

Szoftvertchnikák laborgyakorlat

1. mérés

A modell és a kód kapcsolata

C# alapok

A modell és a kód kapcsolata

Interfész és absztrakt (ős)osztály alkalmazástechnikája

A gyakorlatot kidolgozta: Benedek Zoltán

Utolsó módosítás ideje: 2018.02.10

Contents

A GYAKORLAT CÉLJA	3
BEVEZETŐ	3
FELADAT 1 – „HELLO WORLD” .NET KONZOL ALKALMAZÁS ELKÉSZÍTÉSE	4
AZ UML ÉS A KÓD KAPCSOLATÁNAK ELMÉLETE [HALLGATÓ]*	5
BEVEZETŐ	6
OSZTÁLYOK LEKÉPEZÉSE	6
I. ÁLTALÁNOSÍTÁS, SPECIALIZÁCIÓ KAPCSOLAT	7
II. ASSZOCIÁCIÓ.....	7
III. AGGREGÁCIÓ (TARTALMAZÁS, RÉSZ-EGÉSZ VISZONY).....	9
IV. FÜGGŐSÉG (DEPENDENCY)	9
INTERFÉSZ ÉS ABSZTRAKT (ŐS)OSZTÁLY [HALLGATÓ]*	9
FELADAT 2 - AZ UML ÉS A KÓD KAPCSOLATÁNAK SZEMLÉLTETÉSE.....	11
FELADAT 3 - AZ INTERFÉSZ ÉS AZ ABSZTRAKT ŐSOSZTÁLY ALKALMAZÁSTECHNIKÁJA.....	17
ELLENŐRZŐ KÉRDÉSEK [HALLGATÓ]*	25

A gyakorlat célja

A kapcsolódó előadások:

- A gyakorlathoz nem kapcsolódik előadás.

A gyakorlat célja:

- Ismerkedés a hallgatókkal/gyakorlatvezetővel
- A gyakorlatokra vonatkozó követelmények pontosítása
- Elindulás Visual Studioval és .NET alkalmazások fejlesztésével. A hallgatók korábbi tanulmányaik során C++ gyakorlaton már használták a Visual Studio környezetet, így a cél jelen esetben az ismeretek felelevenítése és életünk első C# alkalmazásának elkészítése. Ugyanakkor előfordulhat, hogy a hallgatók nem emlékeznek pontosan a Visual Studio használatára, így a feladatok megoldása során ezeket folyamatosan elevenítsük fel (pl. Solution Explorer, F5-futtatás, breakpoint használat, stb.)
- Egy egyszerű Hello World .NET alkalmazás elkészítése, C# alapok
- Az UML és a kód kapcsolatának szemléltetése
- Az interfész és az absztrakt őssztály alkalmazástechnikája

A hallgatók .NET alkalmazásokat még nem fejlesztettek, viszont C++ és Java fejlesztési alapismeretekkel rendelkeznek.

Bevezető

A gyakorlatvezető a gyakorlat elején összefoglalja a gyakorlatokra vonatkozó követelményeket:

- A tárgyi adatlapon ezek többsége megtalálható
- Az otthoni feladatokról információ a tárgy honlapján található, illetve itt kerül a félév folyamán hirdetményekben kihirdetésre

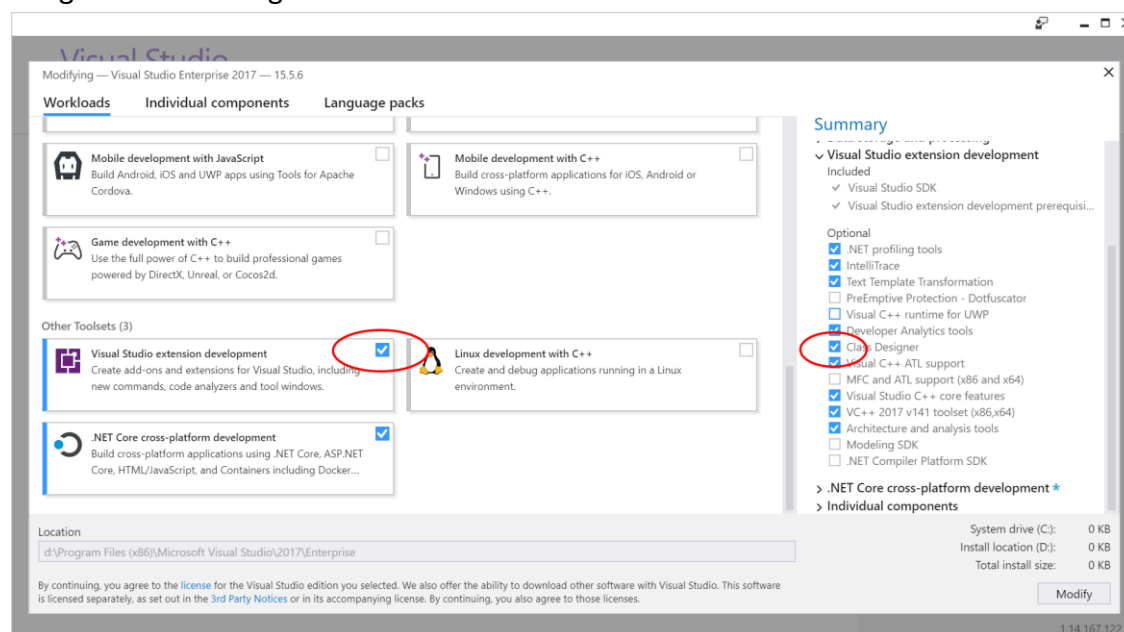
Visual Studio fejlesztőeszközzel, .NET alkalmazásokat fogunk készíteni. Nagyon hasonlít a Javához, fokozatosan ismerjük meg a különbségeket.

Visual Studio-ból a legfrissebb verziót célszerű feltenni. A Community Edition, Professional és az Enterprise verzió is megfelel. A Community Edition ingyenes, letölthető a Microsoft honlapjáról. A Professional fizetős, de az egyetem hallgatói számára ez is ingyenesen elérhető, az msdnaa.bme.hu honlapon.

Visual Studio Class Diagram támogatás

Jelen labor bizonyos feladatainál (és az első házi feladat esetében is) a Visual Studio Class Designer támogatását használjuk. A Visual Studio 2017 – és jó eséllyel a később megjelenő változatok - nem teszik fel minden esetben a Class Designer komponenst a telepítés során. Ha nem lehet Class Diagram-ot felvenni a Visual Studio projektbe (mert a Class Diagram nem szerepel a listában az Add New Item parancs során megjelenő ablak listájában – erről a jelen útmutató későbbi fejezetében bővebben), akkor a Class Diagram komponenst utólag kell telepíteni:

- Visual Studio telepítő indítása (pl. a Windows Start Menuban a „Visual Studio Installer” begépelésével).
- A megjelenő ablakban baloldalt a „Visual Studio Extension Development” workload kártya pipálása
- Jobboldalt a Summary nézetben a „Visual Studio Extension Development” kategória lenyitása, alatta a „Class Designer” komponens pipálása, majd a Modify gombbal a konfiguráció mentése.



Feladat 1 – „Hello world” .NET konzol alkalmazás elkészítése

A feladat egy olyan C# nyelvű konzol alkalmazás elkészítése, amely a konzolra kiírja a „Hello world!” szöveget.

Az alkalmazást C# nyelven készítjük el. A lefordított alkalmazás futtatását a .NET Framework végzi. A fordítás/futtatás elméleti hátterét, valamint a .NET Framework alapjait az első előadás ismerteti.

A solution létrehozásának lépései:

1. Indítsuk el a Visual Studio-t
2. Hozzunk létre egy új projektet
 - a. File/New-Project, új projekt varázsló felugrik
 - b. A bal oldali kategória kiválasztóban: Visual C# kiválasztása
 - c. A középső listában: Console Application kiválasztása
 - d. Name (projekt név): Hello World
 - e. Location: a laborokban a d:\users\<sajátnév>\ mappába dolgozzunk, ehhez van írási jogunk.

A projekttel egy új solution is létrejön, mely struktúrája a Visual Studio Solution Explorer ablakában tekinthető át. Egy solution több projectből állhat, egy project pedig több fájlból. A solution a teljes munkakörnyezetet fogja össze (egy .sln

kiterjesztésű fájl tartozik hozzá), míg egy projekt kimenete egy .exe vagy .dll fájl jellemzően, vagyis egy összetett alkalmazás/rendszer egy komponensét állítja elő. A projektfájlok kiterjesztése C# alkalmazások esetén .csproj.

