehmet.cinsiyetSoyle();
    mehmet.dersVer();

    abdullah.yasiniSoyle();
```

```
    abdullah.cinsiyetSoyle();
    abdullah.dersProgramiHazırla();
}
```

Yeni haliyle sınıf diyagramları aşağıda verilmiştir;



Şekil 23. Kişi Taban Sınıfı Eklenmiş Kalıtım UML Sınıf Diyagramı

Yukarıdaki örnekler incelendiğinde türemiş sınıfların taban sınıftan miras alırken üç farklı şekilde alabilirler;

- Umuma açık (public) kalıtım: Taban sınıfın umuma açık üyelerini türetilmiş sınıfta umuma açık hale getirir ve taban sınıfın korumalı (protected) üyeleri de türetilmiş sınıfta korumalı olarak kalır.
- Mahrem (private) kalıtım: Taban sınıfın umuma açık ve korumalı üyelerini türetilmiş sınıfta mahrem hale getirir.
- Korumalı (protected) kalıtım: Taban sınıfın umuma açık ve korumalı üyelerini türetilmiş sınıfta korumalı hale getirir.

Bunu aşağıdaki kod örneğiyle anlayabiliriz;

```
class Base { //taban sınıf
public:
    /* burada tanımlanan üyeler umuma açıktır.
       Yani bu üyelere sınıf dışı ve içinden erişilebilir.
    */
    int x;
protected:
    /* burada tanımlı üyelere türeyen sınıftan türemiş sınıflardan erişilir.
       Bir başka deyişle miras alan sınıflar tarafından burada tanımlanan
       üyelere erişilir. */
    int y;
private:
    /* burada tanımlı üyelere yalnızca bu sınıftan imal edilen nesneler
       erişir. Bir başka deyişle yalnızca bu sınıf erişebilir.*/
    int z;
};
class PublicDerived: public Base { //public inheritance
  // Bu sınıfta x alanı(field), public olmuştur.
  // Bu sınıfta y alanı(field), protected olmuştur.
  // Bu sınıfta z alanı(field) ERİŞİLEMEZ olmuştur. -> Public-Derived
};
class PrivateDerived: private Base { //private inheritance
  // Bu sınıfta x alanı(field), private olmuştur.
  // Bu sınıfta y alanı(field), private olmuştur.
  // Bu sınıfta z alanı(field) ERİŞİLEMEZ olmuştur. -> Private-Derived
};
```

```
class ProtectedDerived: protected Base { //protected inheritance
  // Bu sınıfta x alanı(field), protected olmuştur.
  // Bu sınıfta y alanı(field), protected olmuştur.
  // Bu sınıfta z alanı(field) ERİŞİLEMEZ olmuştur. -> Protected-Derived
};
```

Değişim yönetimini kolayca yapma üstünlüğünü veren kalıtımın (inheritance) iki önemli özelliği vardır; Paylaşılan bir ortak kod (shared code) vardır ve paylaşılan kodda yapılan değişiklik anında alt sınıflara yansır (snap change).

Örneğimizden gidecek olursak, Kişi sınıfında yapılacak bir özellik veya davranış eklemesi anında bu sınıftan türemiş sınıflarda yapılmış olur.

Korumalı (protected) olarak taban sınıftan alınan durum ve davranışlara ait bir örnek aşağıda verilmiştir;

```
#include <iostream>
using namespace std;
class Kisi {
protected: //Burada tanımlana alan ve davranışlara türemiş sınıflardan da erişilebilir.
    unsigned yas;
    char cinsiyet;
public:
    Kisi() {
        yas=18;
        cinsiyet='E';
    }
    void setYas(unsigned yeniYas) {
        if (yeniYas <18)
            cout << "Yaş 18 den Küçük Olamaz." << endl;
        else
            yas=yeniYas;
    }
    unsigned getYas() {
        return yas;
    }
    void setCinsiyet(char yeniCinsiyet) {
        if (yeniCinsiyet == 'E' ||
            yeniCinsiyet == 'E' ||
            yeniCinsiyet == 'K' ||
            yeniCinsiyet == 'k' ||
            yeniCinsiyet == 'B' ||
            yeniCinsiyet == 'b')
            cinsiyet=yeniCinsiyet;
        else
            cout << "Cinsiyet E,e,K,k,B,b Dışında Bir Değer Olamaz." << endl;
    }
    char getCinsiyet() {
        return cinsiyet;
    }
    void yasiniSoyle()
    {
        cout << "Yaşım:" << yas << endl;
    }
    void cinsiyetSoyle()
    {
        if (cinsiyet=='E' || cinsiyet=='e')
            cout << "Cinsiyetim: ERKEK" << endl;
        else if (cinsiyet=='K' || cinsiyet=='k')
            cout << "Cinsiyetim: KADIN" << endl;
        else
            cout << "Cinsiyetim: SÖYLEMEK İSTEMİYORUM!" << endl;
```

```
    }
};
class Ogrenci: public Kisi {
private:
    unsigned ogrenciNo;
public:
    Ogrenci() {
        yas=35;  // Artık yas alanına bu türemiş sınıftan da erişilebiliyor.
        cinsiyet='K'; // Artık cinsiyet alanına bu türemiş sınıftan da erişilebiliyor.
    }
    void dersCalis() {
        cout << "Ders Çalışıyorum..." << endl;
    }
};
int main() {
    Ogrenci ayse;

    ayse.yasiniSoyle();
    ayse.cinsiyetSoyle();
    ayse.dersCalis();
}
```

Kalıtımda üyelere erişim, özet olarak aşağıdaki tabloda verilmiştir;

Üyelere Erişim	public	protected	private
Aynı Sınıf İçinde	var	var	var
Türemiş Sınıfta	var	var	yok
Sınıf Dışından	var	yok	yok

Tablo 19. Kalıtımda Sınıf Üyelerine erişim.

Türetilmiş bir sınıf, aşağıdaki istisnalar dışında tüm taban sınıf yöntemlerini miras alır;

1. Taban sınıfın yapıcıları, yıkıcıları (destructor) ve kopya yapıcıları.
2. Taban sınıfın aşırı yüklenmiş işleçleri.
3. Taban sınıfın arkadaş (friend) fonksiyonları.

Nesne Yönelimli Programlamada sınıflar, birden fazla taban sınıftan miras alamayacak şekilde tasarlanır. Mantık olarak bir ev, birden fazla mimari plandan hazırlanmayacağından, çoklu kalıtım (multiple inheritance) sınıf tasarımında kullanılmaz.

C++ dilinde bir türemiş sınıfın birden fazla ebeveyn sınıftan miras alması engellenmemiştir. İleride göreceğimiz ara yüzler (interface) de C++ dilinde sınıf olarak kodlandığından sanki çoklu kalıtım varmış gibi algılanmaktadır.

Saf nesne yönelimli diller olan Java ve .Net dillerinde ara yüzler ayrı bir anahtar kelimeyle tanımlandığından doğal olarak birden çok sınıftan miras alma engellenmiştir.

Aşırı Yükleme

C dilinden farklı olarak C++ dilinde aşırı yükleme (overloading); aynı davranışın farklı yöntemlerle (method) yeniden tanımlanmasıdır.

Fonksiyonlarda (function) aşırı yükleme, farklı parametrelerle fonksiyonun yeniden tanımlanmasıdır. Aşırı yüklemede fonksiyonun kimliği ve geri dönüş tipi değişmez. Farklı argümanlar ile fonksiyon gövdesi ile yeniden tanımlanır.

Aşağıda bir fonksiyonun aşırı yüklemesine bir örnek verilmiştir.

```
#include <iostream>
using namespace std;
void topla(int a, int b) {  //İlk tanımlanan topla fonksiyonu
  cout << "sum = " << (a + b);
}
```

```
void topla(double a, double b) {  // Aşırı yüklenmiş topla fonksiyonu
    cout << endl << "sum = " << (a + b);
}
int main() {
    topla(1, 2);
    topla(2.5, 5.5);
}
```

Fonksiyonlarda varsayılan argümanlar kullanıldığında aşırı yükleme tanımlanmasına çok dikkat edilmelidir;

```
void topla(int a = 10, int b = 20); // geçerli
void topla(int a = 22, int b = 2); // HATA: fonksiyon imzası (signature) aynı!
/* Yani parametreler veri tipleriyle aynı sırada tanımlı. Sadece varsayılan argümanlar
değişik.
*/
void f(int a); // HATA: varsayılan argümanlı fonksiyon bunu karşılıyor.
void f(int a, b); // HATA: varsayılan argümanlı fonksiyon bunu karşılıyor.
```

Sınıflardaki yöntemlere (method) de fonksiyonlar gibi aşırı yükleme yapılır. Yani farklı parametrelerle yöntem yeniden tanımlanır. Aşırı yüklemede yöntemin kimliği ve geri dönüş tipi değişmez. Farklı argümanlar ile yöntemin gövdesi ile yeniden tanımlanır

```
#include <iostream>
using namespace std;
class Karmasik {
public:
    float gercek, sanal;
    Karmasik() { // varsayılan yapıcı (default constructor)
        gercek = 0.0;
        sanal = 0.0;
    }
    void yaz() { //yaz yöntemi
        cout<< gercek << "+" << sanal << +"i" << endl;
    }
    void yaz(char pKarakter) { // aşırı yüklenmiş yaz yöntemi
        //aşırı yüklenmiş yaz yöntemi
        cout << pKarakter << gercek << "+" << sanal << +"i" <<pKarakter << endl;
    }
    void yaz(char pIlkKarakter, char pSonKarakter) { //aşırı yüklenmiş bir başka yaz yöntemi
        cout << pIlkKarakter << gercek << "+" << sanal
             << "i" << pSonKarakter << endl;
    }
};
int main(void) {
    Karmasik c1;
    c1.gercek=1;
    c1.sanal=2;
    c1.yaz();             // 1+2i
    c1.yaz('@');          // @1+2i@
    c1.yaz('[', ']');     // [1+2i]
}
```

Hali hazırda tanımlı olan işleçlerle (operator) bizim tanımladığımız sınıflardan imal edilen nesneler üzerinde işlem yapmak için aşırı yükleme yapılabilir. Ya da istenmeyen işleçler =delete belirleyicisi (=delete specifier) ile sınıfla işlem yapması kaldırılabilir. Bu işleç aynı zamanda kalıtımdan davranılan davranışları ve yapıcıları kaldırmak için de kullanılabilir. Aşağıda toplama işlecine aşırı yükleme örneği verilmiştir;

