Eötvös Loránd Tudományegyetem
Informatikai Kar

Objektumelvű alkalmazások fejlesztése

6. gyakorlat

Öröklődés, polimorfizmus

© 2011.10.27. Giachetta Roberto
groberto@inf.elte.hu
http://people.inf.elte.hu/groberto

Öröklődés
Kódismétlődés objektum-orientált szerkezetben

- Az objektum-orientált programokban az osztályok definiálják az objektumok működési sémáját
  - sok esetben hasonló, de mégis eltérő viselkedésre van szükségünk különböző objektumoktól, ekkor azoknak külön osztályt kell definiálnunk akkor is, ha sok ismétlődés előfordul
  - emiatt *kódismétlődéssel* kell szembenéznünk, amelyet a procedurális programokban kódrészlet kiemeléssel, alprogramokkal oldottunk fel, itt is valamilyen hasonló eszközhöz kell folyamodnunk
  - pl.: az egyetemi oktató és az egyetemi hallgató is tartalmaz nevet, azonosítót, kurzuslistát, és vannak ugyanolyan műveleteik

Öröklődés
Osztályok közötti hasonlóságok

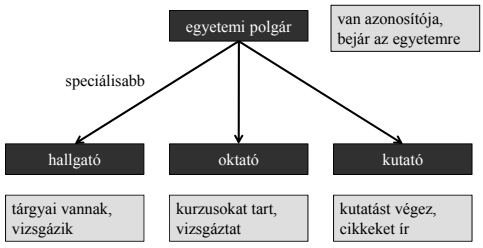
- Az osztályok között felfedezhetők az ismétlődések, amelyeknek két esete van:
  - két, vagy több osztály közös tagokkal rendelkezik, ekkor célszerű lenne a közös részt kiemelni
    - pl. a háromszög és a négyszög is rendelkezik pontokkal, kerülettel és területtel
  - egy osztály rendelkezik egy másik osztály valamennyi tagjával, és ezen felül továbbiakkal, ekkor mondhatjuk azt, hogy az osztály *speciális esete* a másiknak (illetve a másik *általános esete* ezen osztálynak)
    - pl. a háromszög és a négyszög speciális esete a sokszögnek, és minden sokszög rendelkezik kerülettel és területtel

Öröklődés
Általánosítás és specializáció

- Az első eset kombinálható a másodikkal, hiszen ha több osztálynak közös tulajdonságai vannak, akkor van olyan általánosabb osztály, amely azokat a tulajdonságokat tartalmazza
  - pl. az egyetemi hallgató és oktató közös tulajdonságait az egyetemi polgár osztály tartalmazza, így azok ennek speciálisabb változatai lesznek
- Egy osztálynak tehát lehet egy, vagy több speciálisabb változata, amely mindent tud, amit az általánosabb, és ezen felül még kiegészítheti tagjait tetszőleges számban
  - fordítva, egy osztálynak lehet egy (esetekben akár több) általánosabb változata, amelytől átveszi a viselkedést

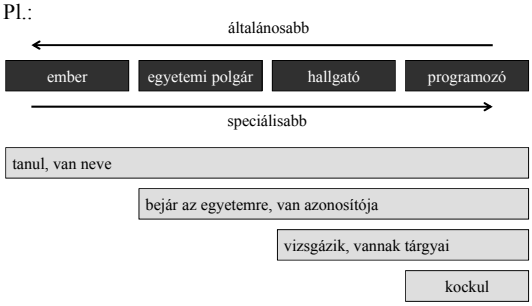
Öröklődés
Általánosítás és specializáció

- Pl.:



Öröklődés
Általánosítás és specializáció

- Pl.:



Objektumorientált programozás
Az öröklődés

- Ezt a folyamatot, amikor a speciálisabb átveszi az általános jellemzőit és működését, *öröklődés*nek nevezzük
- Az öröklődés két irányát *specializáció*nak, illetve *általánosítás*nak nevezzük, az általánosabb objektumot *ős*nek, a speciálisabbat *leszármazott*nak nevezzük (ha csak egy szint a különbség, akkor szülőnek, illetve gyereknek)
- Általában egy szülőnek több gyereke is van, ám csak egyes esetekben engedélyezett, hogy egy gyereknek több szülője is legyen, ezt *többszörös öröklődés*nek nevezzük
  - ekkor a gyerek megkaphatja minden szülője minden tulajdonságát, vagy szabályozhatjuk, hogy mely szülőtől melyeket kapja