A program.cs fájlt módosítsuk a következőnek megfelelően:

```
using System;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello world!");
            Console.ReadKey();
        }
    }
}
```

Futtassuk az alkalmazást (pl. az F5 billentyű használatával). A kód felépítése nagyon hasonlít a Java-hoz, illetve a C++-hoz. Az osztályaink névterekbe szervezettek. Névtereket hatókörbe „hozni” a `using` kulcsszóval tudjuk. Névtérrel definiálni a `namespace` kulcsszóval tudunk.

Egy közönséges C# alkalmazásban az alkalmazásunk belépési pontját egy statikus `Main` nevű függvény megírásával adjuk meg. Az osztályunk neve bármi lehet, a VS egy `Program` nevű osztályt generált esetünkben. A `Main` függvény paraméterlistája kötött: vagy ne adjunk meg paramétereket, vagy egy `string[]`-öt adjunk meg, amiben futás közben megkapjuk az parancssori argumentumokat.

.NET-ben a standard ki és bemenet kezelésére a `System` névtér `Console` osztálya használandó. A `WriteLine` statikus műveletével egy sort tudunk kiírni, a `ReadKey` művelettel egy billentyű lenyomására várakozhatunk.

Megjegyzés

A félév során a programozási feladatok megvalósítása során találkozhatunk „**inconsistent visibility**”-re vagy „**inconsistent accessibility**”-re panaszkodó fordítási hibaüzenetekkel. A jelenség hátterében az áll, hogy .NET környezetben lehetőség van az egyes típusok (osztály, interfész, stb.) láthatóságának szabályozására:

- `internal` vagy nem adjuk meg: a típus csak az adott szerelvényen (.exe, .dll) belül látható
- `public`: a típus más szerelvények számára is látható

A hiba legegyszerűbben úgy hárítható el, ha minden típusunkat publikusnak definiáljuk, pl.:

```
public class HardDisk
{ ... }
```

Az UML és a kód kapcsolatának elmélete [hallgató]*

A fejezet nem tartalmaz feladatot, a hallgatók számára ismerteti a kapcsoló elméletet.

Bevezető

A fejezet egy rövid, vázlatos áttekintést ad az UML osztálydiagram és a forráskód közötti leképezés alapjairól, a megelőző félévben Szoftvertechnológia tárgyból már tanultak ismétléseként.

Napjainkban számos szoftverfejlesztési módszertan létezik. Ezek különböző mértékben építenek arra, illetve követelik meg, hogy a szoftver elkészítése során modellezést alkalmazzunk. Az azonban kétségtelen, hogy még a legagilisebb, leginkább „kódcentrikus” szemléletmódok követői is hasznosnak ítélik a szoftver fontosabb/komplexebb komponenseinek és szerkezeti elemeinek vizuális modellezését annak grafikus voltából adódó nagyobb kifejező ereje miatt.

Tegyük fel, hogy feladatunk egy alkalmazás, vagy annak adott moduljának elkészítése. A választott módszertanunkat követve – jó eséllyel több iterációban – a követelmény elemzés, analízis, tervezés, implementáció és tesztelés lépéseit fogjuk érinteni. Koncentráljunk most a tervezési fázisra. Ennek során elkészül a rendszer (legalábbis bizonyos részeinek) részletes terve, mely kimenete a részletes/ implementációs terv, illetve modell. Ezen a szinten a modellben szereplő bizonyos elemek (pl. osztályok) egyértelműen leképezhetők az adott alrendszer implementációjául választott programozási nyelv elemeire. Ha jó a fejlesztő/modellező eszközünk, akkor az le tudja generálni az osztályok vázát (pl. C++, Java, C# osztályok). A feladatunk ezt követően a generált kódban szereplő a metódusok törzsének kitöltése.

Fogalmak

- Forward engineering: modellből kód generálása. A részletes tervből a modellező eszköz le tudja generálni a programvázat. Előnye, hogy kevesebbet kell kódolni.
- Reverse engineering: kódból modell generálása. A már kész kód megértését segíti.
- Round-trip engineering: az előző kettő együttes alkalmazása. A lényeg: a modell és a kód végig szinkronban van. Ha a kódban változtatunk, a változás megjelenik a modellben, ha a modellben változtatunk, a változás megjelenik a kódban.

Ahhoz, hogy a kódgenerálás előnyeivel élni tudjunk, a következőkkel kell tisztában legyünk: ismernünk kell, hogy az adott modellező eszköz az egyes modell elemeket hogyan képezi le az adott programozási nyelv elemeire. A leképezés függ a nyelvtől és a modellező eszköztől is, nincs rá univerzális szabvány. A leképezések általában maguktól értetődőek, túl nagy eltérés nem szokott lenni.

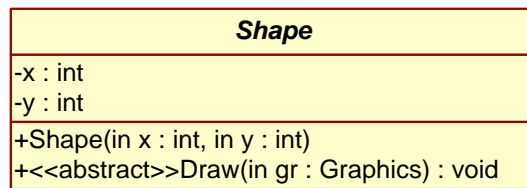
A következőkben azt tekintjük át, hogy az UML osztálydiagram egyes modellelemei hogyan képződnek le forráskódra, és viszont.

Osztályok leképezése

Mondhatni triviálisan egyszerű.

- UML osztály -> osztály
- UML attribútum -> tagváltozó
- UML művelet -> művelet/metódus

Egy C# példa:



, mely a következő kódnak felel meg:

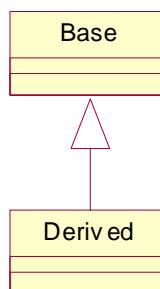
```
public abstract class Shape
{
    private int x;
    private int y;
    public Shape(int x, int y) { this.x = x; this.y = y; }
    public abstract void Draw(Graphics gr);
}
```

A láthatóság kapcsán a leképezés:

- +: public
- -: private
- #: protected

Ennél izgalmasabb kérdéskör, hogy milyen módon történik az osztályok közötti kapcsolatok leképezése, ezt a következő fejezetek ismertetik.

I. Általánosítás, specializáció kapcsolat



C# leképezés:

```
public class Base
{ };
public class Derived : Base
{ };
```

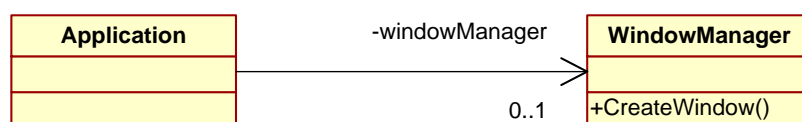
II. Asszociáció

Ez a kapcsolattípus mindig kommunikációt jelent az osztályok objektumai között. Egy adott osztály igénybe veszi egy másik osztály szolgáltatásait.

A) Leképezés 0..1 multiplicitású asszociációs kapcsolat esetén.

Ebben az esetben egy pointert vagy referenciát tartalmaz a kliens osztály, melyen keresztül igénybe tudja venni a célosztály szolgáltatásait (meg tudja hívni annak műveleteit).

Példa:



C++ leképezés:

```

class Application
{
    WindowManager* windowManager;
};

class WindowManager
{
};
  
```

C# leképezés (nincsenek pointetek, csak referenciák):

```

class Application
{
    WindowManager windowManager;
};

class WindowManager
{
};
  
```

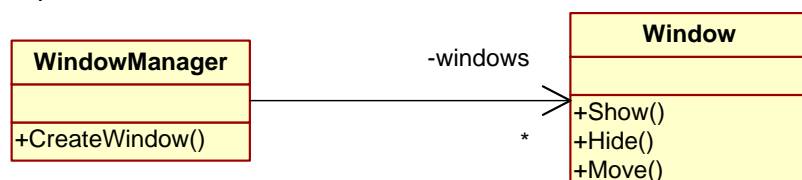
Minkét esetben azt látjuk, hogy a **kliens osztályba felvesszünk egy pointer vagy referencia tagváltozót, melynek típusa megegyezik az asszociációban hivatkozott célosztály típusával, illetve a tagváltozó neve az asszociációs kapcsolatra a célosztályra megadott szereppel (role), ami a példában a windowManager.**

A leképezés logikus, hiszen a kliens ezen pointeren/referencián keresztül tudja a célosztály metódusait meghívni.

Megjegyzés. Előfordulhat, hogy az asszociáció kétirányú, mindkét osztály igénybeveszi a másik szolgáltatásait. Ilyenkor általában nem tesszük ki az asszociáció mindkét végére a nyilat, hanem mindkét végéről elhagyjuk azt. Ilyen kétirányú kapcsolat esetén a szerepet (role) a kapcsolat mindkét végén meg kell adni. A leképezés során mindkét osztályba felvesszünk egy pointert/referenciát a másikra.