```
#include <iostream>
using namespace std;
class Karmasik {
```

```
public:
    float gercek, sanal;
    Karmasik() { // varsayılan yapıcı (default constructor)
        gercek = 0.0;
        sanal = 0.0;
    }
    Karmasik(float pGercek, float pSanal): gercek(pGercek),sanal(pSanal) {
    }
    Karmasik(float pGercek): gercek(pGercek) {
        sanal = 0.0;
    }
    void yaz() { //yaz yöntemi
        cout<< gercek << "+" << sanal << +"i" << endl;
    }
    Karmasik operator+(const Karmasik& pKarmasik) {
        /* + işleci aşırıyükleniyor (overloading): iki karmaşık sayı toplanıyor */
        Karmasik toplam;
        toplam.gercek = gercek + pKarmasik.gercek;
        toplam.sanal = sanal + pKarmasik.sanal;
        return toplam;
    }
    Karmasik operator+(const float pGercek) {
        /* + işleci aşırı yükleniyor (overloading): karmaşık sayı ile
           gerçek sayı toplanıyor */
        Karmasik toplam;
        toplam.gercek = gercek + pGercek;
        toplam.sanal = sanal;
        return toplam;
    }
    Karmasik(const Karmasik& pKarmasik) { /*bir başka aşırı yüklenmiş,
                                            kopya yapıcı (copy constructor):
                                            atama (=) için tanımlanmalıdır. */
        gercek= pKarmasik.gercek;
        sanal=pKarmasik.sanal;
    }
    void operator-(const Karmasik &) = delete;/* Örnek olarak Çıkarma işlemi
                                                 engellenmiş. */
};
int main(void) {
    Karmasik c1;
    c1.yaz();
    Karmasik c2(1.0,1.0);
    c2.yaz();
    Karmasik c3(1.0);
    c3.yaz();
    Karmasik c4=c2+c3;
    c4.yaz();
    c1=c2+c3;
    c1.yaz();
}
```

Yapıcıların Aşırı Yüklenmesi

Yapıcıların görevi kısaca imal edilecek nesnelerin durumlarına (state) ilk değer vermektir. Durumlara ilişkin veriler de alanlarda (field) tutulurlar. Nesneleri de dışarıdan verilen parametrelerle farklı durumlarda imal etmek için aşırı yüklenmiş yapıcıları kullanabiliriz. Yani farklı parametrelerle yapıcıları yeniden tanımlayabiliriz.

Yapıcıların kimliği sınıf kimliğiyle aynıdır. Aşırı yüklemede tanımlanacak yapıcının kimliği de sınıf kimliği ile aynı olur. Yapıcılar, yöntemlerin aksine geri değer döndürmezler. Aşağıda aşırı yüklenmiş yapıcı, parametreli (parameterized constructor),

```
#include <iostream>
using namespace std;
class Karmasik {
public:
    float gercek, sanal;
    Karmasik() { /* varsayılan (default constructor):
        Gerçek ve sanal kısım 0 olarak nesneler imal edilir. */
        gercek = 0.0;
        sanal = 0.0;
    }
    Karmasik(float pGercek, float pSanal) { /* aşırı yüklenmiş yapıcı:
        Gerçek ve sanal kısım parametrelerden gelen değerler olacak şekilde
        olarak nesneler imal edilir. */
        gercek = pGercek;
        sanal = pSanal;
    }
    Karmasik(float pGercek) {/* aşırı yüklenmiş bir başka yapıcı:
        Gerçek parametrelerden gelen değer olacak şekilde sanal kısım 0
        olarak nesneler imal edilir. */
        sanal=0.0;
    }
    Karmasik(const Karmasik& pKarmasik) { /*bir başka aşırı yüklenmiş,
        kopya yapıcı (copy constructor):
        Parametre olarak bir başka karmaşık sayı veriliyor ve gerçek ve sanal
        Durumları, parametreden gelen karmaşık sayı ile aynı olacak şekilde
        nesneler imal edilir. */
        gercek= pKarmasik.gercek;
        sanal=pKarmasik.sanal;
    }
    void yaz() { //yaz yöntemi
        cout<< gercek << "+" << sanal << +"i" << endl;
    }
};
int main(void) {
    Karmasik c1; // varsayılan yapıcı ile imal edilen c1 nesnesi
    c1.yaz(); // 0+0i
    Karmasik c2(2.0,1.0); // aşırı yüklenmiş yapıcı ile imal edilen c2 nesnesi
    c2.yaz(); // 2+1i
    Karmasik c3=c2; // kopya yapıcı ile imal edilen c3 nesnesi
    c3.yaz(); // 2+1i
    Karmasik c4(12.0);
    c4.yaz(); //12+0i
}
```

Yukarıdaki örnekte;

- Aynı sınıftan bir nesneyi bir başka imal edilmiş nesnenin durumlarına sahip olarak imal edebiliriz. Bir nesneyi bir yapıcıya argüman olarak geçirmek ve yeni nesneye argümanın durumlarını vermek gerekir. `Karmasik(const Karmasik& pKarmasik)` şeklinde kodlanmış bu yapıcıya kopya yapıcı (copy constructor) adı verilir.
- Yukarıdaki örnekte `Karmasik(float pGercek, float pSanal)` ve `Karmasik(float pGercek)` yapıcıları parametreli yapıcı (parameterized constructor) olarak aşağıdaki şekilde tanımlanabilir.

```
    Karmasik(float pGercek, float pSanal): gercek(pGercek), sanal(pSanal) {
      /* parametreli yapıcı, her bir alana parametrede atama yapılır.*/
    }
    Karmasik(float pGercek): gercek(pGercek), sanal(0.0) {
```

```
    }
```

– Üstte örneği verilen ikinci yapıcı, yapıcı devretme (delegating constructor) ile aşağıdaki şekilde de tanımlanabilir;

```
    Karmasik(float pGercek): Karmasik (pGercek, 0.0) {
       /* Bir yapıcı görevini bir başka yapıcıya devrederek
          Alanların (field) ilk değerleri belirleniyor. */
    }
```

Yukarıda belirtilen ön tanımlı yapıcı (default constructor), =default belirleyicisi (=default specifier) ile değiştirilebilir. Bu belirleyici aşırı yüklenmiş fonksiyonlarda da kullanılabilir[16]. Aşağıda bu belirleyiciye ilişkin örnek verilmiştir;

```
#include <iostream>
using namespace std;
class A {
public:
    A(int x) // Kullanıcı Tanımlı parametreli yapıcı
    {
        cout << "Parametreli Yapıcı" << endl;
    }
    A() = default; // derleyiciye A() yapıcısının ön tanımlı olduğu bildiriliyor
};
int main()
{
    A a; // A() yapıcısı ile nesne imal edilir.
    A x(1); // A(int x) yapıcısı ile nesne imal edilir.
    return 0;
}
```

Kalıtım (inheritance) durumunda ise taban sınıfın yapıcısı çağrılır. Bu duruma aşağıdaki örnek verilebilir;

```
#include <iostream>
using namespace std;

class Kisi {
private:
    unsigned yas;
    char cinsiyet;
public:
    Kisi() {
        yas=18;
        cinsiyet='E';
    }
    Kisi(unsigned pYas, char pCinsiyet): yas(pYas), cinsiyet(pCinsiyet) {

    }
    void setYas(unsigned yeniYas) {
        if (yeniYas <18)
            cout << "Yaş 18 den Küçük Olamaz." << endl;
        else
            yas=yeniYas;
    }
    unsigned getYas() {
        return yas;
    }
    void setCinsiyet(char yeniCinsiyet) {
```

[16] https://www.geeksforgeeks.org/explicitly-defaulted-deleted-functions-c-11/

```
        if (yeniCinsiyet == 'E' ||
            yeniCinsiyet == 'E' ||
            yeniCinsiyet == 'K' ||
            yeniCinsiyet == 'k' ||
            yeniCinsiyet == 'B' ||
            yeniCinsiyet == 'b')
            cinsiyet=yeniCinsiyet;
        else
            cout << "Cinsiyet E,e,K,k,B,b Dışında Bir Değer Olamaz." << endl;
    }
    char getCinsiyet() {
        return cinsiyet;
    }
    void yasiniSoyle()
    {
        cout << "Yaşım:" << yas << endl;
    }
    void cinsiyetSoyle()
    {
        if (cinsiyet=='E' || cinsiyet=='e')
            cout << "Cinsiyetim: ERKEK" << endl;
        else if (cinsiyet=='K' || cinsiyet=='k')
            cout << "Cinsiyetim: KADIN" << endl;
        else
            cout << "Cinsiyetim: SÖYLEMEK İSTEMİYORUM!" << endl;
    }
};
class Ogrenci: public Kisi {
private:
    unsigned ogrenciNo;
public:
    Ogrenci() {
        //yas 18 ve cinsiyet 'E' olarak taban sınıfta tanımlanır.
    }
    Ogrenci(unsigned pYas, char pCinsiyet): Kisi(pYas,pCinsiyet) /* Taban
      sınıftaki yas ve cinsiyet private olduğundan ulaşılamaz.
      Ama taban sınıfın yapıcısına parametreler gönderilerek ilk değer
      verilebilir. */
    {
        ogrenciNo=1;
    }
    void dersCalis() {
        cout << "Ders Çalışıyorum..." << endl;
    }
};

int main() {
    Ogrenci ilhan;
    ilhan.yasiniSoyle(); // Yaşım:18
    ilhan.cinsiyetSoyle(); // Cinsiyetim ERKEK
    ilhan.dersCalis();

    Ogrenci ayse(25,'K');
    ayse.yasiniSoyle(); // Yaşım:25
    ayse.cinsiyetSoyle(); // Cinsiyetim KADIN
    ayse.dersCalis();
}
```

Ön tanımlı argümanlar (default argument) da yapıcılarda kullanılabilir. Karmaşık sayılar örneği aşağıda verilmiştir;