Öröklődés
Működése

- Az öröklődést az osztályoknál kell jelölünk úgy, hogy megadjuk, mely ősosztály(ok)nak örökli a tulajdonságait
  - az osztály tagjainak láthatósága szabályozza az öröklődés módját, de maga az öröklődés is rendelkezik láthatósággal
  - egyszeres öröklődés:

```
class <osztálynév> : <láthatóság> <ős> {
  <osztályfelület>
};
```

  - többszörös öröklődés:

```
class <osztálynév> : <láthatóság> <ős 1>,
                     <láthatóság> <ős 2>, ... {
  <osztályfelület>
};
```


Öröklődés
Megvalósítása

- Pl.:

```
class DemoBaseClass { // általános osztály
public:
   int baseValue;
   void BaseMethod(int val) { baseValue = val; }
   DemoBaseClass() { baseValue = 1; }
};
class DemoChildClass : public DemoBaseClass {
// speciális osztály, a fenti leszármazottja
public:
   // automatikusan megkapja a baseValue,
   // BaseMethod tulajdonságokat
   int childValue;
   DemoChildClass() { childValue = 2; }
```


Öröklődés
Megvalósítása

```
   void ChildMethod() {
      BaseMethod(childValue);
      // örökölt metódus meghívása
   }
};

DemoBaseClass dbc;
cout << dbc.baseValue; // kiírja: 1
DemoChildClass dcc;
cout << dbc.childValue; // kiírja: 2
dcc.ChildMethod();
cout << dcc.baseValue; // kiírja: 2
dcc.BaseMethod(5);
dcc << dcc.baseValue; // kiírja: 5
```


Öröklődés
A láthatóság szerepe

- Az osztálytulajdonságokra 3 láthatóságot alkalmazhatunk:
  - `public`: látható, öröklődik
  - `private`: rejtett, öröklődik, azonban a leszármazott osztályokban nem lesz látható
  - `protected`: rejtett, öröklődik, az összes leszármazott osztályban látható lesz
- Magára az öröklődésre is három láthatóságot alkalmazhatunk:
  - `public`: minden a megadott láthatósággal öröklődik tovább
  - `protected`: a látható, valamint a rejtett tagok is rejtettként kerülnek a gyerekbe, de továbbra is örökölhetőek lesznek
  - `private`: a látható, valamint a rejtett tagok is rejtettként kerülnek a gyerekbe

Öröklődés
Példa

- Pl.:

```
class DemoBaseClass { // ős osztály
private: // rejtett, öröklődik, de nem látható
   int privateValue;
   void PrivateMethod();
protected: // rejtett, öröklődik, és látható
   int protectedValue;
   void ProtectedMethod();
public: // látható és öröklődik
   int publicValue;
   void PublicMethod();
   DemoBaseClass();
   ~DemoBaseClass();
};
```

Öröklődés
Példa

```
class DemoPublicChild : public DemoBaseClass {
  public:
    void ChildPublicMethod(){
    // itt elérhető:
    // protectedValue, ProtectedMethod,
    // publicValue, PublicMethod
    }
}

…
DemoPublicChild x;
// elérhető: x.ChildPublicMethod, x.publicValue,
// x.PublicMethod
```


Öröklődés
Példa

```
class DemoProtectedChild : protected DemoBaseClass
{
public: void PublicMethod(){
   // elérhető: ProtectedValue, ProtectedMethod,
   //           PublicValue, PublicMethod
    }
}
// az örökölt tagok nem érhetőek el az osztályon
// kívül

class DemoPrivateChild : private DemoBaseClass {
public: void PublicMethod(){ /*...*/ }
//... a helyzet ugyanaz
};
```


Öröklődés
Példa

```
class DemoChildOfProtected
   : public DemoProtectedChild {
public: void SomeMethod(){
   // elérhető: protectedValue, publicValue,
   //           ProtectedMethod, PublicMethod,
   //           ChildPublicMethod
    }
};
class DemoChildOfPrivate : public DemoPrivateChild
{
public: void SomeMethod(){
    // itt csak a ChildPublicMethod érhető el
    }
};
```