B) Leképezés 0..n multiplicitású asszociációs kapcsolat esetén

Ebben az esetben egy kliensoldali objektum több céloldali objektummal van kapcsolatban. Példa:



Egy WindowManager objektum több Window objektumot menedzsel. **A leképezés során a kliens osztályba a célosztálybeli objektumok valamilyen gyűjteményét vesszük fel.** Ez lehet tömb, lista, stb., ami a célunknak az adott helyzetben leginkább megfelel.

Egy leképzési lehetőség a fenti példára C++ nyelven:

```
class WindowManager
{
    vector<Window*> windows;
};
```

Illetve C# nyelven:

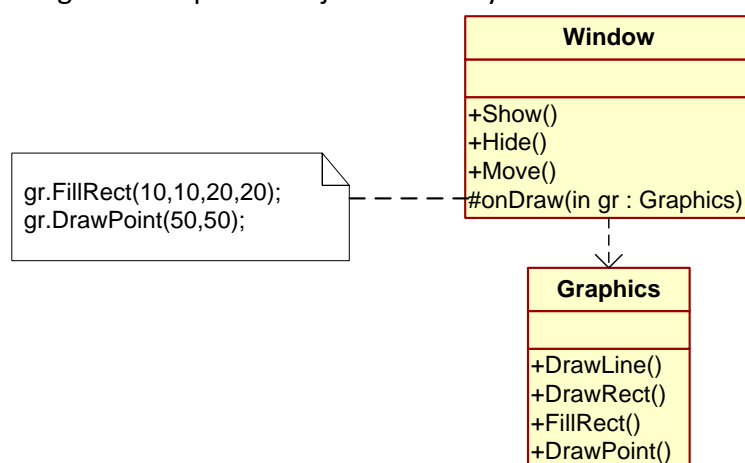
```
class WindowManager
{
    List<Window> windows;
};
```

III. Aggregáció (tartalmazás, rész-egész viszony)

Általában a leképezése pontosan úgy történik, mint az asszociáció esetében.

IV. Függőség (dependency)

A leglazább kapcsolatot jelenti osztályok között. Példa:



A jelentése: a `Window` osztály függ a `Graphics` osztálytól. Vagyis, ha a `Graphics` osztály megváltozik, akkor lehet, hogy a `Window` osztályt is meg kell változtatni.

Ezt a kapcsolattípust akkor szoktuk használni, ha a függőségi kapcsolat elején levő osztály metódusai paraméterlistájában/visszatérési értékében szerepel a kapcsolat végén levő osztály. A példában a `Window` osztály `onDraw` művelete paraméterként megkapja a `Graphics` osztály egy objektumát, így függ tőle, hiszen a metódus törzsében így meg tudja hívni a `Graphics` osztály metódusait. Ha pl. a `Graphics` osztály `FillRect` metódusának nevét megváltoztatjuk, akkor ezt a változást át kell vezetni a hívások helyén, vagyis a `Window` osztály `onDraw` metódusának törzsében is.

Interfész és absztrakt (ős)osztály [hallgató]*

A fogalmak korábbi tárgyak keretében már ismertetésre kerültek, így most csak a legfontosabbakat foglaljuk össze, illetve a C# vonatkozására térünk ki.

Absztrakt osztály

Olyan osztály, mely nem példányosítható. C# nyelven az osztálydefinícióban az `abstract` kulcsszót kell kiírni, pl.:

```
abstract class Shape { ... }
```

Absztrakt osztályoknak lehetnek absztrakt metódusaik, melyeknek nem adjuk meg a törzsét, ezeknél is az `abstract` kulcsszót kell használni:

```
...  
abstract void Draw();  
...
```

Absztrakt osztályok használatának két célja lehet:

- Egy osztályhierarchiában a leszármazottakra közös kódot fel tudjuk vinni egy absztrakt közös ősbbe, így elkerüljük a kódduplikációt.
- Egységesen tudjuk absztrakt ősként hivatkozva a leszármazottakat kezelni (pl. heterogén kollekciók).

.NET környezetben, csakúgy, mint Java nyelven, egy osztálynak csak egy ősosztálya lehet.

Interfész

Az interfész nem más, mint egy művelethalmaz. Tulajdonképpen egy olyan absztrakt osztálynak felel meg, melynek minden művelete absztrakt.

C# nyelven az `interface` kulcsszóval tudunk interfészt definiálni:

```
public interface ISerializable  
{  
    void WriteToStream(Stream s);  
    void LoadFromStream(Stream s);  
}  
  
public interface IComparable  
{  
    int CompareTo(Object obj);  
}
```

Míg egy osztálynak csak egy őse lehet, akárhány interfészt implementálhat:

```
public class Rect : Shape, ISerializable, IComparable  
{  
    ...  
}
```

Ebben a példában `Rect` osztály a `Shape` osztályból származik, valamint az `ISerializable` és `IComparable` interfészeket implementálja (kötelezően az ősosztályt kell először megadni). Az interfészt implementáló osztályban annak valamennyi műveletét meg kell valósítani, vagyis meg kell írni a törzsét (kivéve azt a ritka esetet, amikor absztrakt művelettel valósítjuk meg).

Interfészek használatának egy fő célja van. Interfészként hivatkozva egységesen tudjuk az interfészt implementáló valamennyi osztályt kezelni (pl. heterogén kollekció). Ennek egy következménye: az **interfészek lehetővé teszik széles körben használható osztályok és függvények megírását**. Pl. tudunk írni egy univerzális `Sort` sorrendező függvényt, mely bármilyen osztállyal használható, mely implementálja az `IComparable` interfészt.

Az interfész alkalmazásának előnyei még:

- A kliensnek elég a kiszolgáló objektum interfészét ismernie, így egyszerűen tudja a kiszolgálót használni.
- **Ha a kliens csak az interfészen keresztül használja a kiszolgálót, így a kiszolgáló belső implementációja megváltozhat, a klienst nem kell módosítani (újra sem**

kell fordítani). Ennek megfelelően az interfész egy szerződés is a kiszolgáló és a kliens között: amíg a kiszolgáló *garantálja* az interfész támogatását, a klienst nem kell változtatni.

Absztrakt ős és interfész összehasonlítása

Az absztrakt ős előnye az interfésszel szemben, hogy adhatunk meg a műveletekre vonatkozóan alapértelmezett implementációt, illetve vehetünk fel tagváltozókat.

Az interfészek előnye az absztrakt ősrel szemben, hogy egy osztály akárhány interfészt implementálhat, míg ős maximum egy lehet.

Az interfészek használatának van még egy következménye, ami bizonyos esetben kellemetlenségeket okozhat. **Amikor az interfészbe új műveletet veszünk fel, akkor valamennyi implementáló osztályt szintén bővíteni kell, különben a kód nem fordul.** Absztrakt ős bővítése esetén ez nincs így: amennyiben új műveletet veszünk fel, lehetőségünk van azt virtuális függvényként felvenni, és így az ősben alapértelmezett implementációt adni rá. Ez esetben az leszármazottak igény szerint tudják ezt felüldefiniálni, erre nincsenek rákényszerítve. Az interfészek ezen tulajdonsága különösen osztálykönyvtárak/keretrendszerek esetén lehet kellemetlen. Tegyük fel, hogy a .NET Framework új verziójának kiadásakor a keretrendszer egyik interfészébe új műveletet vesznek fel. Ekkor valamennyi alkalmazásban valamennyi implementáló osztályt módosítani kell, különben nem fordul a kód. Ezt kétféleképpen lehet elkerülni. Vagy ősosztály használatával, vagy ha mégis interfészt kellene bővíteni, akkor inkább új interfészt bevezetésével, amely már az új műveletet is tartalmazza. Bár itt az első megközelítés (ősosztály alkalmazása) tűnik első érzésre vonzóbbnak, ennek is van hátránya: ha az alkalmazás fejlesztésekor egy keretrendszerbeli ősből származtatunk, akkor osztályunknak már nem lehet más ős, és ez bizony sok esetben fájdalmas megkötést jelent.

Mivel mind az interfészek, mind az absztrakt ősosztályok alkalmazása járhat negatív következményekkel is, **számos esetben a kettő együttes használatával tudjuk kihozni megoldásunkból a maximumot** (vagyis lesz a kódunk könnyen bővíthető úgy, hogy nem, vagy csak minimális mértékben tartalmaz kódDuplikációt).