```
#include <iostream>
```

```
using namespace std;
class Karmasik {
    float gercek, sanal;  //bu üyeler varsayılan olarak private
public:
    Karmasik(float pGercek=0.0, float pSanal=0.0): gercek(pGercek),sanal(pSanal) {
    }
    void yaz() { //yaz yöntemi
        cout<< gercek << "+" << sanal << +"i" << endl;
    }
    Karmasik operator+(const Karmasik& pKarmasik) {
        Karmasik toplam;
        toplam.gercek = gercek + pKarmasik.gercek;
        toplam.sanal = sanal + pKarmasik.sanal;
        return toplam;
    }
    Karmasik operator+(const float pGercek) {
        Karmasik toplam;
        toplam.gercek = gercek + pGercek;
        toplam.sanal = sanal;
        return toplam;
    }
    Karmasik(const Karmasik& pKarmasik) { // copy constructor
        gercek= pKarmasik.gercek;
        sanal=pKarmasik.sanal;
    }
};
int main(void) {
    Karmasik c1; // Karmasik(0.0,0.0) yapıcısı ile imal edildi.
    c1.yaz();
    Karmasik c2(1.0,1.0); // Karmasik(1.0,1.0) yapıcısı ile imal edildi.
    c2.yaz();
    Karmasik c3(1.0); // Karmasik(1.0,0.0) yapıcısı ile imal edildi.
    c3.yaz();
    Karmasik c4=c2+c3; // c4 kopya yapıcısı ile imal edildi.
    c4.yaz();
    c4=c2+c3; // c4 kopya yapıcısı ile imal edildi.
    c4.yaz();
}
```

Yapıcıların aşırı yüklenmesinin çeşitli üstünlükleri vardır;

- Nesne imal etmede esneklik: Aynı sınıftan farklı durumlara sahip nesneler imal edilebilir.
- Gelişmiş kod bakımı ile daha temiz ve okunabilir kod oluşur.
- Nesne imal etme mantığı diğer nesnelerden/programlardan gizlenir.
- Kopya yapıcılar ile nesne klonlama basitleşir.

Taşıma Yapıcısı

Bir kopya yapıcı (copy constructor) yazmak için, yani bir nesneyi kopyalayan ve yeni bir nesne oluşturan bir fonksiyon oluşturmak için, normalde aşağıda gösterilen sözdizimini seçerdik;

```
class A {
public:
    int a;
    int b;
    A(const A &other) {
        this->a = other.a;
        this->b = other.b;
    }
};
```

A için A tipinde başka bir nesneye referans alan bir yapıcımız olurdu ve nesneyi yöntemin içinde elle kopyalardık. Alternatif olarak, tüm üyeleri otomatik olarak kopyalayan `A(const A &) = default;` yazabilirdik.

Ancak bir taşıma yapıcısı (move constructor) oluşturmak için lvalue referansı yerine rvalue referansı alacağız. Yani referans alınan nesne durumları yeni nesneye taşınıyor. Örneğin;

```
class Cuzdan {
public:
    int para;
    Cuzdan () = default; //ön tanımlı yapıcı (default constructor)
    Cuzdan (Cuzdan &&diger) {
        this->para = diger.para;
        diger.para = 0;
    }
};
```

Burada durumları taşınan referans nesnenin durumunun sıfırlandığına dikkat edilmesi gerekir. Taşıma semantiği orijinal örnekten (instance) 'durum çalmaya' izin verecek şekilde tasarlanmıştır. Orijinal örneğin bu çalmadan sonra nasıl görünmesi gerektiğini üzerinde durulmalıdır. Bu durumda, değeri sıfıra değiştirmeseydik tüm nesnelerdeki para miktarını iki katına çıkarmış olurduk;

```
Cuzdan a;
a.para = 10;
Cuzdan b (std::move(a)); // B(B&& diger); yapıcısı çağrılır.
std::cout << a.para << std::endl; //0
std::cout << b.para << std::endl; //10
```

Böylece eski bir nesneden yeni bir nesne imal etmiş olduk.

This Göstericisi Kelimesi

C++'da her nesne, `this` adı verilen önemli bir gösterici aracılığıyla kendi adresine erişebilir. `this` göstericisi (this pointer) tüm üye yöntemler için örtük bir parametredir. Aşağıda karmaşık sayıları örneği verilmiştir;

```
#include <iostream>
using namespace std;

class Karmasik {
    float gercek, sanal;
public:
    Karmasik(float pGercek=0.0, float pSanal=0.0) {
        this->gercek=pGercek;
        this->sanal=pSanal;
    }
    void yaz() { //yaz yöntemi
        cout<< this->gercek << "+" << this-> sanal << +"i" << endl;
    }
    Karmasik operator+(const Karmasik& pKarmasik) {
        Karmasik toplam;
        toplam.gercek = this->gercek + pKarmasik.gercek;
        toplam.sanal = this->sanal + pKarmasik.sanal;
        return toplam;
    }
    Karmasik operator+(const float pGercek) {
        Karmasik toplam;
        toplam.gercek = this->gercek + pGercek;
        toplam.sanal = this->sanal;
        return toplam;
    }
    Karmasik(const Karmasik& pKarmasik) {
```

```
        this->gercek= pKarmasik.gercek;
        this->sanal=pKarmasik.sanal;
    }
};
int main(void) {
    Karmasik c1(1.0,2.0);
    c1.yaz();
    Karmasik c2(3.0,5.0);
    c2.yaz();
    Karmasik c3=c1+c2;
    c3.yaz();
    Karmasik c4;
    c4=c2+c3;
    c4.yaz();
}
```

Friend Anahtar Kelimesi

Bir sınıfın arkadaş (friend) yöntemi, o sınıfın bloğu dışında tanımlanır ancak sınıfın tüm mahrem (private) ve korumalı (protected) üyelerine erişim hakkına sahiptir. Arkadaş yöntemlerin bildirimleri/prototipleri sınıf içerisinde yer almasına rağmen, arkadaşlar üye fonksiyon değildir.

```
#include <iostream>
using namespace std;

class Karmasik {
    float gercek, sanal;  //private members
public:
    Karmasik(float pGercek=0.0, float pSanal=0.0) {
        this->gercek=pGercek;
        this->sanal=pSanal;
    }
    friend void yaz(Karmasik pKarmasik);
};
void yaz(Karmasik pKarmasik) {
    cout << pKarmasik.gercek << "+" <<pKarmasik.sanal <<"i" <<endl;
}
int main(void) {
    Karmasik c1(1.0,2.0);
    yaz(c1); // 1+2i
    Karmasik c2(3.0,5.0);
    yaz(c2); // 3+5i
}
```

Bu durum bir sınıfa akış (stream) nesnelerinden veri çıkarma işleci (stream extraction operator) olan `>>` ile akışlara veri ekleme işleci (stream insertion operator) olan `<<` için çok kullanılırlar. Karmaşık sayı sınıfı için aşağıdaki gibi bir örnek verilebilir;

```
#include <iostream>
using namespace std;

class Karmasik {
private:
float gercek, sanal;  //private members
public:
    Karmasik(float pGercek=0.0, float pSanal=0.0) {
        this->gercek=pGercek;
        this->sanal=pSanal;
    }
    friend ostream& operator<<(ostream& output, const Karmasik& pKarmasik) {
        /*
```

```
        ostream sınıfına, pKarmasik nesnesindeki mahrem (private) ve
        korumalı (protected) üyelere erişme yetkisi, friend anahtar
        kelimesiyle verilmiştir.
        */
        output << pKarmasik.gercek << "+" << pKarmasik.sanal << +"i";
        return output;
    }

};
int main(void) {
    Karmasik c1(1.0,2.0);
    cout << c1 << endl;  // 1+2i
    Karmasik c2(3.0,5.0);
    cout << c2 << endl;  // 3+5i
}
```

Statik Sınıf Üyeleri

C++ Dilinde statik üye yöntemi (static member method), imal edilecek herhangi bir nesneye değil, sınıfın kendisine ait olan özel bir fonksiyon türüdür. Bu yöntemleri tanımlamak için `statik` anahtar kelimesi kullanılır. Sınıfın bir örneği olacak nesne imal edilmesine gerek kalmadan, sadece sınıf kimliğini kullanarak doğrudan çağrılabilirler. Bu tür yöntemlere yardımcı yöntem (utility method) adı verilir. Bu yöntemler program çalışmaya başladığında var olur ve program bitinceye kadar erişilebilirler. Ayrıca yalnızca sınıfın statik üyeleri veya diğer statik fonksiyonlarıyla çalışabilirler.

```
#include <iostream>
using namespace std;

class YardimciSinif {
public:
    static void programciAdi() {
        cout << "Programcı Adı: Ilhan OZKAN" <<endl;
        cout << "Tarih:" << __DATE__  << endl;
    }
};

int main(void) {
    YardimciSinif::programciAdi(); /* static üye yöntemi çağırma;
    SınıfKimligi::yöntemadi() şeklinde kapsam çözümleme işleci :: ile
    çağrılır. */
}
```

Statik yöntemlere benzer şekilde statik alanlar da tanımlanabilir. Statik olan bu veriler veri bellekte (data segment) saklanırlar ve bu veri üyeleri de statik yöntemler gibi kullanılır. Statik üyeler genellikle bir sınıfın tüm örnekleri arasında paylaşılması gereken paylaşılan kaynakları veya sayaçları yönetmek için kullanılır.

İleride anlatılacak olan tasarım desenlerinden (design pattern) tekil nesne deseni (singleton pattern) statik üyelere örnek olarak verilebilir. Bu desende sınıftan yalnızca bir nesne/örnek imal edilen ve bu tekil nesneye evrensel erişim sağlayan bir tasarım desenidir. Burada statik üyeler, sınıfın tek bir paylaşılan örneğinin bakımına izin verdikleri için bu deseni uygulamak için mükemmeldir.

Const Sınıf Üyeleri

Sınıfların `const` yöntemleri (const method), veri üyelerinin yani durumları (state) tutan alanları (field) değiştirme izni verilmeyen fonksiyonlardır. Bir üye fonksiyonunu `const` yapmak için, const anahtar kelimesi fonksiyon prototipine ve ayrıca fonksiyon tanımındaki başlığına eklenir.

Üye yöntemler gibi bir sınıfın nesneleri de `const` olarak bildirilebilir. `const` olarak bildirilen bir nesne değiştirilemez ve bu nedenle, bu fonksiyonlar nesneyi değiştirmemeyi garantilediğinden, yalnızca const üye yöntemlerini çağırabilir.

Bir `const` nesnesi, nesne bildirimine `const` anahtar sözcüğünü önek olarak ekleyerek tanımlanabilir. `const` nesnelerinin veri üyesini değiştirmeye yönelik herhangi bir girişim, derleme zamanı hatasıyla (compile time error) sonuçlanır.

