Ödev-07

1. Ödev-07

1.1. Dependency Inversion Prensibi

Bu prensibin amacı yüksek seviye sınıfların düşük seviye sınıflara olan bağımlılığını kaldırmaktır. Dependency Inversion Prensibi, bu işlemi arayüz kullanarak gerçekleştirir.

Örnek olarak bir oyun yazıldığını düşünelim. Bu oyunda Savaşçı, Düşman ve Bilgi için 3 tane sınıfı **Dependency Inversion kullanmadan** yaratalım.

Savaşçı Sınıfı

```
class Savasci
{
private:
    std::string isim;
    int can;
    int hasarGucu;
    std::string tip;

public:
    Savasci(const std::string &isim, int can, int hasarGucu, const
 std::string &tip) : isim(isim), can(can),

                hasarGucu(hasarGucu), tip(tip){}

    void bilgiGoster(){
    std::cout << "Savasci bilgileri: " << std::endl
                << "Isim: " << isim << std::endl
                << "Can: " << can << std::endl
                << "Hasar Gucu: " << hasarGucu << std::endl
                << "Tip: " << tip << std::endl << std::endl;
    }
};
```

Bu savaşçı sınıfında 4 tane özellik var: İsim, can, hasar gücü, silah tipi.

Bu Savaşçının bilgilerini göstermek için **bilgiGoster()** fonksiyonu kullanılmaktadır.

Savaşçı sınıfı gibi bir de düşman sınıfımızı oluşturalım.

Düşman Sınıfı

```
class Dusman
{
private:
    int can;
    int hasarGucu;

public:
    Dusman(int can, int hasarGucu) : can(can), hasarGucu(hasarGucu) {}

    void bilgiGoster() {
        std::cout << "Dusman bilgileri: " << std::endl
                  << "Can: " << can << std::endl
                  << "Hasar Gucu: " << hasarGucu << std::endl <<
 std::endl;
    }
};
```

Düşman sınıfının Savaşçı sınıfından tek farkı isim ve silah tipi özelliklerini barındırmamasıdır. Bu nedenle **bilgiGoster()** fonksiyonu daha farklı çalışmaktadır.

Bir de bu sınıflardaki özellikleri ekrana bastıracak olan bilgi sınıfını oluşturalım.

Bilgi Sınıfı

```
class Bilgi
{
public:
    Bilgi(){}

    void bilgiGoster(Savasci& karakter)                ❶
    {
        karakter.bilgiGoster();
    }

    void bilgiGoster(Dusman& karakter)                 ❷
    {
        karakter.bilgiGoster();
```

```
    }
};
```

❶ Savaşçı için bilgileri basan fonksiyondur. Girdi olarak Savaşçı sınıfından bir örnek kullanır.

❷ Düşman için bilgileri basan fonksiyondur. Girdi olarak Düşman sınıfından bir örnek kullanır.

Main Fonksiyonu

```
int main()
{
    Savasci s1("Savasci1",100,50,"Kilic");
    Dusman d1(75,20);

    Bilgi b;

    b.bilgiGoster(s1);
    b.bilgiGoster(d1);

    return 0;
}
```

Bu programın çıktısı bize s1 ve b1 değişkenlerini b değişkenini kullanarak vermektedir.

```
Savasci bilgileri:
Isim: Savasci1
Can: 100
Hasar Gucu: 50
Tip: Kilic

Dusman bilgileri:
Can: 75
Hasar Gucu: 20
```

Oluşturulan Bilgi sınıfı işimizi görür mü? Evet. Fakat bu yapıyı değiştirmek istersek ne olur?

> **Bilgi sınıfının çıkarabileceği sorunlar:**
>
> - Savaşçı ve Düşman dışında bir sınıf eklemek istediğimizde Bilgi sınıfını da yeniden düzenlememiz gerekecek.
> - Bilgi sınıfındaki fonksiyonlar temel sınıflardan oluşturulan örneklere bağlıdır.

Peki Bu Sorunun Önüne Nasıl Geçilir?

Kodda değişiklik yaptığımızda Bilgi sınıfı ile ilgili herhangi bir sıkıntı yaşamak istemiyorsak **Dependency Inversion Prensibini** kullanmamız gerekiyor.

Dependency Inversion Prensibinin iki amacı vardır.

- Yüksek seviyeli sınıflar, düşük seviyeli sınıflara bağlı olmamalı. İkisi de arayüzlere bağlı olmalıdır.
- Arayüzler detaylara bağlı olmamalı. Detaylar arayüzlere bağlı olmalı.

Bu maddelere baktığımız zaman karşımıza şöyle bir tablo çıkıyor:

Üst Seviye Sınıflar → Arayüz → Temel Seviye Sınıfları

O zaman bu sınıflar için bir arayüz oluşturmamız gerekiyor. Bu arayüzü de Karakter adı altında oluşturalım.