Feladat 2 - Az UML és a kód kapcsolatának szemléltetése

Feladat: Egy számítógépalkatrész nyilvántató alkalmazás kifejlesztésével bíztak meg bennünket. Bővebben:

- Különböző típusú alkatrészeket kell tudni kezelni. Kezdetben a `HardDisk`, `SoundCard` és `LedDisplay` típusokat kell támogatni, de a rendszer legyen könnyen bővíthető új típusokkal.
- Az alkatrészekhez tartozó adatok: beszerzés éve, életkora (számított), beszerzési ára és aktuális ára (számított), de ezeken felül típusfüggő adatokat is tartalmazhatnak (pl. a `HardDisk` esetében a kapacitás).
- Az aktuális ár függ az alkatrész típusától, a beszerzési ártól és az alkatrész gyártási évétől. Pl. minél öregebb egy alkatrész, annál nagyobb kedvezményt adunk rá, de a kedvezmény mértéke függ az alkatrész típustól is.
- Listázni kell tudni a készleten levő alkatrészeket.

- Az `LedDisplay` osztálynak kötelezően egy `DisplayBase` osztályból kell származnia, és a `DisplayBase` osztály forráskódja nem megváltoztatható. Jelen példában ennek nincs sok értelme, a gyakorlatban azonban gyakran találkozunk hasonló helyzettel, amikor is az általunk használt keretrendszer/platform előírja, hogy adott esetben egy-egy beépített osztályból kell származtassunk. Tipikusan ez a helyzet, amikor ablakokkal, űrlapokkal, saját vezérlőtípusokkal dolgozunk: ezeket a keretrendszer beépített osztályaiból kell származtatnunk, és a keretrendszer - pl. Java, .NET - forráskódja nem áll rendelkezésünkre (de legalábbis biztosan nem akarjuk megváltoztatni). A példánkban a `DisplayBase`-ből való származtatás előírásával ezt a helyzetet szimuláljuk.

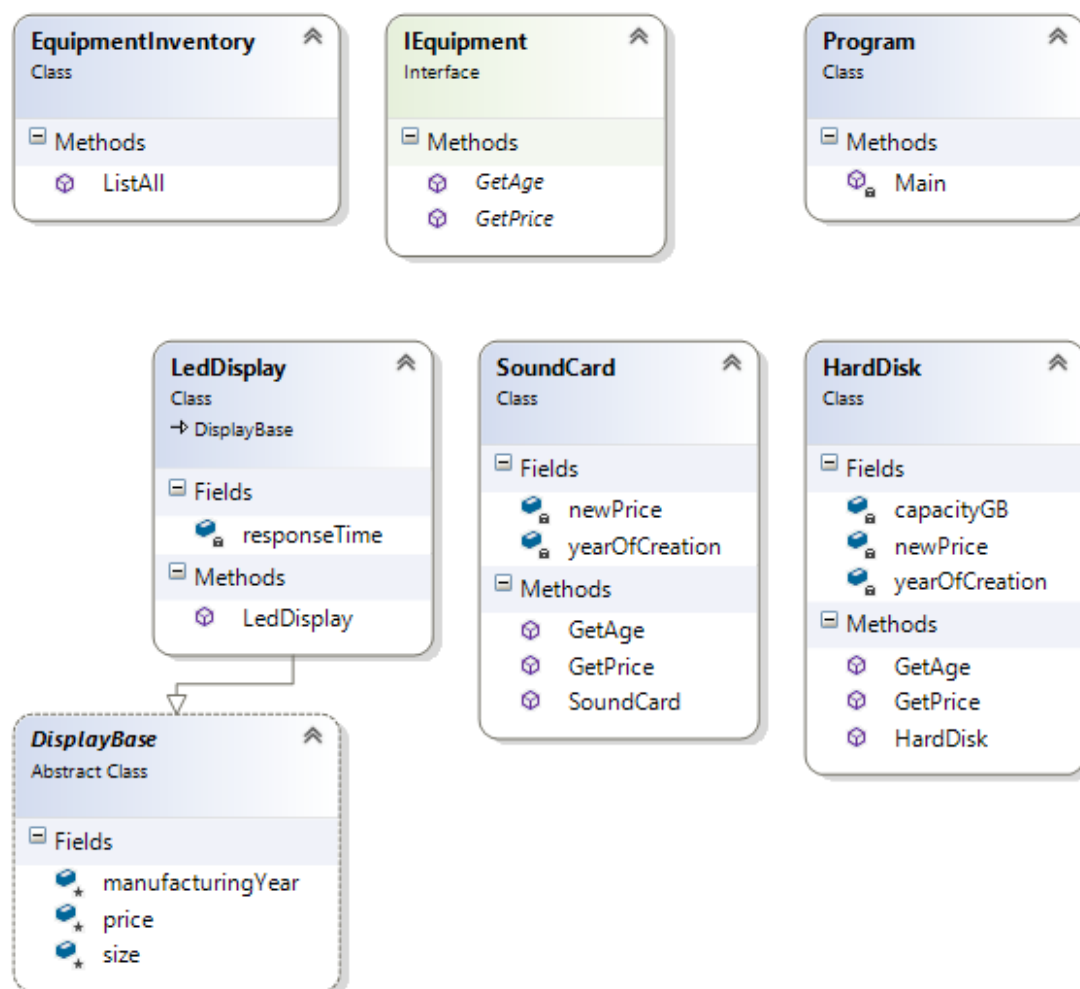
A megvalósítás során jelentős egyszerűsítéssel élünk: az alkatrészeket csak memóriában tarjuk nyilván, a listázás is a lehető legegyszerűbb, egyszerűen csak kiírjuk a nyilvántartott alkatrészek adatait a konzolra.

A kezdeti egyeztetések során a megrendelőnkől a következő információt kapjuk: egy belső munkatársuk már elindult a fejlesztéssel, de idő hiányában csak félkész megoldásig jutott. A feladatunk részét képezi a félkész megoldás megismerése, illetve ebből kiindulva kell a feladatot megvalósítani.

Nyissuk meg a megrendelőnkől kapott forráskód solution-jét (EquipmentPélda – Kiindulás mappa) Visual Studio alatt (EquipmentInventory.sln). A Solution Explorerben szemmel fussuk át a fájlokat. Az megértést segítené, ha egy osztálydiagramon megjelenítenénk az osztályok közötti kapcsolatokat. Vegyünk is fel egy osztálydiagramot a projektünkbe. A Solution Explorerben a projekten (és nem a solutionön!) jobb gombbal kattintva a felugró menüben az *Add/New Item ...* elemet választva, majd a megjelenő ablakban a *Class Diagram* elemet válasszuk ki, az ablak alján a diagram nevének a *Main.cd*-t adjuk meg, és OK-zuk le az ablakot.

Megjegyzés: ha a *Class Diagram* elem nem jelenik meg a listában, akkor nincs telepítve a VS megfelelő komponense. Erről jelen dokumentum Bevezető fejezetében olvashatsz bővebben.

Ekkor a Solution Explorerben megjelenik a *Main.cd* diagramfájl, duplakattintással nyissuk meg. A diagramunk jelenleg üres. A Solution Explorerből drag&droppal dobjuk rá a .cs forrásfájlokat a diagramra. Ekkor a VS megnézi, milyen osztályok vannak ezekben a forrásfájlokban, és visszafejti őket UML osztályokká. Alakítsuk ki a következő ábrának megfelelő elrendezést (az osztályok tagjainak megjelenítését a téglalapuk jobb felső sarkában levő duplanyíllra kattintással érhetjük el):



Az osztályokhoz tartozó forráskódot is megnézhetjük, akár a digramon a megfelelő osztályra duplán kattintva, akár a Solution Explorerből a .cs fájlokat megnyitva.