```
#include <iostream>
using namespace std;

class Sinif {
private:
    int alan;
public:
    Sinif() {
        alan=-1;
    }
    int getAlan() const {
        return alan;
    }
    void setAlan(int pAlan) { // Bu yöntem const tanımlanılırsa hata alınır!
        alan=pAlan;
    }
};

int main() {
    Sinif* nesne=new Sinif();
    cout<<"nesne.alan:" << nesne->getAlan() << endl;
    nesne->setAlan(12);
    cout<<"nesne.alan:" << nesne->getAlan() << endl;
    delete nesne;
    const Sinif* constNesne=new Sinif();
    constNesne->setAlan(9); // Hata cont nesne değiştirilemez!
}
```

Mutable Depolama Sınıfı

Bazen, `const` ile sabit olarak tanımlanmış nesne veya yapının (struct) bir veya daha fazla veri üyesini değiştirme gereksinimi doğabilir. İşte bu durumda bu üyeler `mutable` anahtar kelimesiyle tanımlanır.

```
#include <iostream>
using namespace std;
class Kisi {
public:
    int tcno;
    mutable int okulno;

    Test(){
        tcno = 43233445505;
        okulno = -1;
    }
};
int main(){
   const Kisi ogrenci1; // ogrenci1 nesnesi const olarak tanımlandı

   ogrenci1.okulno = 2000; // sabit olan nesnenin okulno değiştiriliyor.
   cout << ogrenci1.okulno;

   ogrenci1.tcno = -1; //HATA: sabit nesnenin alanı değiştirilemez.
}
```

Nesne Yıkıcı

Yıkıcı (destructor), bir nesne yok edileceği zaman otomatik olarak çağrılan yapıcı gibi bir yöntemdir. Bir sınıf içinde yalnızca bir yıkıcı tanımlanabilir. Yani, bir yıkıcı, bir nesne yok edilmeden önce çağrılacak son yöntemdir. Aşağıdaki gibi tanımlanır.

```
~sınıf-kimliği() {
 // ...
}
```

Aşağıda imal edilen karmaşık sayıların nasıl yok edildiğine ilişkin örnek verilmiştir;

```
#include <iostream>
using namespace std;

static int nesneSayaci=0;

class Karmasik {
    float gercek, sanal;
public:
    Karmasik(float pGercek=0.0, float pSanal=0.0): gercek(pGercek),sanal(pSanal) {
        nesneSayaci++;
        cout << nesneSayaci << " nesne imal edildi" << endl;
    }
    void yaz() { //yaz yöntemi
        cout<< gercek << "+" << sanal << +"i" << endl;
    }
    Karmasik operator+(const Karmasik& pKarmasik) {
        Karmasik toplam;
        toplam.gercek = gercek + pKarmasik.gercek;
        toplam.sanal = sanal + pKarmasik.sanal;
        return toplam;
    }
    Karmasik operator+(const float pGercek) {
        Karmasik toplam;
        toplam.gercek = gercek + pGercek;
        toplam.sanal = sanal;
        return toplam;
    }
    Karmasik(const Karmasik& pKarmasik) {
        gercek= pKarmasik.gercek;
        sanal=pKarmasik.sanal;
    }
    ~Karmasik() {
        nesneSayaci--;
        cout << nesneSayaci << " nesne yok edildi" << endl;
    }
};
int main(void) {
    Karmasik c1(1.0,2.0);
    c1.yaz();
    Karmasik c2(3.0,5.0);
    c2.yaz();
    Karmasik c3=c1+c2;
    c3.yaz();
    Karmasik c4;
    c4=c2+c3;
    c4.yaz();
}
/*Program Çıktısı:
1 nesne imal edildi
```

```
1+2i
2 nesne imal edildi
3+5i
3 nesne imal edildi
4+7i
4 nesne imal edildi
5 nesne imal edildi
4 nesne yok edildi
7+12i
3 nesne yok edildi
2 nesne yok edildi
1 nesne yok edildi
0 nesne yok edildi

...Program finished with exit code 0
*/
```

Geçersiz Kılma

Yöntemi geçersiz kılma (overriding), nesne yönelimli programlamanın, taban sınıfta önceden tanımlanmış bir yöntemi türemiş bir sınıfta yeniden tanımlamasına olanak tanıyan bir kavramıdır. Yani taban sınıfın davranışının türemiş sınıfta değiştirilmesidir. C++ dili iki tür fonksiyon geçersiz kılmayı destekler:

- Derleme Zamanı Geçersiz Kılma
- Çalışma Zamanı İşlevi Geçersiz Kılma

Derleme zamanında geçersiz kılma aşağıdaki şekilde yapılır;

```
class Taban-Sınıf-Kimliği {
erişim-belirleyici:
     // geçersiz kılınacak yöntem:
     geri-dönüş-veri-tipi yöntem-kimliği() {
     }
};

class Türemiş-Sınıf-Kimliği: erişim-belirleyici Taban-Sınıf-Kimliği {
erişim-belirleyici:
     // geçersiz kılan yöntem:
     geri-dönüş-veri-tipi yöntem-kimliği() {
     }
};
```

Her iki sınıf ta da yöntemin kimliği aynıdır. Taban sınıftan imal edilen nesne taban sınıfın yöntemini, türemiş sınıftan imal edilen nesne ise türemiş sınıfın yöntemini icra eder.

```
#include <iostream>
using namespace std;

class Baba {
public:
    void yap()
    {
        cout << "Baba şu şekilde yapar ..." << endl;
    }
};
class Cocuk : public Baba {
public:
    void yap()
    {
        cout << "Çocuk şu şekilde yapar..." << endl;
    }
```

```
};
int main() {
    Baba baba;
    baba.yap(); // Baba şu şekilde yapar ...
    Cocuk cocuk;
    cocuk.yap(); // Çocuk şu şekilde yapar...
}
```

Çalışma zamanında geçersiz kılma aşağıdaki şekilde yapılır;

```
class Taban-Sınıf-Kimliği {
erişim-belirleyici:
     // geçersiz kılınacak yöntem:
     virtual geri-dönüş-veri-tipi yöntem-kimliği() {
     }
};

class Türemiş-Sınıf-Kimliği: erişim-belirleyici Taban-Sınıf-Kimliği {
erişim-belirleyici:
     // geçersiz kılan yöntem:
     geri-dönüş-veri-tipi yöntem-kimliği() override {
     }
};
```

Her iki sınıf ta da yöntemin kimliği aynıdır. Geçersiz kılınacak yöntem, taban sınıfın yöntemini türemiş sınıfta değiştirilebileceğini belirtmek için `virtual` anahtar kelimesiyle tanımlanır. Türemiş sınıfta ise geçersiz kılacağı yöntemi `override` anahtar kelimesiyle yeniden tanımlar.

```
#include <iostream>
using namespace std;

class Baba {
public:
    virtual void yap()
    {
        cout << "Baba şu şekilde yapar ..." << endl;
    }
};
class Cocuk : public Baba {
public:
    void yap() override
    {
        cout << "Çocuk şu şekilde yapar..." << endl;
    }
};
int main() {
    Cocuk cocuk;
    cocuk.yap(); // Çocuk şu şekilde yapar...
    Baba* babaPtr=&cocuk;
    babaPtr->yap(); // Çocuk şu şekilde yapar...
/*
    Baba& babaPtr=cocuk;
    babaPtr.yap(); // Çocuk şu şekilde yapar...
*/
}
```

Roller

Ara yüzler (interface) nesnelerin sahip oldukları rollerdir (role). C++ dilinde roller, sınıf (class) olarak kodlanır. Ancak saf nesne yönelimli dillerde ara yüzler (interface) `interface` saklı kelimesiyle tanımlanırlar.

Örnek olarak bir öğretmen; Evde baba rolünde olup babanın sahip olduğu çocuklara bakma, yemek yapma gibi davranışları gösterir. İşte öğretmen rolündedir ve öğrenmenin sahip olduğu; öğretme, sınav/değerlendirme yapma, karne hazırlama, ders notu hazırlama gibi davranışları vardır. Bu öğretmen müzisyen rolünde ise gitar çalma, akort yapma gibi davranışları da vardır. İşte bunların hepsi ayrı bir roldür ve her bir rolde o role ilişkin davranışları sergiler.

C++ Dilinde, saf nesne yönelimli dillerin (Java, C# gibi) aksine çoklu kalıtımı (multiple inheritance) destekler gibi görünür, çünkü `interface` gibi bir anahtar kelimeye sahip olmadığından ara yüz olarak tanımlanmış birçok sınıftan miras alabilir.

Ara yüzler, soyut (abstract) sınıflarda tanımlanır ve veri soyutlaması (data abstraction) yapılmaz. Bir sınıfın soyut olabilmesi için en az bir saf sanal yönteme (pure virtual method) sahip olmalıdır. Sanal sınıflardan nesne imal edilemez! Sanal sınıflardaki sanal yöntemler (virtual method) türeyen sınıfta yeniden tanımlanabilir. Türeyen sınıfta mutlaka geçersiz kılınır (override).