Öröklődés
A láthatóság szerepe

- Egy **`private`** tulajdonságot is el tudunk érni a leszármazottban egy nem **`private`** örökölt metóduson keresztül, pl.:

```
class DemoBaseClass{
private: // leszármazottban nem látható
   int baseValue;
public: // leszármazottban látható
   int GetBaseValue() { return baseValue; }
};

class DemoChildClass : public DemoBaseClass {
public:
   int GetValue() { return GetBaseValue(); }
   // így elérjük az értéket
};
```

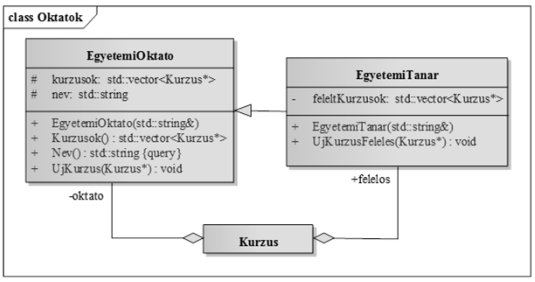

Öröklődés
Példa

Feladat: Készítsük el az egyetemi oktató, valamint az egyetemi tanár osztályát. Az egyetemi oktatónak van neve, illetve vannak oktatott kurzusai, ahova tud újakat felvenni. Az egyetemi tanár nagyobb jogkörrel bír, ő felelős is lehet kurzusokért, ezért külön kezeljük a felelt, illetve oktatott kurzusait. A kurzusoknál is nyilván kell tartani az oktatót, valamint a felelős tanárt.

- hozzuk létre az egyetemi oktatót, és annak egy leszármazottja lesz az egyetemi tanár, aki a felelős kurzusokat is tartalmazza
- használjuk a korábbi kurzus osztályt, amit kiegészítünk az oktatóval, illetve a felelős tanárral, amiket kötelező megadni a kustruktoron keresztül

Öröklődés
Példa

Tervezés:



Öröklődés
Példa

Megoldás:

```
class EgyetemiOktato{ // oktató osztály
public:
   EgyetemiOktato(const std::string& n) : nev(n)
   {}
   std::string Nev() const { return nev; }
   const std::vector<Kurzus*> Kurzusok() const {
      return kurzusok;
   }
   void UjKurzus(Kurzus* k);
protected:
   std::string nev;
   std::vector<Kurzus*> kurzusok;
};
```


Öröklődés
Példa

Megoldás:

```
class EgyetemiTanar : public EgyetemiOktato {
// egyetemi tanár, az oktató leszármazottja
public:
   // minden tulajdonságot megkap az oktatótól
   EgyetemiTanar(const std::string& n)
   : EgyetemiOktato (n){}
   // meghívjuk az ős konstruktorát
   void UjKurzusFeleles(Kurzus* k);
private:
    std::vector<Kurzus*> feleltKurzusok;
};
```


Öröklődés
Példa

Megoldás:

```
class Kurzus{ // kurzus osztálya
public:
   Kurzus(const std::string& nev, EgyetemiTanar*
          fel, EgyetemiOktato* okt …
private:
   std::string kurzusNev;
   EgyetemiTanar* felelos;
   EgyetemiOktato* oktato;
   …
}
```


Öröklődés
Tulajdonságok elrejtése és elérése

- Az őssel megegyező nevű attribútumok, illetve megegyező szintaktikájú metódusok *elrejtik* az örökölt tulajdonságokat, azaz híváskor a leszármazott osztály megfelelő tulajdonságát érjük el
- Lehetőségünk van explicit hivatkozni az ős bármely látható tulajdonságára az ***<ős osztály>*****::** előtaggal (csak az azonos nevű attribútumok, illetve azonos szintaktikájú metódusoknál szükséges)
- Pl.:

```
class DemoBaseClass {
public:
   void SomeMethod() { cout << 1 << endl; }
};
```