A következőket tapasztaljuk:

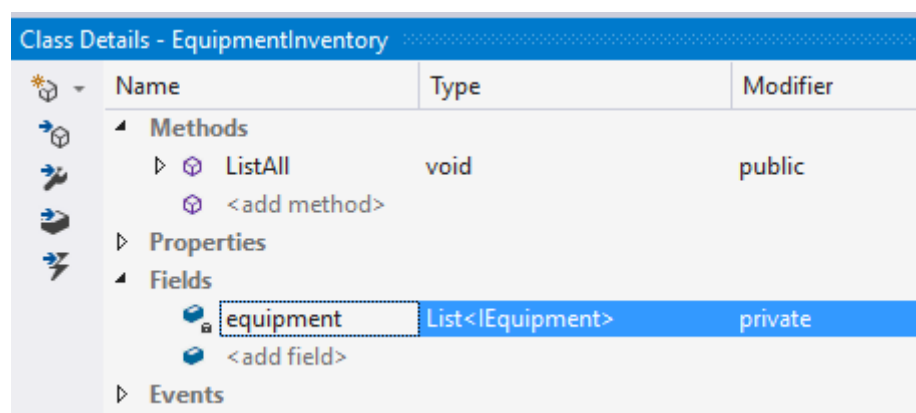
- Az `SoundCard`, `HardDisk` és `LedDisplay` osztályok viszonylag jól kidolgozottak, rendelkeznek a szükséges attribútumokkal és lekérdező függvényekkel.
- Az `LedDisplay` a követelményeknek megfelelően a `DisplayBase` osztályból származik.
- Az `EquipmentInventory` felelős ugyan a készleten levő alkatrészek nyilvántartásáért, de gyakorlatilag semmi nincs ebből megvalósítva.
- Találunk egy `IEquipment` interfészt, `GetAge` és `GetPrice` műveletekkel.

Álljunk neki a megoldás kidolgozásának. Először is az alapkoncepciókat fektessük le. Az `EquipmentInventory` osztályban egy heterogén kollekciónban tároljuk a különböző alkatrész típusokat. Ez a kulcsa az alkatrészek egységes kezelésének, vagyis annak, hogy a megoldásunk új alkatrésztípusokkal könnyen bővíthető legyen.

Mint korábban taglaltuk, az egységes kezelést vagy közös ősosztály, vagy közös interfész bevezetésével lehet megoldani. Esetünkben a közös ősosztály (pl. `EquipmentBase`) úgy tűnik, kiesik, mert ennek bevezetésével az `LedDisplay` osztálynak két ősosztálya is lenne: a kötelezőnek kikötött `DisplayBase`, és az általunk az egységes kezelésre bevezetett `EquipmentBase`. Ez nem lehetséges, .NET

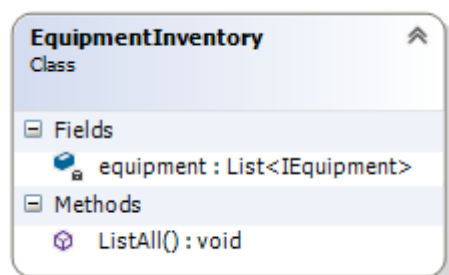
környezetben egy osztálynak csak egy őse lehet. Az a megoldás pedig, hogy a `DisplayBase`-t úgy módosítjuk, hogy ő is az `EquipmentBase`-ből származik, a követelményünknek megfelelően nem lehetséges (kikötés volt, hogy a forráskódja nem módosítható). Marad tehát az interfész alapú megközelítés. Minden bizonnyal az alkalmazás korábbi fejlesztője is erre a következtetésre jutott, ezért is vezette be az `IEquipment` interfészt.

Vegyünk fel egy `IEquipment` típusú elemekből álló generikus listát (ne property-t hanem field-et!) az `EquipmentInventory` osztályba. A láthatósága – az egységbezárásra törekedve – legyen `private`. A neve legyen `equipment` (ne legyen „s” a végén, angolban az `equipment` többes száma is `equipment`). A tagváltozó felvételéhez a Visual Studio *Class Details* ablakát használjuk. Ha az ablak nem látható, a *View/Other Windows/Class Details* menü kiválasztásával jeleníthető meg.



A tagváltozó típusa tehát `List<IEquipment>`. A .NET `List` típusa egy dinamikusan nyújtózkodó generikus tömb (mint Java-ban az `ArrayList`).

A diagramon az `EquipmentInventory` osztályra pillantva azt látjuk, hogy csak a tagváltozó neve jelenik meg, a típusa nem. A diagram hátterén jobb gombbal kattintva a *Change Members Format* menüből a *Display Full Signature*-t választjuk ki. Ezt követően a diagramon láthatóvá válik a tagváltozók típusa, valamint a műveletek teljes szignatúrája.

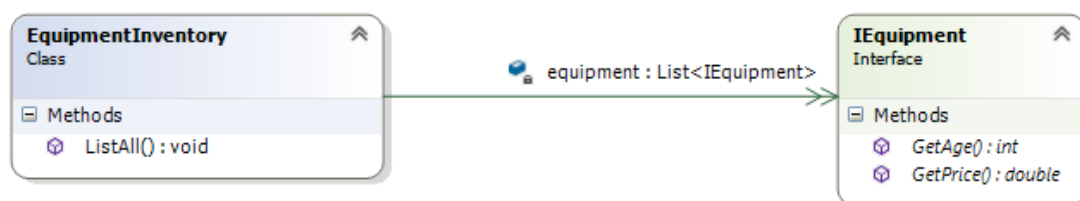


Az `EquipmentInventory` osztályon duplán kattintva elnavigálhatunk a forráskódba, és mint látható, valóban egy lista típusú tagváltozóként jelenik meg a kódban:

```
class EquipmentInventory
{
    private List<IEquipment> equipment;
```

Ennek egyrészt örülünk, mert a Visual Studio támogatja a round-trip engineering technikát: a modellbeli változásokat azonnal átvezeti a kódba, és viszont. Másrészt a

korábbiakban azt taglaltuk, hogy ha egy osztályban egy gyűjtemény tag van egy másik osztály elemeiből, akkor annak az UML modellben egy 1-több típusú asszociációs kapcsolatként „illik” megjelennie a két osztály között. A modellünkben egyelőre nem ezt tapasztaljuk. Szerencsére a VS modellező felülete rávehető, hogy ilyen formában jelenítse meg ezt a kapcsolattípust. Ehhez kattintsunk a diagramon jobb gombbal az equipment tagváltozón, és a menüből válasszuk ki a *Show as Collection Association* elemet. Az IEquipment osztályt ezt követően mozgassuk ki jobbra, hogy kellő hely legyen a diagramon az asszociációs kapcsolat és a kapcsolaton levő szerep (role) adatainak megjelenítésére:



A dupla nyíl végződés a „többes” oldalon nem szabványos UML, de ne szomorodjunk el tőle különösebben, nincs semmi jelentősége. Annak mindenképpen örülünk, hogy a kapcsolatot reprezentáló nyíl IEquipment végén a szerepben a tagváltozó neve (sőt, még a pontos típusa is) fel van tüntetve.

Navigáljunk el az EquipmentInventory forráskódjához, és írjuk meg a konstruktorát, ami inicializálja az equipment gyűjteményt!

```

public EquipmentInventory()
{
    this.equipment = new List<IEquipment>();
}
  
```

Ezután írjuk meg a ListAll metódust, ami kiírja az elemek életkorát, és az aktuális értéküket:

```

public void ListAll()
{
    foreach (IEquipment eq in equipment)
    {
        Console.WriteLine( "Életkor: {0}\tÉrtéke: {1}",
            eq.GetAge(), eq.GetPrice() );
    }
}
  
```

Az elemeken a `foreach` utasítással iterálunk végig. A `foreach` utasítás használata során az `in` kulcsszó után egy gyűjteménynek kell állnia, az `in` előtt pedig egy változó deklarációnak (esetünkben `IEquipment eq`), ahol a típus a gyűjtemény elemtípusa. Minden iterációban ez a változó a gyűjtemény iterációbeli értékét veszi fel.

A `Console.WriteLine` műveletnek vagy egy egyszerű sztringet adunk meg, vagy, mint esetünkben, egy formázási sztringet, és az abba behelyettesítendő értékeket. A behelyettesítendő értékekre a formázási sztringben `{<sorszám>}` formában tudunk hivatkozni, ahol a sorszám a behelyettesítendő paraméter pozíciója az argumentumlistában.

Írjunk meg egy `AddEquipment` nevű függvényt, ami felvesz egy új eszközt a készletbe:

```

public void AddEquipment(IEquipment eq)
{
  
```

```
equipment.Add(eq);
}
```

Korábbi döntésünk értelmében az `IEquipment` interfészt használjuk az különböző alkatrész típusok egységes kezelésére. Estünkben mind a `SoundCard`, mind a `HardDisk` osztály rendelkezik `GetAge()` és `GetPrice()` metódussal, mégsem tudjuk őket egységesen kezelni (pl. közös listában tárolni). Ahhoz, hogy ezt meg tudjuk tenni, el kell érünk, hogy mindkét osztály megvalósítsa az `IEquipment` interfészt. Módosítsuk a forrásukat:

```
public class SoundCard : IEquipment
{
    ...
}
```

```
public class HardDisk : IEquipment
{
    ...
}
```

Ezt követően a `SoundCard` és `HardDisk` osztályban implementálnunk kell az `IEquipment` interfészben levő metódusokat. Azt tapasztaljuk, hogy ezzel nincs most teendők, a `GetPrice` és `GetAge` függvények már meg vannak írva mindkét helyen.