```
#include <iostream>
using namespace std;

class IBaba { //Baba Rolündeki davranışlar:
public:
   virtual void cocugaBak() = 0; // saf sanal yöntem-pure virtual method
   virtual void hanimaYardimEt() = 0;
   /* Bu sınıfta en az bir saf sanal yöntem olduğundan
      bu sınıftan nesne imal edilemez! */
};
class IMuzisyen { //Müzisyen Rolündeki Davranışlar:
public:
   virtual void gitarCal() = 0; // saf sanal yöntem-pure virtual method
   virtual void piyanoCal() = 0;
   /* Bu sınıfta en az bir saf sanal yöntem olduğundan
      bu sınıftan nesne imal edilemez! */
};
class IOgretmen { //Öğretmen Davranışları:
public:
   virtual void ogret() = 0; // saf sanal yöntem-pure virtual method
   virtual void dersNotuHazirla() = 0;
   /* Bu sınıfta en az bir saf sanal yöntem olduğundan
      bu sınıftan nesne imal edilemez! */
};
class Kisi:  public IBaba, public IMuzisyen, public IOgretmen {
    /*
      (interface realization/implementation):
      Gerçekleştirilen IBaba, IMuzisyen ve IOgretmen arayüzlerinde
      saf sanal yöntemler olduğundan bu Kişi sınıfında hepsi
      geçersiz kılınmalıdır (override).
    */
public:
    string adi;
    string soyadi;
    void adiniSoyle() {
        cout << adi << " " << soyadi << endl;
    }
    // IBaba aracılığıyla miras alınan davranışlar:
    void cocugaBak() override {
        cout << "Çocuğa Bakıyorum..." << endl;
    }
    void hanimaYardimEt() override {
        cout << "Hanıma Yardım Ediyorum..." << endl;
    }
    // IMuzisyen aracılığıyla miras alınan davranışlar:
    void gitarCal() override {
```

```
        cout << "Gitar Çalıyorum..." << endl;
    }
    void piyanoCal() override {
        cout << "Piyano Çalıyorum..." << endl;
    }
    // IOgretmen aracılığıyla miras alınan davranışlar:
    void ogret() override {
        cout << "Öğretiyorum..." << endl;
    }
    void dersNotuHazirla() override {
        cout << "Ders notu hazırlıyorum..." << endl;
    }
};
int main() {
    Kisi ilhan;
    ilhan.adi = "İlhan";
    ilhan.soyadi = "ÖZKAN";

    //imal edilen nesne tüm arayüzdeki davranışları gösterebilir:
    ilhan.adiniSoyle(); // İlhan ÖZKAN
    ilhan.ogret(); // Öğretiyorum...
    ilhan.gitarCal(); // Gitar Çalıyorum...
    ilhan.hanimaYardimEt(); // Hanıma Yardım Ediyorum...

    //imal edilen nesne sadece bir roldeki davranışları da gösterebilir:
    IBaba& baba=ilhan;
    baba.cocugaBak(); // Çocuğa Bakıyorum...
    baba.hanimaYardimEt(); // Hanıma Yardım Ediyorum...

    IMuzisyen& muzisyen = ilhan;
    muzisyen.gitarCal(); // Gitar Çalıyorum...
    muzisyen.piyanoCal(); // Piyano Çalıyorum...

    IOgretmen& ogretmen = ilhan;
    ogretmen.dersNotuHazirla(); // Ders notu hazırlıyorum...
    ogretmen.ogret(); // Öğretiyorum...
}
```

Bir sınıfın bir ara yüzdeki davranışları kendine uygulamasına ara yüz gerçekleştirme (interface realization/interface implementation) adı verilir.

Çok Biçimlilik

Çok biçimlilik (polymorphism), birden çok nesnenin olduğu bir ortamda, verilen aynı iletiye (message) karşı, nesnelerin farklı davranış (behavior göstermeleridir (same message-different behavior).

Çok biçimliliğin uygulanabilmesi (implementation) için üç şart vardır;

- Kalıtım (inheritance): Ortak davranışın tanımlandığı taban sınıf ve bu davranışın değiştiği türeyen sınıflar.
- Nesnelere aynı iletiyi verebilmek için ortak davranışı tanımlayan Sanal Yöntem (virtual method).
- Geçersiz Kılan Yöntem (override method): Devralınan davranışı kendine uyarlayan ve devralınan yöntemi geçersiz kılan yeni bir yöntem.

```
#include <iostream>
using namespace std;

class SoyutKisi { //Soyut Kisi Sınıfı
public:
    virtual void raporYaz() = 0;
    /* saf sanal metot:Bu metot, Kişi sınıfını SOYUT yapar. Ayrıca;
```

```
     raporYaz(): Türeyen sınıflardan imal edilecek nesnelere
     aynı mesajı vermek için tanımlanan ortak davranıştır.
    */

    // adi özelliği için veri soyutlaması (data abstraction) yapılıyor.
    string getAdi() {
        return adi;
    }
    void setAdi(string pAdi) {
        if (pAdi!="") this->adi=pAdi;
    }
    SoyutKisi(string pAdi): adi(pAdi) {
    }
private:
    string adi;
};

class Ogrenci: public SoyutKisi {  //Ortak davranış SoyutKisi sınıfından devralınıyor.
public:
    void raporYaz() override { //devralınan yöntemler geçersiz kılınıyor (override)
        cout << getAdi() <<":Öğrenci Ödevilerine İlişkin Rapor Yazıyor..." << endl;
    }
    Ogrenci(string pOgrenciAdi):SoyutKisi(pOgrenciAdi){
    }
};

class Ogretmen: public SoyutKisi {  //Ortak davranış SoyutKisi sınıfından devralınıyor.
public:
    void raporYaz() override { //devralınan yöntemler geçersiz kılınıyor (override)
        cout << getAdi() <<":Ogretmen Verdiği Derslere İlişkin Rapor Yazıyor..."
             << endl;
    }
    Ogretmen(string pOgretmenAdi):SoyutKisi(pOgretmenAdi){
    }
};

class Mudur: public SoyutKisi {  //Ortak davranış SoyutKisi sınıfından devralınıyor.
public:
    void raporYaz() override { //devralınan yöntemler geçersiz kılınıyor (override)
        cout << getAdi() <<":Müdür Akademik Takvime İlişkin Rapor Yazıyor..."
             << endl;
    }
    Mudur(string pMudurAdi):SoyutKisi(pMudurAdi){
    }
};
int main() {
    SoyutKisi* kisi[3];
    kisi[0]=new Ogrenci("İlhan ÖZKAN");
    kisi[1]=new Ogretmen("Hasan YILMAZ");
    kisi[2]=new Mudur("Recep AY");

    for (int i=0; i<3; i++)
        kisi[i]->raporYaz();
        /*
         Her kişiye aynı "raporYaz()" mesajı veriliyor->
         Karşılığında farklı davranışlar sergileniyor.
        */

    for (int i=2; i>=0; i--)
        delete kisi[i];
}
```

```
/* Program Çıktısı:
İlhan ÖZKAN:Öğrenci Ödevilerine İlişkin Rapor Yazıyor...
Hasan YILMAZ:Ogretmen Verdiği Derslere İlişkin Rapor Yazıyor...
Recep AY:Müdür Akademik Takvime İlişkin Rapor Yazıyor...

...Program finished with exit code 0
*/
```

Çok biçimlilik bize çalıştırma anında (run time) üstünlük sağlar. Bu nedenle nesneler de çalıştırma anında imal edilmiştir. Program çalıştırıldığında her `kisi` nesnesine aynı ileti verilmiş ama bu nesneler farklı davranış göstermiştir.

Yazılımcı için çok biçimlilik, farklı tipte nesnelerin aynı isimli yöntemleri (method) çağrıldığında her bir nesnenin kendine özgü davranışı gerçekleştirme yeteneği olup çalıştırma anında üstünlük sağlar. Yani yazılımcı, nesnenin tipine bakmaksızın, nesnelerden aynı davranışı göstermesini ister. Burada gösterilecek davranışın, taban sınıfın davranışı mı olduğu veya türeyen sınıfın davranışı mı olduğuna çalıştırma anında (run time) karar verilir. İşte bu karar verme işlemine geç bağlama (late binding) veya dinamik bağlama (dynamic binding) adı verilir.

Esnekliğin istenmediği bazı durumlarda ise karar verme işlemi derleme anında yapılır. Buna ise statik bağlama (static binding) adı verilir. İkisi arasındaki seçim programcı tarafından yapılır.

Yazılım yapılacak sistemin başlangıç koşullarına bağlı tasarım kararlarının tümü bilinemez. Ayrıca sistemin genişlemesi için gerekli tüm kodlamanın bitirilmesi beklenemez. İşte bunu gerçekleştiren dinamik bağlama, yazılım mimarisinin daha esnek (mevcut bileşenin sistemde yeniden yapılandırmasının kolayca yapılması) ve genişleyebilir (yeni bileşenlerin kolayca sisteme eklenmesi) olmasını sağlar.

Sınıf Çeşitleri

Soyut sınıfların (abstract class) bir örneği (instance) olamaz. Yani bu sınıftan nesne imal edilemez. Bu sınıfın en az bir soyut yöntemi (abstract method) olur. Soyut sınıflar hangi davranışın gösterileceğini belirler ama bu davranışların nasıl gösterileceğini anlatmaz. Bu nedenle soyut yöntem için gövde tanımlanmaz.

```
#include <iostream>
using namespace std;
class SoyutKisi { //Soyut Kisi Sınıfı
public:
    virtual void raporYaz() = 0;  // Bu saf sanal yöntem sınıfı soyut yapar.
    string getAdi() {
        return adi;
    }
    void setAdi(string pAdi) {
        if (pAdi!="") this->adi=pAdi;
    }
    SoyutKisi(string pAdi): adi(pAdi) {
    }
private:
    string adi;
};
int main() {
    // SoyutKisi soyutkisi; //HATA: Soyut sınıftan nene imal edilemez!
}
```

Somut sınıflardan (concrete class) nesne imal edilebilir yani örneği (instance) olabilir. Bu sınıflarda davranış ve durumlar eksiksiz tanımlıdır. Bir sınıf tanımlarken soyut bir sınıfa genel davranış ve durumları toplamak ve ondan miras alarak somut sınıfları tanımlamak her zaman iyi bir seçimdir.