Öröklődés
Tulajdonságok elrejtése és elérése

```
class DemoChildClass : public DemoBaseClass {
public:
   void SomeMethod() { cout << 2 << endl; }
   // elrejtő metódus
   void RunBaseMethod() {
      DemoBaseClass::SomeMethod();
   } // ős metódusának meghívása
};

…
DemoBaseClass dbc; dbc.SomeMethod(); // kiírja: 1
DemoChildClass dcc;
dcc.SomeMethod(); // kiírja: 2
dcc.RunBaseMethod(); // kiírja: 1
```


Öröklődés
A konstruktor és destruktor öröklődése

- A konstruktor automatikusan öröklődik
  - a paraméteres nélküli konstruktor automatikusan meghívódik amikor a leszármazottból létrehozunk egy példányt
    - elsőként az ős konstruktora hívódik meg, aztán a leszármazottaké lefelé haladó sorrendben az öröklődési fán
  - lehetőségünk van az ős konstruktorának explicit meghívására is ***<osztálynév> <konstruktor> : <ős konstruktornév>(<átadott paraméterek>)*** formában
  - paraméteres konstruktorokra csak az explicit hívás használható

Öröklődés
Konstruktor és destruktor

- A destruktor automatikusan öröklődik és minden ős destruktor megívódik a leszármazott destruktor meghívásakor
  - elsőként a leszármazotté, majd a sorrendben az ősök destruktorai felfelé az öröklődési fán
- Pl.:

```
class DemoBaseClass {
public:
   DemoBaseClass() { cout << "1 start" << endl; }
   ~DemoBaseClass() { cout << "1 stop" << endl; }
};

class DemoChildClass : public DemoBaseClass {
public:
```


Öröklődés
Konstruktor és destruktor

```
   DemoChildClass() { cout << "2 start" << endl; }
   ~DemoChildClass() { cout << "2 end" << endl; }
};

int main(){
   DemoChildClass dcc; // konstruktor hívás
   return 0;
} // destruktor hívás

/* eredmény:
1 start
2 start
2 stop
1 stop */
```


Öröklődés
Viselkedés felüldefiniálás

- A leszármazott amellett, hogy örököl minden tulajdonságot az őstől, lehetősége van *felüldefiniálni* az örökölt viselkedést
  - azaz az örökölt metódusok definícióját átírhatja, de a deklarációját nem (azaz a törzse átírható, a szintaxisa nem, különben nem felüldefiniálásnak minősül, hanem új metódus létrehozásának)
  - a felüldefiniálható metódusokat nevezzük *virtuális metódusok*nak
    - a konstruktor sohasem lehet virtuális
  - az ősben előre kell jelezni, ha megengedjük a működés felüldefiniálhatóságát a `virtual` kulcsszóval
    - a kulcsszó csak egy felüldefiniálást tesz lehetővé

Öröklődés
Viselkedés felüldefiniálás

- Pl.:

```
class DemoBaseClass {
public:
   virtual void SomeMethod(){ cout << 1 << endl; }
   // felüldefiniálható (virtuális) metódus
};
class DemoChildClass : public DemoBaseClass {
public:
   void SomeMethod(){ cout << 2 << endl; }
   // felüldefiniáló metódus
   void RunBaseMethod() {
      DemoBaseClass::SomeMethod();
   } // ős metódusának meghívása
};
```


Öröklődés
Viselkedés felüldefiniálás

```
…
DemoBaseClass dbc; dbc.SomeMethod(); // kiírja: 1
DemoChildClass dcc;
dcc.SomeMethod(); // kiírja: 2
dcc.RunBaseMethod(); // kiírja: 1
```

- Azon metódusokat, amelyek nem definiálhatóak felül nevezzük *véglegesített*nek, tehát alapértelmezetten minden metódus véglegesített
  - a konstruktor csak véglegesített lehet, viszont automatikusan meghívódik
- Önmagában a felüldefiniálás és az elrejtés között nincs különbség, de a későbbiekben lesz

Öröklődés
Példa

Feladat: Amennyiben az egyetemi tanár oktat egy kurzust, akkor annak egyben felelőse is lesz.