Próbaképpen a `Program.cs` fájlban található `Main` függvényünkben hozzunk létre egy `EquipmentInventory` objektumot, töltsük fel `HardDisk` és `SoundCard` objektumokkal, majd listázzuk a készletet a konzolra. Amennyiben nem 2016 az aktuális év, az alábbi soroknál a 2016-os évet írjuk át az aktuális évre, a 2015-öt pedig ennél eggyel kisebb számra!

```
static void Main( string[] args )
{
    EquipmentInventory ei = new EquipmentInventory();

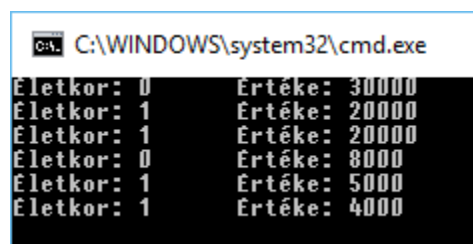
    ei.AddEquipment(new HardDisk(2016, 30000, 80));
    ei.AddEquipment(new HardDisk(2015, 25000, 120));
    ei.AddEquipment(new HardDisk(2015, 25000, 250));

    ei.AddEquipment(new SoundCard(2016, 8000));
    ei.AddEquipment(new SoundCard(2015, 7000));
    ei.AddEquipment(new SoundCard(2015, 6000));

    ei.ListAll();

    Console.ReadKey();
}
```

Az alkalmazást futtatva azt tapasztaljuk, hogy bár megoldásunk kezdetleges, de működik:



```
C:\WINDOWS\system32\cmd.exe
Életkor: 0      Értéke: 30000
Életkor: 1      Értéke: 20000
Életkor: 1      Értéke: 20000
Életkor: 0      Értéke: 8000
Életkor: 1      Értéke: 5000
Életkor: 1      Értéke: 4000
```


Folytassuk a munkát az `LedDisplay` osztállyal. A `DisplayBase` és forráskódját a követelmények miatt nem módosíthatjuk. De ez semmiféle problémát nem okoz, az `LedDisplay` osztályunk fogja az `IEquipment` interfészt implementálni, módosítsuk a kódot ennek megfelelően:

```
public class LedDisplay : DisplayBase, IEquipment
```

Az `LedDisplay` osztályban már meg kell írni az interfészben szereplő függvényeket:

```
public double GetPrice()
{
    return this.price;
}

public int GetAge()
{
    return DateTime.Today.Year - this.manufacturingYear;
}
```

Bővítsük a `Main` függvényünket is, vegyünk fel két `LedDisplay` objektumot a készletünkbe (itt is él, hogy amennyiben nem 2016 az aktuális év, az alábbi soroknál a 2016-os évet írjuk át az aktuális évre, a 2015-öt pedig ennél eggyel kisebb számra!

```
ei.AddEquipment( new LedDisplay( 2015, 80000, 17, 16) );
ei.AddEquipment( new LedDisplay ( 2016, 70000, 17, 12) );

ei.ListAll();
Console.ReadKey();
```

Tesztelésképpen futtassuk az alkalmazást.

Feladat 3 - Az interfész és az absztrakt ősosztály alkalmazástechnikája

Értékeljük a jelenlegi, interfész alapú megoldásunkat.

Az egyik fő probléma, hogy kódunk tele van a karbantarthatóságot és bővíthetőséget romboló kódduplikációval:

- A `yearOfCreation` és `newPrice` tagok minden alkatrész típusban (kivéve a speciális `LedDisplay`-t) közősek, ezeket új típus bevezetésekor is copy-paste technikával át kell venni.
- A `GetAge` függvény implementációja szinten minden alkatrész típusban (kivéve a speciális `LedDisplay`-t) azonos, szintén copy-paste-tel „szaporítandó”.
- A konstruktorok `yearOfCreation` és `newPrice` tagokat inicializáló sorai szintén duplikáltak az egyes osztályokban.

Bár ez a kódduplikáció egyelőre nem tűnik jelentősnek, új alkatrész típusok bevezetésével egyre inkább elmérgesedik a helyzet, jobb időben elejét venni a jövőbeli fájdalomnak.

A másik probléma abból adódik, hogy az alkatrész adatok listázása jelenleg fájdalmasan hiányos, nem jelenik meg az alkatrész típusa (csak a kora és az ára). A típus megjelenítéséhez az `IEquipment` interfészt bővíteni kell, pl. egy

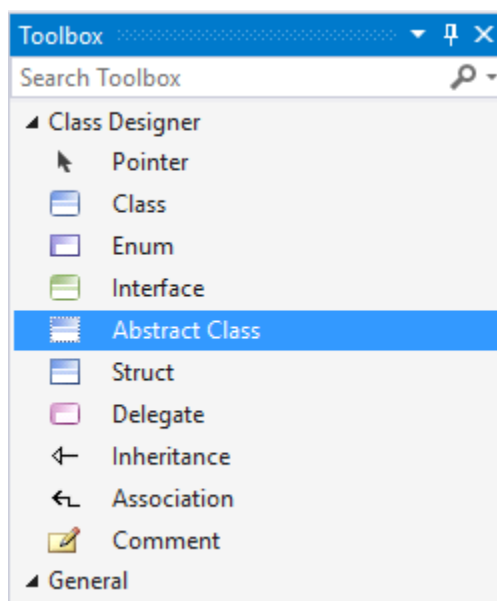
`getDescription` nevű művelet bevezetésével. Vegyünk is fel egy `getDescription` függvényt az interfészbe!

```
public interface IEquipment
{
    double GetPrice();
    int GetAge();
    string GetDescription();
}
```

Ekkor minden `IEquipment` interfészt implementáló osztályban meg kellene valósítani ezt a metódust is, ami sok osztály esetén sok munka (valamint egy többkomponensű, vagyis több DLL-ből álló alkalmazás esetében, amikor ezek nem egy fejlesztő cég kezében vannak, sokszor nem is megoldható). A `build` parancs futtatásával ellenőrizzük, hogy a `getDescription` felvétele után három helyen is fordítási hibát kapunk.

Mindkét problémára megoldást jelent egy közös absztrakt ős bevezetése (kivéve az `LedDisplay` osztályt, amire még visszatérünk). Ebbe fel tudjuk költöztetni a leszármazottakra közös kódot, valamint az újonnan bevezetett `getDescription` művelethez egy alapértelmezett implementációt tudunk megadni. Legyen az új absztrakt ősosztályunk neve `EquipmentBase`. Kérdés, szükség van-e a továbbiakban az `IEquipment` interfészre, vagy az teljesen kiváltható az új `EquipmentBase` osztállyal. Az `IEquipment` interfészt meg kell tartasuk, mert az `LedDisplay` osztályunkat nem tudjuk az `EquipmentBase`-ből származtatni: már van egy kötelezően előírt ősosztálya, a `DisplayBase`: emiatt az `EquipmentInventory` a továbbfejlesztett megoldásunkban is `IEquipment` interfészként hivatkozik az különböző alkatrészekre.

Álljunk is neki az átalakításnak. Legyen az osztálydiagramunk az aktív tabfűl. A Toolbox-ból drag&drop-pal dobjunk fel egy *Abstract Class* elemet a diagramra, a neve legyen `EquipmentBase`.