```
#include <iostream>
using namespace std;
```

```
class SoyutKisi { //Soyut Kisi Sınıfı
public:
    virtual void raporYaz() = 0; // Saf sanal yöntem. Bu sınıfı soyut yapar.
    string getAdi() {
        return adi;
    }
    void setAdi(string pAdi) {
        if (pAdi!="") this->adi=pAdi;
    }
    SoyutKisi(string pAdi): adi(pAdi) {
    }
private:
    string adi;
};

class SomutKisi: public SoyutKisi {
    //Ortak davranış SoyutKisi sınıfından devralınıyor.
public:
    void raporYaz() override {
        cout << getAdi() <<": Rapor Yazıyor..." << endl;
    }
    // Bu sınıfta sanal yöntem olmadığından nesne imal edilebilir.
    SomutKisi (string pAdi):SoyutKisi(pAdi){
    }
};
int main() {
    SomutKisi somutKisi("Ilhan");
    somutKisi.raporYaz(); // "Ilhan: Rapor Yazıyor..."
}
```

Taban sınıf (base class), miras alınan sınıf veya genel özellik ve davranışların toplandığı yani genelleştirmenin yapıldığı genel sınıf (general class) veya ebeveyn sınıftır (parent).

Türemiş sınıf (derived class), miras alan sınıf veya davranış ve durumlar hakkında uzmanlaşmış yani uzman sınıf (special class) veya çocuk sınıftır (child class). Türemiş ya da miras almış sınıf.

Kök sınıf (root class), babası olmayan ya da yetim sınıf olup aynı zamanda baba sınıftır.

Yaprak sınıf (leaf class), kendisinden henüz miras alan bir sınıf olmayan bir sınıftır. Kısaca türememiş sınıftır.

Kısır sınıf (sealed class) kendisinden türeyen bir sınıf tanımlayamayan bir sınıftır. Kısaca türeyemeyen bir sınıftır. C++ diline 2011 yılında bunu yapabilmek için `final` anahtar kelimesi (final keyword) eklenmiştir.

```
#include <iostream>
using namespace std;

class Dikdortgen final { //Somut Dikdirtgen Sınıfı
public:
    virtual float alanHesapla() {
        return en*boy;
    };
    Dikdortgen(float pEn,float pBoy): en(pEn),boy(pBoy) {
    }
private:
    float en, boy;
};

class Kare: public Dikdortgen {  //HATA: Dikdörtgen sınıfı final ile tanımlanmış
public:
    Kare(float pEn) : Dikdortgen(pEn, pEn) {
```

```
    }
};
int main() {
    Dikdortgen dikdortgen(4.0,5.0);
    cout << "Dikdörgenin alanı:" << dikdortgen.alanHesapla() << endl;
}
```

Final anahtar kelimesi bir sınıfta sadece yöntemler içinde kullanılabilir. Bu durumda türeyen sınıfta bu yöntem geçersiz kılınamaz (override);

```
#include <iostream>
using namespace std;

class Dikdortgen { //Somut Dikdirtgen Sınıfı
public:
    virtual float alanHesapla() {
        return en*boy;
    };
    virtual float kisaKenar() final {
        return (en>boy)?boy:en;
    }
    Dikdortgen(float pEn,float pBoy): en(pEn),boy(pBoy) {
    }
private:
    float en, boy;
};

class Kare: public Dikdortgen {
public:
    float kisaKenar() overide {
        //HATA: Dikdörtgen sınıfında kisaKenar yöntemi final ile tanımlanmış
        //...
        return 0.0;
    }
    Kare(float pEn) : Dikdortgen(pEn, pEn) {
    }
};
int main() {
    Dikdortgen dikdortgen(4.0,5.0);
    cout << "Dikdörgenin kısa kenarı:" << dikdortgen.kisaKenar() << endl;
}
```

Tekil sınıftan (singleton class) yalnızca bir nesne imal edilebilir. Yani yalnızca bir örneği vardır. Bunun olabilmesi için mahrem (private) yapıcısı olması gerekir. *Tekil Nesne* başlığında incelenecektir.

Yardımcı sınıf (utility class), üyeleri statik olan sınıflardır. Ayrıca evrensel yapılandırma ayarlarını (configuration settings) veya değişmezleri (literal) depolamak için kullanılabilir. Bir kaynak havuzunu (önbellek, veri tabanı bağlantı havuzu, vb.) yönetmek ve örnekler arasında paylaşılan bir günlük sistemi uygulamak için yararlıdır. Bunların dışında, izleme yöntemi (trace method) çağrıları için de kullanılabilir.

Unified Modeling Language

Unified Modeling Language (UML) nesneye dayalı yazılım geliştirme ile birlikte gelişen bir modelleme aracıdır. Bu nedenle Nesne Yönelimli Programlama başlığında açıklanan nesne yönelimli birçok kavram, izleyen sayfalarda daha da pekişecektir. Adından da anlaşılacağı üzere nesneye dayalı problem çözmenin kalbi, model oluşturmaktır.

Model, gerçek dünyadaki karmaşık problemin esaslarını soyut olarak önümüze koyar. Buradan hareketle UML'i, basit bir dil, gösterim ya da sözdizimi (syntax) gibi algılamalıyız. UML'in, yazılımı nasıl geliştireceğimizi kesin olarak belirtmediği göz önüne alınmalıdır.

UML, bir grafik modelleme dili olup yazılım sistemlerini oluşturan ana elemanları (Ki bunlar İngilizce "artifact" olarak adlandırılıp ve el yapımı anlamına gelmektedir.) çeşitli şekillerden oluşacak şekilde tanımlamak için oluşturulmuş bir kurallar bütünüdür. Örnek verecek olursak, bir mimari projeye ilişkin maket ne ise yazılım için UML de aynı şeyi ifade eder. Yani UML, söz konusu yazılımın neyi yapacağını anlatan çeşitli şekillerin anlamlı bir şekilde bir araya getirilmesini sağlayan bir standarttır.

Yazılım mühendisliğiyle (software engineering) birlikte 1989 ve 1994 yılları arasında elliden fazla modelleme dili kullanılmaya başlanmıştır. Bu nedenle bu dönem yöntem savaşlarının yaşandığı bir dönem olarak adlandırılır. Bu yöntemlerin çoğu kendi sözdizimini oluşturmuş olmakla birlikte kullandıkları dil elemanları açısından birbirleriyle benzerlik göstermektedirler.

Doksanlı yılların ortalarında üç yöntem, daha güçlü ve yaygın hale gelmiştir. Bu yöntemlerin her biri, diğerlerinin içerdiği elemanları da içeriyordu ancak her birinin diğerine karşı çeşitli güçlü özellikleri bulunuyordu. Bu yöntemler:

– *Booch* yönteminin tasarım (design) ve kodlama (implementation) yönleri oldukça iyi idi. *Grady Booch* Ada diliyle oldukça çalışmıştı ve nesne yönelimli tekniklerin Ada diline uygulanması konusunda yaptığı çalışmalarla bu konuda önemli bir oyuncu olmuştur. Booch metodu güçlü olmasına rağmen model, birçok bulut şeklinden oluşmasından dolayı gösterim şekli oldukça sevimsizdi.
– Object Modeling Technique (OMT) yöntemi, *Jim Rumbaugh* tarafından geliştirilmiş olup, analiz ve veri merkezli bilgi sistemlerinin modellenmesinde oldukça iyi bir yöntemdir.
– Object Oriented Software Engineering (OOSE) modeli use-case olarak bilinen bir model olup *Ivar Jacobson* tarafından geliştirilmiştir. Use-case modeli, yazılımı yapılacak sistemin davranışlarını anlamaya yönelik gelişmiş güçlü bir tekniktir.

1994 yılında *Jim Rumbaugh* ve General Electric firmasından ayrılan *Grady Booch* birlikte Rational şirketini kurarak bir araya geldiler. Bu birlikteliğin amacı, kendi yöntemlerini bir araya getirecek yeni ve tek bir yöntem olan "Unified Method"u oluşturmak idi.

1995 yılında *Ivar Jacobson* da Rational şirketine katıldı. *Ivar Jacobson*'ın use-case'ler konusundaki fikirleriyle birlikte Rational şirketi, bugün Unified Modeling Language-UML olarak adlandırılan, yeni bir "Unified Method" geliştirdi. Üç kişiden oluşan bu grup "Three Amigos" olarak bilinir.

Yöntem savaşlarının ardından güçlü yazılım üreticileri bir araya gelerek OMG adında bir şirketler birliği kurdular[17]. Bu şirketler birliği 1997 yılında UML'e sahip çıkarak herkes tarafından kullanılan bir dil olmasını ve bir standart haline gelmesini sağlamıştır.

UML sadece, gerçek dünyadaki süreçlerin modelini ortaya koyan bir gösterim (notation) sistemidir. Süreçlerin standartlaştırılmasına yönelik bir dil değildir. UML ile ortaya çıkan modeller, süreçleri soyut (abstract) olarak tanımlayacaktır, ancak süreçlerin doğruluğu hakkında bir bilgi vermeyecektir. Yani, A şirketi için çalışan bir süreç, B şirketine uygulandığında felaketle sonuçlanabilir.

Model, yukarıda da anlatıldığı üzere çözülecek probleme ilişkin bir soyutlamadır (abstraction). Etki alanı (domain) ise problemin yaşandığı gerçek dünyayı ifade eder.

Model, birbirlerine ileti (message) gönderen nesnelerden (object) oluşur. Bu modelde nesneler canlı (alive) olarak düşünülmelidir. Nesneler bir şeyler bilir (know) ve bildikleriyle bir şeyler yaparlar (do).

Modelde nesneler, nesne tanımında verilen özellik (property) ve alanları (field) gösteren niteliklere (attribute) sahiptirler. Nesnelerin gösterdikleri davranış (behavior) ya da geliştirdiği yöntemler (method) ise işlem (operation) olarak adlandırılır. Nesnenin özelliklerine ait o anki değerler ise nesnenin durumunu (state) belirler.

UML, sekiz tür diyagrama sahiptir ve bu kitapta sınıf diyagramlarına yer verilecektir[18].

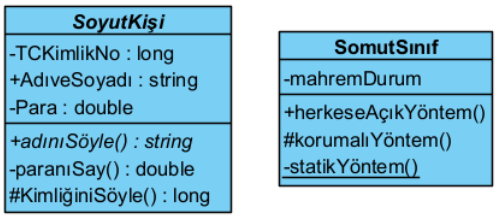
[17] OMG (Object Management Group), www.omg.org

[18] www.togethersoft.com/services/practical_guides/umlonlinecourse/index.html

Sınıf Diyagramları

Sınıf diyagramları (class diagram), yazılımı geliştirilecek olan sisteme ait sınıfları (class) ve bu sınıflar arasındaki ilişkiyi (relationship) gösterir. Sınıf diyagramları durağandır (static). Yani sadece sınıfların birbiriyle etkileşimini (what interacts) gösterir, bu etkileşimler sonucunda ne olduğunu (what happens) göstermez.

Sınıf diyagramlarında sınıflar, bir dikdörtgen ile temsil edilir. Bu dikdörtgen üç parçaya ayrılır. En üstteki birincisine sınıf ismi yazılır. Ortadaki ikincisinde nitelikler (attribute), en alttaki ve sonuncusunda ise ve işlemler (operation) yer alır.



Şekil 24. Örnek Bir Sınıf Diyagramı

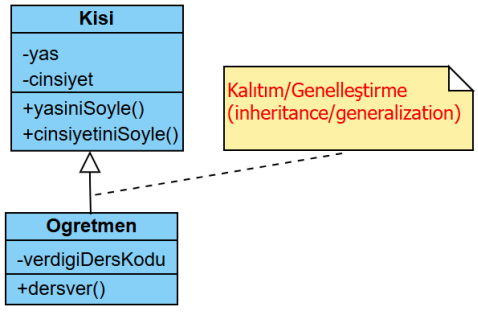
Sınıf isminin italik yazılması, sınıfın soyut (abstract class) sınıf olduğunu gösterir. Benzer şekilde işlemlerin yer aldığı kısımda, yöntemlerin italik yazılması halinde, ilgili yöntemin soyut yöntem (abstract method) olduğunu gösterir.

Sınıflarda erişim belirleyiciler (access modifier), özel karakterlerle birbirinden ayrılır. Burada "-" karakteri mahrem (private), "+" karakteri umuma açık (public), "#" karakteri ise korumalı (protected) olduğunu anlatmaktadır. Statik üyeler (static member), altı çizili olarak gösterilir.

Sınıflar Arası İlişkiler

Nesneleri imal ettiğimiz sınıflar arasında ya ilişki yoktur ya da aşağıda sıralanan altı çeşit ilişki vardır;

Genelleştirme (generalization), taban sınıfla (base class) ile türemiş sınıf (derived class) arasındaki kalıtım (inheritance) ilişkisidir. Örneği *Kalıtım* başlığında verilmiştir. Aşağıda Kişi sınıfı ile bu sınıftan miras alan Öğretmen sınıfının UML diyagramı verilmiştir. Ara yüzler (interface) veya roller de benzer şekilde gösterilir.



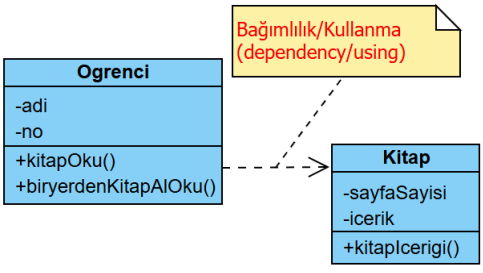
Şekil 25. Genelleştirme İlişkisi UML Diyagramı

Bağımlılık (dependency) ya da bir başka deyişle kullanma (using), bir sınıf ile diğer arasındaki geçici ilişkidir. Burada kullanan sınıf istemci (client) diğer sınıf ise sağlayıcı (supplier) olarak adlandırılır. İki türlü bağımlılık vardır; Birincisinde istemci sınıf, sağlayıcıyı sınıfı parametre olarak kullanır (dependency as a parameter). İkincisinde ise istemci sınıf, sağlayıcı sınıftan örnekleme yapar (dependency as an instantiation).