Karakter Arayüzü

```
class Karakter
{
public:
    virtual void bilgiGoster() = 0;
};
```

Bu arayüzü kullanarak bütün sınıfları tekrardan oluşturalım.

Savaşçı Sınıfı

```
class Savasci : public Karakter                ❶
{
```

```
private:
    std::string isim;
    int can;
    int hasarGucu;
    std::string tip;

public:
    Savasci(const std::string &isim, int can, int hasarGucu, const
 std::string &tip) : isim(isim), can(can),

                hasarGucu(hasarGucu), tip(tip){}

    void bilgiGoster() override {                    ❷
        std::cout << "Savasci bilgileri: " << std::endl
                  << "Isim: " << isim << std::endl
                  << "Can: " << can << std::endl
                  << "Hasar Gucu: " << hasarGucu << std::endl
                  << "Tip: " << tip << std::endl << std::endl;
    }
};
```

❶ Savaşçı sınıfı **Karakter sınıfı implement edilerek** oluşturulmuştur.

❷ bilgiGoster() fonksiyonu Karakter sınıfındaki saf sanal fonksiyonu override ederek oluşturulmuştur.

Düşman Sınıfı

```
class Dusman : public Karakter                   ❶
{
private:
    int can;
    int hasarGucu;

public:
    Dusman(int can, int hasarGucu) : can(can), hasarGucu(hasarGucu) {}

    void bilgiGoster() override {                    ❷
        std::cout << "Dusman bilgileri: " << std::endl
                  << "Can: " << can << std::endl
                  << "Hasar Gucu: " << hasarGucu << std::endl <<
 std::endl;
    }
};
```

❶ Düşman sınıfı **Karakter sınıfı implement edilerek** oluşturulmuştur.

❷ bilgiGoster() fonksiyonu Karakter sınıfındaki saf sanal fonksiyonu override ederek oluşturulmuştur.

Bütün temel sınıfları oluşturduğumuza göre artık üst seviye sınıfımız olan Bilgi sınıfını oluşturabiliriz.

Bilgi Sınıfı

```
class Bilgi
{
public:
    Bilgi(){}

    void bilgiGoster(Karakter& karakter)     ❶
    {
        karakter.bilgiGoster();
    }
};
```

❶ Burada Karakter arayüzü kullanılarak sadece bir fonksiyon yazılmıştır.

Sınıflar oluşturulduğuna göre yine aynı Main fonksiyonu ile kodu çalıştıralım.

Main Fonksiyonu

```
int main()
{
    Savasci s1("Savasci1",100,50,"Kilic");
    Dusman d1(75,20);

    Bilgi b;

    b.bilgiGoster(s1);
    b.bilgiGoster(d1);

    return 0;
}
```

Çıktı

```
Savasci bilgileri:
Isim: Savasci1
```

```
Can: 100
Hasar Gucu: 50
Tip: Kilic

Dusman bilgileri:
Can: 75
Hasar Gucu: 20
```

Görüldüğü gibi çıkan sonuçta hiç bir değişiklik yok

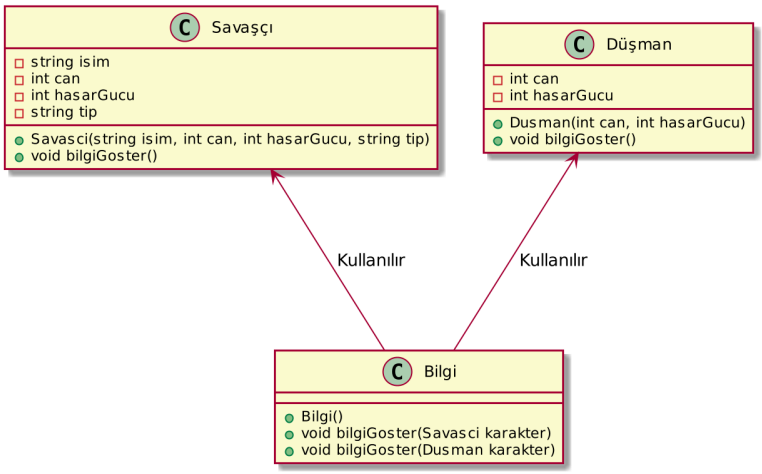
Peki sonuç değişmediğine göre ben bunu yapıp ne kazandım?

- Bununla beraber ben Savaşçı, Düşman gibi yeni bir temel seviye sınıf eklemek istediğim zaman Bilgi sınıfını değiştirmeme gerek kalmayacak.
- Bilgi sınıfındaki fonksiyonlar hiç bir şekilde temel seviye sınıfların örneklerini kullanmamaktadır.

Bu tür yazılımlarda kodlamayı kolaylaştırmak için kullanılan bir yöntemdir. Bir sınıfta değişiklik yaptığınızda diğer sınıfların etkilenmemesi her zaman öncelik olmalıdır.

1.2. UML Diagramları

Dependency Inversion Kullanmadan



Dependency Inversion Kullanarak