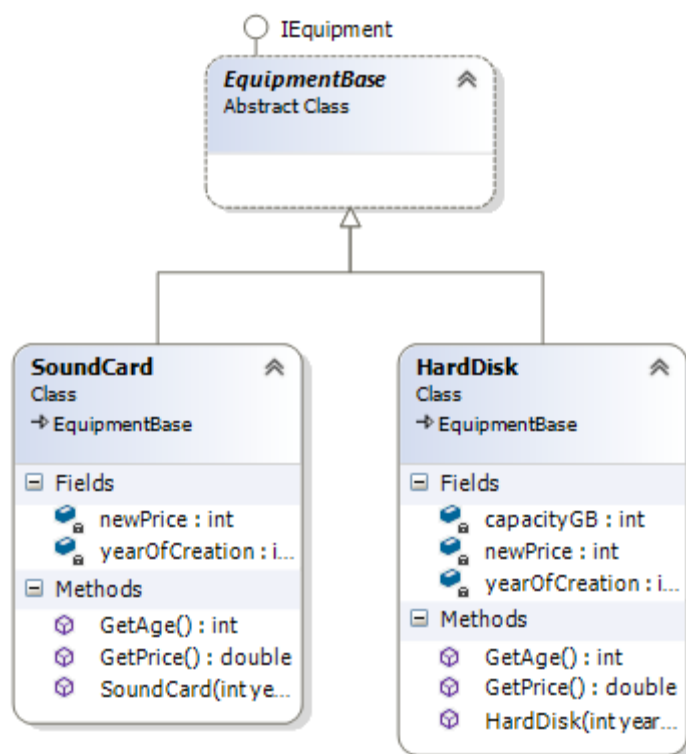


A következőkben azt kell elérjünk, hogy a `SoundCard` és a `HardDisk` osztályok származzanak az `EquipmentBase`-ből (az `LedDisplay`-nek már van másik őse, így

ott ezt nem tudjuk megtenni). Ehhez válasszuk ki az Inheritance kapcsolatot a Toolbox-ban, majd húzzunk egy-egy vonalat a gyermekosztályból kiindulva az őszosztályba a `SoundCard` és `HardDisk` esetében egyaránt.

A következő lépésben alakítsuk át úgy a kódot, hogy ne a `HardDisk` és `SoundCard` valósítsák meg külön-külön az `IEquipment` interfészt, hanem a közös ősök, az `EquipmentBase` egyszer. Ehhez módosítsuk az `EquipmentBase` osztályt úgy, hogy valósítsa meg az interfészt (akár a diagramon húzzunk be egy inheritance kapcsolatot az `EquipmentBase`-ből az `IEquipment`-be, vagy az `EquipmentBase` forráskódját módosítsuk). A `HardDisk` és `SoundCard` osztályokból töröljük az `IEquipment` megvalósítását (az ős már implementálja).

A diagramunk és a forráskódunk vonatkozó részei ezt követően így néznek ki:



```
public abstract class EquipmentBase : IEquipment
{

```

```
public class HardDisk : EquipmentBase
{ ...

```

```
public class SoundCard : EquipmentBase
{ ...

```

A kódunk még nem fordul, ennek több oka is van. Az `EquipmentBase` implementálja az `IEquipment` interfészt, de még nincsenek benne implementálva az interfész műveletei. Vagy generáltassuk le a metódusokat a smart tag használatával, vagy gépeljük be a következő elveknek megfelelően:

- A `newPrice` és `yearOfCreation` duplikálva vannak a `HardDisk` és `SoundCard` osztályokban: mozgassuk (és ne másoljuk!) át ezeket a közös `EquipmentBase` ősbbe, és `protected` láthatóságot adjunk meg.
- A `GetAge` művelet duplikálva van a `HardDisk` és `SoundCard` osztályokban, ezekből töröljük ki az implementációt és vigyük át az `EquipmentBase` osztályba.
- A `GetPrice` műveletet **absztrakt** műveletként vegyük fel az ősbbe. Ez szándékos tervezői döntés, így rákényszerítjük a leszármazott osztályokat, hogy mindenképpen definiálják felül ezt a műveletet.
- A `GetDescription` esetében viszont pont fordítottja a helyzet: ezt **virtuálisnak** definiáljuk (és nem absztraktnak), vagyis már az ősbbe is adunk meg implementációt. Így a leszármazottak nincsenek rákényszerítve a művelet felüldefiniálására.

A fentieknek megfelelő kód a következő:

```
public abstract class EquipmentBase : IEquipment
{
    protected int yearOfCreation;
    protected int newPrice;

    public int GetAge()
    {
        return DateTime.Today.Year - yearOfCreation;
    }

    public abstract double GetPrice();

    public virtual string GetDescription()
    {
        return "EquipmentBase";
    }
}
```

Néhány kiegészítő gondolat a kódrészletre vonatkozóan:

- Az absztrakt osztályok esetében az `abstract` kulcsszót ki kell írni a `class` szó elé.
- Az absztrakt műveletek esetében az `abstract` kulcszót kell megadni
- .NET környezetben lehetőségünk van szabályozni, hogy egy művelet virtuális-e vagy sem. Ebből a szempontból a C++ nyelvhez hasonlít. Amennyiben egy műveletet virtuálissá szeretnénk tenni, a `virtual` kulcsszót kell a műveletre megadni. Emlékeztető: akkor definiáljunk egy műveletet virtuálisnak, ha a leszármazottak azt felüldefiniál(hat)ják. Csak ekkor garantált, hogy egy ősrreferencián meghívva az adott műveletet a leszármazottbeli verzió hívódik meg.

A következő lépésben térjünk át az `EquipmentBase` leszármazottakra. C# nyelven az absztrakt és virtuális műveletek felüldefiniálásakor a leszármazottban meg kell adni az `override` kulcsszót. Első lépésben a `GetPrice` műveletet definiáljuk felül:

```
public class HardDisk : EquipmentBase
{
    ...
    public override double GetPrice()
```

```

    {
        return yearOfCreation < (DateTime.Today.Year - 4) ? 0 :
            newPrice - (DateTime.Today.Year - yearOfCreation) * 5000;
    }
}

```

```

public class SoundCard : EquipmentBase
{
    ...
    public override double GetPrice()
    {
        return yearOfCreation < (DateTime.Today.Year - 4) ? 0 :
            newPrice - (DateTime.Today.Year - yearOfCreation) * 2000;
    }
}

```

A következőkben lépésben a `GetDescription` műveletet írjuk meg a `HardDisk` és `SoundCard` osztályokban. Mivel itt az ősbeli virtuális függvényt definiáljuk felül, szintén meg kell adni az `override` kulcsszót:

```

public class HardDisk : EquipmentBase
{
    ...
    public override string GetDescription()
    {
        return "HardDisk";
    }
}

```

```

public class SoundCard : EquipmentBase
{
    ...
    public override string GetDescription()
    {
        return "SoundCard";
    }
}

```

Felmerülhet bennünk a kérdés, miért döntöttek úgy a C# nyelv tervezői, hogy a műveletek felüldefiniálásakor egy extra kulcsszót kelljen megadni, hasonlóra pl. a C++ nyelv esetében nem volt szükség. Az ok egyszerű: a kód így kifejezőbb. A leszámazottak kódját nézve az `override` szó azonnal egyértelművé teszi, hogy valamelyik ősből ez a művelet absztrakt vagy virtuális, nem kell valamennyi ősből a kódját ehhez áttekinteni.

Az `LedDisplay` osztályunk őse meg van kötve, annak kódja nem módosítható, így nem tudjuk az `EquipmentBase`-ből származtatni. A `GetAge` műveletet így nem tudjuk törölni, ez a kód duplikáció itt megmarad (de csak az `LedDisplay` esetében, ami csak egy osztály a sok közül!).

Megjegyzés: valójában egy kis plusz munkával ettől a duplikációtól is meg tudnánk szabadulni. Ehhez valamelyik osztályban (pl. `EquipmentBase`) fel kellene venni egy statikus segédfüggvényt, mely paraméterben megkapná a gyártási évet, és visszatérné az életkort. Az `EquipmentBase.GetAge` és az `LedDisplay.GetAge` ezt a segédfüggvényt használná kimenete előállítására.

Az `LedDisplay` osztályunkban adósak vagyunk még a `GetDescription` megírásával:

```
public class LedDisplay : DisplayBase, IEquipment
{
    ...
    public string GetDescription()
    {
        return "LedDisplay";
    }
}
```

Figyeljük meg, hogy itt NEM adtuk meg az override kulcsszót. Mikor egy interfész függvényt implementálunk, az `override`-ot nem kell/szabad kiírni.