- ehhez a kurzus hozzáadását felül kell definiálnunk az egyetemi tanárnál, így az oktatóban virtuálisként kell megadnunk a metódust

Megoldás:

```
class EgyetemiOktato{
   …
   virtual void UjKurzus(Kurzus* k);
   …
}
```

Öröklődés
Példa

```
class EgyetemiTanar : public EgyetemiOktato{
   …
   void UjKurzus(Kurzus* k);
   …
}
```

- Felmerülő problémák:
  - az oktató és a tanár két külön osztály, ezért külön kell őket kezelni, és külön kell őket adatszerkezetbe szervezni
  - továbbra is két külön paraméterben kell megadni a tárgy felelősét és oktatóját, de mi van, ha a kettő megegyezik
  - tehát kezelhető lenne-e a tanár oktatóként (azaz ősosztálya példányaként), amennyiben a környezet ezt indokolttá teszi

Polimorfizmus
Dinamikus kötés

- Ha egy objektum olyan osztálynak példánya, amelynek több őse is van, akkor az objektumorientált paradigma szerint az objektum tekinthető bármely ősének példányaként is, ezt nevezzük *polimorfizmus*nak (többalakúságnak)
  - ezáltal lehetővé válik, hogy egy objektumnak több osztálya is legyen, és aktuálisan azt az osztályt használjuk, amely az aktuális környezetbe beleillik
- A polimorfizmust a programozási nyelvek többnyire mutatókon keresztül kezelik, amikor a mutató típusa más, mint az általa hivatkozott objektumé, *dinamikus kötés*nek nevezzük
  - pl.:
```
DemoBaseClass* dbc = new DemoChildClass();
   // a mutató típusa DemoBaseClass, de
   // egy DemoChildClass objektumra mutat
```


Polimorfizmus
Statikus és dinamikus típus

- Dinamikus kötés esetén
  - az ősosztály a változó *statikus típusa*, amit a fordítóprogram is értelmezni tud, tehát a mutatón keresztül csak azok a tulajdonságai érthetőek el, amelyekkel már az ős is rendelkezik
  - a leszármazott a változó *dinamikus típusa*, ugyanis futás közben bár csak az ős műveletei érhetőek el, a változó a leszármazott viselkedését fogja produkálni, hiszen így lett példányosítva
- Mivel egy objektumhoz több mutató is hozzárendelhető, ezért egy objektum egyszerre több statikus típussal is rendelkezhet, dinamikus típusa mindenképpen csak egy lehet, ugyanakkor egy mutatóhoz futás közben több dinamikus típus is tartozhat

Polimorfizmus
Statikus és dinamikus típus

- A dinamikus típus futás közben szabályozható, ezért egy változó konkrét típusa tetszőlegesen variálható a leszármazottak és a statikus típus között futás közben
- Pl.:

```
DemoBaseClass* dbc = new DemoBaseClass();
dbc->SomeMethod(); // kiírja: 1
dbc = new DemoChildClass();
dbc->SomeMethod(); // kiírja: 2
dbc->RunBaseMethod();
// fordítási hiba, a statikus típusnak nincs
// RunBaseMethod művelete
DemoChildClass* dcc = new DemoChildClass();
dcc->SomeMethod(); // kiírja: 2
```


Polimorfizmus
Virtuális függvények szerepe

- Polimorfizmus esetén, amennyiben a metódus
  - virtuális, életbe lép a viselkedés felüldefiniálás, és a dinamikus típus szerint kerül lefutásra, pl.:

```
… virtual void SomeMethod() …
DemoBaseClass* dbc = new DemoChildClass();
dbc->SomeMethod(); // kiírja: 2
```

  - véglegesített, akkor csak elrejtés történik, amit a program nem vesz figyelembe, így a statikus típus szerint kerül lefutásra, pl.:

```
… void SomeMethod() …
DemoBaseClass* dbc = new DemoChildClass();
dbc->SomeMethod(); // kiírja : 1
```


Polimorfizmus
Adatszerkezetbe szervezés

- A leszármazottak példányai elérhetőek egyazon típusú mutatóval, ezért egy gyűjteménybe (például tömbbe) szervezhetőek a különböző leszármazott típusú változók, és egységesen tudjuk kezelni őket
- Pl.:

```
DemoBaseClass* dbcs[3];
dbcs[0] = new DemoBaseClass();
dbcs[1] = new DemoChildClass();
dbcs[2] = new DemoChildClass();

for (int i = 0; i < 3; i++)
   dbcs[i]->SomeMethod();
// kiírja: 1 2 2
```