```
#include <iostream>
using namespace std;

class Kitap {
public:
    string kitapIcerigi() {
```

```
        return icerik;
    }
    Kitap(int pSayfaSayisi, string pIcerik): sayfaSayisi(pSayfaSayisi), icerik(pIcerik) {
    }
private:
    string icerik;
    int sayfaSayisi;
};
class Ogrenci {
public:
    void kitapOku() {
        Kitap* kitap=new Kitap(150,"C++ ile Nesne Yönelimli Programlama Notları");
        /* pKitap nesnesine örnekleme olarak bağımlılık:
           dependency as an instantiation */
        cout << "kitap imal edilip kullanılıyor..." << endl;
        string notlar=kitap->kitapIcerigi();
        //...
        delete kitap;
    }
    void biryerdenKitapAlOku(Kitap* pKitap) {
        /* pKitap nesnesine parametre olarak bağımlılık:
           dependency as a parameter */
        cout << "kitap dışardan alınıp kullanılıyor..." << endl;
        string okunacak=pKitap->kitapIcerigi();
        //...
    };

    Ogrenci(string pAdi,int pNo): adi(pAdi),no(pNo) {
    }
private:
    string adi;
    int no;
};
int main() {
    Ogrenci* ogrenci= new Ogrenci("Ali",12000);
    ogrenci->kitapOku();
    Kitap* kitap=new Kitap(350,"Kurumsal yazılım Geliştirme");
    ogrenci->biryerdenKitapAlOku(kitap);
    delete kitap;
    delete ogrenci;
}
/* Program Çıktısı:
kitap imal edilip kullanılıyor...
kitap dışardan alınıp kullanılıyor...

...Program finished with exit code 0
*/
```

Örnekteki Öğrenci ve Kitap sınıfı arasındaki bağımlılık ilişkisi aşağıdaki UML diyagramında verilmiştir.



Şekil 26. Bağımlılık İlişkisi UML Diyagramı

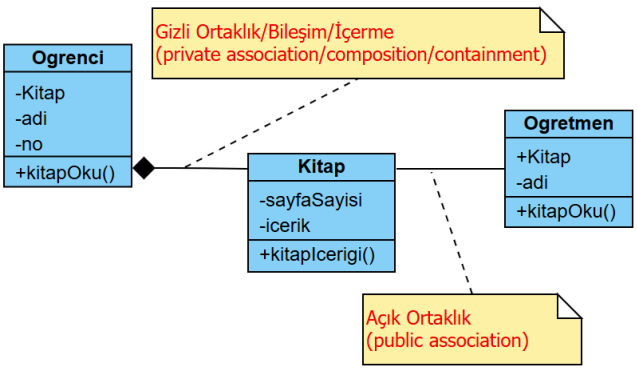
İş Birliği (association), iki farklı sınıf arasındaki sürekli olan bütün-parça (whole-part) ilişkisidir. Bu ilişkide, bir sınıf diğer sınıftan nesne örnekler (imal eder), kullanır ve öldürür. İlişkinin sürekli olabilmesi için kullanılan sınıftan bir değişken alan (field) olarak tanımlanır. İki türü vardır;

- Açık ortaklık (public association): Kullanılan sınıfa ilişkin değişken `public` erişimi ile tanımlanır.
- Gizli ortaklık (private association), içerme (containment) ya da bileşim (composition): Kullanılan sınıfa ilişkin değişken `private` erişimi ile tanımlanır ve dışarıdan erişime izin verilmez.

```
#include <iostream>
using namespace std;

class Kitap {
public:
    string kitapIcerigi() {
        return icerik;
    }
    Kitap(int pSayfaSayisi, string pIcerik): sayfaSayisi(pSayfaSayisi), icerik(pIcerik) {
    }
private:
    string icerik;
    int sayfaSayisi;
};
class Ogrenci { // kitapsız öğrenci olmaz! (private association to kitap)
private:
    string adi;
    int no;
    Kitap* ptrKitap;
public:
    Ogrenci(string pAdi,int pNo): adi(pAdi),no(pNo) {
        ptrKitap=new Kitap(150,"C ile Yapısal Programlama");
        //kitap nesnesi ile ogrenci nesnesi birlikte imal ediliyor:
    }
    void kitapOku() {
        cout << "Öğrenci ile birlikte imal edilen kitap kullanılıyor..." << endl;
        cout << "Okunan Kitap:" << ptrKitap->kitapIcerigi() << endl;
        //...
    }
};
class Ogretmen { // Kitapsız öğretmen olmaz! Ama kitabını değiştirebilir.
//public association to kitap
private:
    string adi;
    Kitap* ptrKitap;
public:
    Ogretmen(string pAdi): adi(pAdi) {
        //kitap nesnesinin referansı ile ogretmen nesnesi birlikte imal ediliyor:
        ptrKitap=new Kitap(200,"C++ ile Nesne Yönelimli Programlama");
    }
    // Kitap* üyesi get ve set yöntemleriye public hale getiriliyor.
    void setKitap(Kitap* pKitap) {
        ptrKitap=pKitap;
    }
    Kitap* getKitap() {
       return ptrKitap;
    }
    void kitapOku() {
        cout << "Öğretmen ile birlikte imal edilen kitap kullanılıyor..." << endl;
        cout << "Okunan Kitap:" << ptrKitap->kitapIcerigi() << endl;
        //...
    }
};
```

```
int main() {
    Ogrenci* ogrenci=new Ogrenci("Ali",12000);
    ogrenci->kitapOku();

    Ogretmen* ogretmen=new Ogretmen("Ilhan");
    ogretmen->kitapOku();

    Kitap* ogretmeninOkuduguKitap=ogretmen->getKitap();
    cout << "Orgetmenin Okuduğu Kitaba Main İçinden Ulaşılabiliyor;" << endl
         << "Okunan Kitap:" << ogretmeninOkuduguKitap->kitapIcerigi() << endl;

    Kitap* baskaKitap=new Kitap(250,"Kurumsal Yazılım Geliştirme");
    ogretmen->setKitap(baskaKitap);
    ogretmen->kitapOku();
    delete ogrenci, ogretmen, baskaKitap;
}
/*Program Çıktısı:
Öğrenci ile birlikte imal edilen kitap kullanılıyor...
Okunan Kitap:C ile Yapısal Programlama
Öğretmen ile birlikte imal edilen kitap kullanılıyor...
Okunan Kitap:C++ ile Nesne Yönelimli Programlama
Orgetmenin Okuduğu Kitaba Main İçinden Ulaşılabiliyor;
Okunan Kitap:C++ ile Nesne Yönelimli Programlama
Öğretmen ile birlikte imal edilen kitap kullanılıyor...
Okunan Kitap:Kurumsal Yazılım Geliştirme

...Program finished with exit code 0
*/
```

Yukarıdaki örnekte Öğrenci ile Kitap arasındaki gizli ortaklık ile Öğretmen ve Kitap arasındaki açık işbirliği aşağıdaki UML diyagramında gösterilmiştir;



Şekil 27. Açık ve Gizli İş Birliği UML Diyagramı

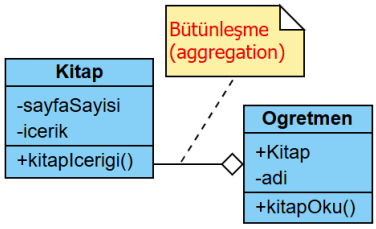
UML diyagramlarında, çift yönlü ortaklıklar iki oka sahip olabilir veya hiç ok olmayabilir ve tek yönlü ortaklıkta veya kendi kendine ortaklık bir oka sahiptir.

Bütünleşme (aggregation), ilişkisi de iki farklı sınıf arasındaki sürekli olan bütün-parça (whole-part) ilişkisidir. Açık ortaklığa (public association) benzer, farklı olarak bu ilişkide kullanılan sınıfa ait başka yerde imal edilmiş nesneyi ya da hazır olan nesneyi kullanır. Tek fark kullanılacak sınıf nesnesinin kullananın iradesi dışında yaratılmış olmasıdır.