Módosítsuk az `EquipmentInventory.ListAll` műveletét, hogy az elemek leírását is írja ki a kimenetre:

```
public void ListAll()
{
    foreach (IEquipment eq in equipment)
    {
        Console.WriteLine("Leírás: {0}\tÉletkor: {1}\tÉrtéke: {2}",
            eq.GetDescription(), eq.GetAge(), eq.GetPrice());
    }
}
```

Így már sokkal informatívabb kimenetet kapunk az alkalmazás futtatásakor:

```
C:\WINDOWS\system32\cmd.exe
Leírás: HardDisk      Életkor: 0      Értéke: 30000
Leírás: HardDisk      Életkor: 1      Értéke: 20000
Leírás: HardDisk      Életkor: 1      Értéke: 20000
Leírás: SoundCard     Életkor: 2015   Értéke: 8000
Leírás: SoundCard     Életkor: 2015   Értéke: 5000
Leírás: SoundCard     Életkor: 2015   Értéke: 4000
Leírás: LcdDisplay    Életkor: 1      Értéke: 80000
Leírás: LcdDisplay    Életkor: 0      Értéke: 70000
```

A kódunkat áttekintve még egy helyen találunk kódDuplikációt. Valamennyi `EquipmentBase` leszármazott (`HardDisk`, `SoundCard`) konstruktorában ott van ez a két sor:

```
this.yearOfCreation = yearOfCreation;
this.newPrice = newPrice;
```

Ha belegondolunk, ezek a `yearOfCreation` és `newPrice` tagok az ősből vannak definiálva, így egyébként is az ő felelőssége kellene legyen ezek inicializálása. Vegyünk is fel egy megfelelő konstruktort az `EquipmentBase`-ben:

```
public abstract class EquipmentBase : IEquipment
{
    ...
    public EquipmentBase(int yearOfCreation, int newPrice)
    {
        this.yearOfCreation = yearOfCreation;
        this.newPrice = newPrice;
    }
    ...
}
```

A `HardDisk` és `SoundCard` leszármazottak konstruktorának törzséből vegyük ki a két tag inicializálását, helyette a `base` kulcsszóval hivatkozva hívjuk meg az ős konstruktorát:

```
public class HardDisk : EquipmentBase
{
    ...
    public HardDisk(int yearOfCreation, int newPrice, int capacityGB):
        base(yearOfCreation, newPrice)
    {
        this.capacityGB = capacityGB;
    }
    ...
}
```

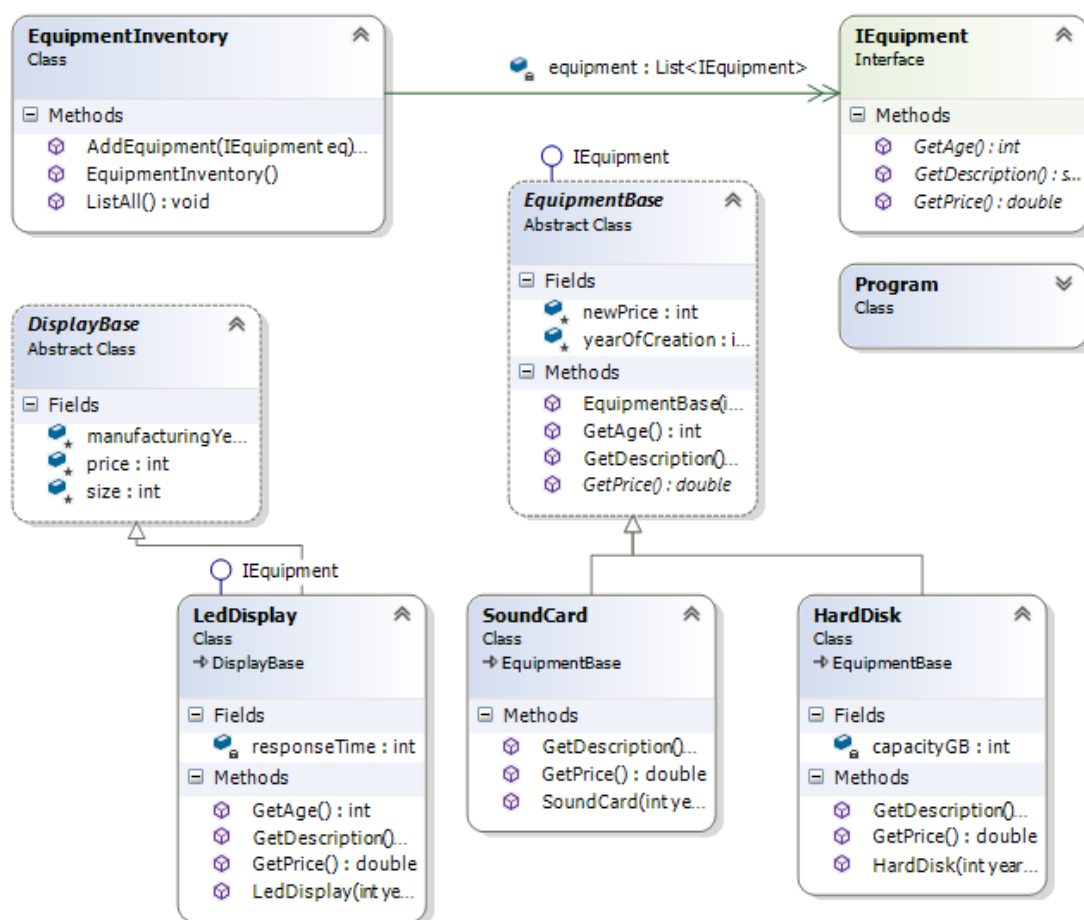
```
public class SoundCard : EquipmentBase
{
    ...
    public SoundCard(int yearOfCreation, int newPrice):
        base(yearOfCreation, newPrice)
    { }
    ...
}
```

Értékelés

Az interfész és absztrakt ős együttes használatával sikerült a legkevesebb kompromisszummal járó megoldást kidolgoznunk:

- `IEquipment` interfészként hivatkozva egységesen tudjuk kezelni az alkatrészek valamennyi típusát, még azokat is, melyeknél az ősosztály még volt kötve (pusztán absztrakt ős használatával ezt nem tudtuk volna elérni).
- Az `EquipmentBase` absztrakt ős bevezetésével egy kivételtől eltekintve a különböző alkatrésztípusokra közös kódot fel tudtuk vinni egy közös ősbe, így el tudtuk kerülni a kódduplikációt.
- Az `EquipmentBase` absztrakt ős bevezetésével alapértelmezett implementációt tudunk megadni az újonnan bevezetett `IEquipment` műveletek esetében (pl. `GetDescription`), így nem vagyunk rákényszerítve, hogy minden `IEquipment` implementációs osztályban meg kelljen azt adni.

Zárásképpen vessünk egy pillantást megoldásunk UML (szerű) osztálydiagramjára:



Megjegyzés (opcionális házi gyakorló feladat)

Jelen megoldásunk nem támogatja az alkatrészspecifikus adatok (pl. **HardDisk** esetében a kapacitás) megjelenítését a listázás során. Ahhoz, hogy ezt meg tudjuk tenni, az alkatrész adatok formázott sztringbe írását az **EquipmentInventory** osztályból az alkatrész osztályokba kellene vinni, a következő elveknek megfelelően:

- Bevezethetünk ehhez az **IEquipment** interfészbe egy `GetFormattedString` műveletet, mely egy `string` típusú objektummal tér vissza. Alternatív megoldás lehet, ha a `System.Object.ToString()` műveletét definiáljuk felül. .NET-ben ugyanis minden típus implicit módon a `System.Object`-ből származik, aminek van egy virtuális `ToString()` művelete.
- Az **EquipmentBase**-ben megírjuk a közös tagok (leírás, ár, kor) sztringbe formázását.
- Amennyiben egy alkatrész típusspecifikus adattal is rendelkezik, akkor osztályában override-oljuk a sztringbe formázó függvényt: ennek a függvénynek egyrészt meg kell hívnia az ős változatát (a `base` kulcsszó használatával), majd ehhez hozzá kell fűzni a saját formázott adatait, és ezzel a sztringgel kell visszatérnie.

Ellenőrző kérdések [hallgató]*

1. Mire használható a Class Diagram Visual Studioban?
 - Kódgenerálás, round-trip engineering, reverse-engineering, kód és UML kapcsolata, stb
2. Mik az interfészek alkalmazásának az előnyei?
3. Mi az interfész alkalmazásának előnye az absztrakt őssosztállyal szemben (Java vagy C# nyelven)?
4. Hogyan tudunk virtuális függvényeket definiálni C#-ban? Mely kulcsszavakat kell alkalmaznunk az ős- illetve a gyerekosztályokban?
5. Hasonlítsa össze az absztrakt őssosztályban lévő virtuális- illetve absztrakt metódusokat!
6. Mi az absztrakt őssosztály használatának előnye az interfészekkel szemben (Java vagy C# nyelven)?
7. Mi a következménye annak, ha egy interfészben új műveletet vezetünk be (az implementáló osztályok szempontjából)?
8. Mit nyerünk azzal, ha interfészeket és absztrakt őssosztályt együtt alkalmazzuk?