Polimorfizmus
Virtuális destruktor

- Polimorfizmus esetén a program az objektum megsemmisítésénél a statikus típust veszi figyelembe, ezért annak destruktorát hívja meg
- Pl.:

```
… ~DemoBaseClass() …

int main(){
   DemoBaseClass* dbc = new DemoChildClass();
   delete dbc;
   return 0;
}
/* eredménye:  1 start    2 start    1 stop */
```


Polimorfizmus
Virtuális destruktor

- Ha a dinamikus típus szerint akarjuk elvégezni a törlést, akkor a statikus típusban a destruktort virtuálisnak kell megadni
- Pl.:

```
… virtual ~DemoBaseClass() …

int main(){
   DemoBaseClass* dbc = new DemoChildClass();
   delete dbc;
   return 0;
}
/* eredménye:
   1 start    2 start    2 stop    1 stop */
```


Polimorfizmus
Típuskonverzió

- Amennyiben szeretnénk a dinamikus típus műveleteit használni típuskonverzióra van szükségünk, amely során rákényszerítjük a fordítóra a dinamikus típus értelmezését
- Típuskonverziót mutatókra biztonságos módon `dynamic_cast<típus>(<mutató>)` utasítással végezhetünk
  - a típuskonverzió feltétele egy virtuális függvény megléte
  - helytelen konverzió esetén `NULL` mutatót ad vissza
- Pl.:

```
DemoBaseClass* dbc = new DemoChildClass();
if (dynamic_cast<DemoChildClass*>(dbc))
   // ha konvertálható az adott típusra
   dynamic_cast<DemoChildClass*>(dbc)
      ->RunBaseMethod(); // így már futtatható
```


Absztrakt osztályok
Absztrakt metódusok

- Lehetőségünk van viselkedéseket kötelezően felüldefiniálhatóvá tenni úgy, hogy nem adjuk meg a metódus törzsét az ősben, csak a leszármazottban
  - ezek az *absztrakt*, vagy *tisztán virtuális* metódusok, ekkor a törzs helyébe `=0;`-t kell írnunk
  - ekkor az leszármazott osztálynak nem csak felül lehet definiálnia, de felül kell definiálnia a metódust
- Pl.:

```
class DemoBaseClass {
public:
   virtual void SomeMethod() = 0;
   // felüldefiniálandó (absztrakt) metódus
};
```


Absztrakt osztályok
Absztrakt osztályok

- Amennyiben egy osztályban van absztrakt metódus, akkor az osztály nem példányosítható – hiszen lenne olyan metódusa, amihez nincs működés társítva –, ekkor az osztályt *absztrakt osztály*nak nevezzük
  - arra jók, hogy az öröklődési hierarchiában összefogják több osztály közös részét, és a polimorfizmus segítségével hivatkozni lehessen ezen közös részekre
- Pl.:

```
DemoBaseClass* dbc = new DemoBaseClass();
// hiba, absztrakt osztály nem példányosítható
DemoBaseClass* dbc = new DemoChildClass();
// a DemoChildClass már példányosítható, mert ott
// meg van adva a SomeMethod törzse
```


Absztrakt osztályok
Interfészek

- Definiálhatunk speciális absztrakt osztályokat, amelyek nem tartalmaznak megvalósítást, csak felületet, ezek az *interfész*ek
  - interfészben csak publikus, absztrakt műveletek szerepelhetnek, azaz nem tartalmazhatnak attribútumokat, vagy rejtett metódusokat
  - leginkább többszörös öröklődés kapcsán használatosak, további ősökként megadva, kivédve ezzel a többszörös öröklődés miatti problémákat

Absztrakt osztályok
Interfészek

- Pl.:

```
class DemoInterface { // interfész
public:
   virtual void SomeMethod() = 0;
}; // csak absztrakt függvényeket tartalmazhat

class DemoBaseClass : public DemoInterface {
public:
   void SomeMethod() { cout << 1 << endl; }
}; // megadjuk a művelet törzsét
```