```
#include <iostream>
using namespace std;

class Kitap {
public:
    string kitapIcerigi() {
        return icerik;
    }
```

```
    Kitap(int pSayfaSayisi, string pIcerik): sayfaSayisi(pSayfaSayisi), icerik(pIcerik) {
    }
private:
    string icerik;
    int sayfaSayisi;
};
class Ogretmen {
private:
    string adi;
    Kitap* ptrKitap;
public:
    Ogretmen(string pAdi,Kitap* ppKitap): adi(pAdi),ptrKitap(ppKitap) {
        //öğretme nesnesi kitap nesnesi olamadan imal edilemez.
        //Bir nesne imal edilmeden bi kitap nesnesi olmalı
        //aggregation to kitap
    }
    //Kitap* durumu get ve set metodlarıyla public hale getiriliyor.
    void setKitap(Kitap* pKitap) {
        ptrKitap=pKitap;
    }
    Kitap* getKitap() {
        return ptrKitap;
    }
    void kitapOku() {
        cout << "Öğretmen ile birlikte imal edilen kitap kullanılıyor..." << endl;
        cout << "Okunan Kitap:" << ptrKitap->kitapIcerigi() << endl;
        //...
    }
};
int main() {
    Kitap kitap= Kitap(150,"C++ ile Nesne Yönelimli Programlama");
    Ogretmen ilhan("Ilhan",&kitap);
    ilhan.kitapOku();
}
/* Program Çıktısı:
Öğretmen ile birlikte imal edilen kitap kullanılıyor...
Okunan Kitap:C++ ile Nesne Yönelimli Programlama

...Program finished with exit code 0
*/
```

Yukarıdaki örnekte verilen Öğretmen ve Kitap arasındaki bütünleşme ilişkisi aşağıdaki UML diyagramında gösterilmiştir.

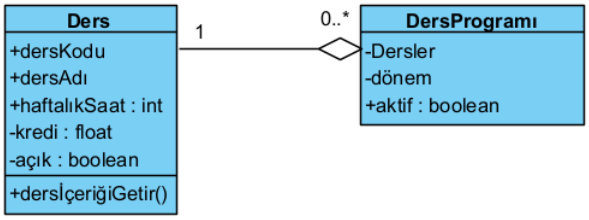


Şekil 28. Bütünleşme UML Diyagramı

Sınıflar arası ilişkilerde birleşme çizgilerinde rakamlar da bulunabilir; Çokluk (multiplicity), ilişkide bir sınıfın örnek (instance) sayısını gösterir. İlişkiyi gösteren çizginin sonunda bir numara olarak gösterilir. Bu numara, mümkün olabilecek örnek sayısıdır. Çokluk, açık ortaklık (public association), gizli ortaklık (private association) ve bütünleşmeye (aggregation) uygulanır;

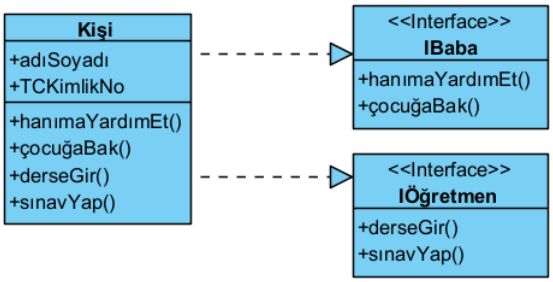
- `0..1` Sıfır ya da 1 örnek.
- `0..` veya `*` Örnek sayısı sınırsızdır.
- `1` Yalnızca 1 örnek

– `1..` En az bir örnek



Şekil 29. Sınıf Diyagramlarında Çokluk Örneği

Ara yüz gerçekleştirme (interface realization) işlemi, ucunda üçgen olan kesikli çizgi ile gösterilir. Üçgenin sivri köşesi gerçekleştirilen ara yüzü gösterir. Çemberler, ara yüzleri (interface) göstermenin bir başka yoludur. Bu durumda daha kısa ve etkili anlatıma sahip diyagramlara sahip oluruz. Bu tip basitleştirilmiş diyagramlar lolipop diyagramı olarak da adlandırılır.



Şekil 30. Sınıf Diyagramlarında Ara yüz gerçekleştirme

Sınıf Tasarlama İlkeleri

Nesne yönelimli programlama (object oriented programming); Kodumuzda değişim yönetimi (change management), gösterici kullanımı yerine referansları kullanarak güvenli kod (safe code) aynı kodları tekrar yazmak (duplicate code) yerine kodların yeniden kullanımı (reusing) sağlayan aşağıdaki özellikleri kullanırız;

– Kod küçüldüğü için kalıtım (inheritance),
– Veri ve kontrol soyutlaması (abstraction) yaparak daha güvenli bileşen tasarladığımız için sarmalama (encapsulation),
– Yazılım mimarisinin daha esnek ve genişleyebilir olması için çok biçimlilik (polymorphism).

Ayrıca yazılım geliştirmede;

– Zayıf bağlaşımın (loosely coupling) sağlanması için yazılımın bir noktasında olabilecek değişikliğin, diğer kısımlarda değişiklik gerektirmeyecek şekilde tasarlanması gerekir. Yani yapılan değişiklik, çorap söküğü gibi yazılımın diğer kısımlarını değiştirmemelidir.
– Aynı çağrışım kümesine ilişkin bütün bileşenler bir arada geliştirilmeli diğer kümelerle karıştırılmamalıdır (seperation of concern). Yani bileşenler arasında yüksek uyum (high cohesion) olmalıdır.

İşte yeniden kullanım, zayıf bağlaşım ve yüksek uyum sağlamak için programcı, nesne yönelimli programlama özellikleriyle birlikte temel ilkelere (principle) uyar.

Programcı kodu tasarlarken, yazarken ve yeniden düzenlerken bu ilkeleri aklında tutarsa; Kod çok daha temiz, genişletilebilir ve test edilebilir olur.

Bunlar ilkelerin İngilizce baş harflerinin alınarak isimlendirilen SOLID ilkeleridir;

Tek Sorumluluk İlkesi (Single Responsibility Principle-SRP):

Sınıflar, tasarlanırken birden fazla işten sorumlu tutulmamalıdır. Mümkün olduğunca bir işle ilgili olarak tasarlanmalıdır. Bir sınıfın tek bir şey yapması gerektiğini ve dolayısıyla değişmesi için tek bir nedeninin olması gerektiğini belirtir.

Açık-Kapalı İlkesi (Open Closed Principle-OCP):

Sınıflar, değişikliklere kapalı (closed for modification) eklentilere açık (open for extension) şekilde tasarlanmalıdır. Bu şekilde değişikliklere açık genişleyebilir modüllere sahip oluruz. Değişikliklere kapalı olmak, mevcut bir sınıfın kodunu değiştirmemek anlamına gelirken, eklentilere açık olma yeni işlevler eklemek anlamına gelir.

Liskov İkame İlkesi (Liskov Substitution Principle-LSP):

Alt sınıflar türediği sınıfların yerine konulabilmelidir. Bu prensipten ilk olarak Barbar Liskov tarafından bahsedilmiştir. Alt sınıflar, türediği sınıf davranışlarıyla kullanılabilmelidir. Burada amaç, alt sınıfların daha çok değişebileceği göz önüne alınarak değişimlerden an az şekilde etkilenmektir.

Türemiş sınıfından bir nesneyi, Taban sınıfından bir nesne bekleyen herhangi bir yönteme parametre olarak geçirdiğimizde yöntemin herhangi bir tuhaf çıktı vermemesi gerektiği anlamına gelir. Bu beklenen davranıştır, çünkü kalıtım kullandığımızda alt sınıfın, üst sınıfın sahip olduğu her şeyi miras aldığını varsayarız. Alt sınıf davranışı genişletir ancak asla daraltmaz. Dolayısıyla bir sınıf bu ilkeye uymadığında, tespit edilmesi zor bazı kötü hatalara yol açar.

Ara Yüzleri Ayırma İlkesi (Interface Segregation Principle-ISP):

Genel amaçlı ara yüzler yerine istemciye bağlı ara yüzler tasarlanmalıdır. Yani roller, birbirinden ayrı olacak şekilde tanımlanmalıdır. Birden fazla rol tek bir rol altında olacak şekilde düşünülmemelidir. Bu nedenle buna rollerin ayrılması prensibi (Role Segregation Principle-RSP) olarak da adlandırılır.

Bağımlılıkları Ters Çevirme İlkesi (Dependency Inversion Principle-DMP):

Sınıflarımızın somut sınıflara ve fonksiyonlara değil, ara yüzlere veya soyut sınıflara bağlı olması gerektiğini belirtir. Nesne yönelimli programlamanın en önemli ilkesidir. Aslında bu ilke ile açık-kapalı ilkesi doğrudan birbiriyle ilişkilidir.