Absztrakt osztályok
Interfészek

```
class DemoChildClass : public DemoBaseClass {
public:
   void SomeMethod() { cout << 2 << endl; }
   void RunBaseMethod() {
      DemoBaseClass::SomeMethod();
      // meghívjuk az ős metódusát
   }
};

…
DemoInterface* di = new DemoBaseClass();
di->SomeMethod(); // kiírja: 1
di = new DemoChildClass();
di->SomeMethod(); // kiírja: 2
```

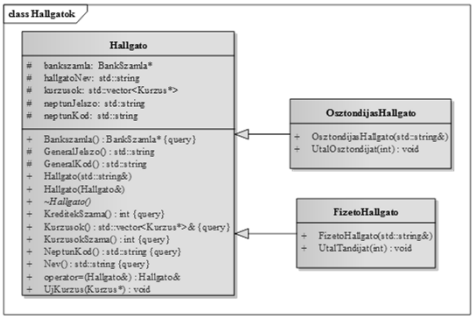

Öröklődés
Példa

Feladat: A hallgatók tekintetében különböztessük meg az ösztöndíjas, illetve a fizető hallgatókat. Előbbiek ösztöndíjat kapnak, utóbbiak tandíjat fizetnek.

- a hallgató így absztrakt osztály lesz a közös tulajdonságoknak, és ennek két specializációját hozzuk létre, a hallgató **`private`** tulajdonságait **`protected`**-dé kell minősíteni
- az ösztöndíjfizetés változatlan marad, a tandíj fizetéshez használjuk a bankszámla **`Kivet`** műveletét
- a bankszámla-, illetve a kurzuskezelés továbbra is egységes lesz a hallgatókra, ezért itt kihasználjuk a polimorfizmust
- a hallgatókat a főprogramban tárolhatjuk egy vektorban szintén polimorfizmust használva

Öröklődés
Példa

Tervezés:



Öröklődés
Példa

Megoldás:

```
class OsztondijasHallgato : public Hallgato{
// a hallgató osztály leszármazottja
public:
    OsztondijasHallgato(const std::string& n)
    : Hallgato(n) {} // ős konstruktor meghívása
    void UtalOsztondijat(const int osszeg);
};
class FizetoHallgato : public Hallgato{
public:
    FizetoHallgato(const std::string& n)
    : Hallgato(n) {}
    void UtalTandijat(const int osszeg);
};
```


Öröklődés
Példa

Megoldás:

```
vector<Hallgato*> hallgatok;
// mutatókat tartalmazó vektor
hallgatok.push_back(
   new OsztondijasHallgato("Gem Géza"));
   // polimorfizmust használva a tényleges típus a
   // leszármazotté lesz
dynamic_cast<OsztondijasHallgato*>(hallgatok[0])
   ->UtalOsztondijat(20000);
   // típuskonverzióval elérhetjük a leszármazott
   // műveletét
hallgatok.push_back(new FizetoHallgato("Go Hugo"));
dynamic_cast<FizetoHallgato*>(hallgatok[1])
   ->UtalTandijat(50000);
```

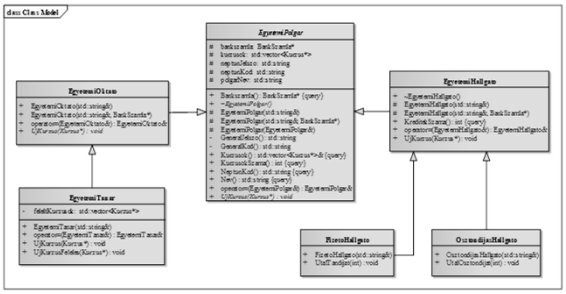
Öröklődés
Példa

Feladat: Integráljuk az oktató osztályokat az eddigi szerkezetbe. Az oktató és a hallgató is speciális esete az egyetemi polgárnak, ezért e mentén általánosíthatóak.

- bevezetjük az egyetemi polgár absztrakt osztályt, amely átveszi a hallgató tulajdonságainak nagy részét, a hallgató továbbra is absztrakt lesz
- a bankszámla és a kurzus mostantól az egyetemi polgárral lesz kapcsolatban, itt kihasználjuk a polimorfizmust
- az oktatóknak fizetést utalunk, az ösztöndíjas hallgatóknak ösztöndíjat, míg a fizető hallgatók tandíjat fizetnek

Öröklődés
Példa

Tervezés:



Öröklődés
Példa

Megoldás:

```
class EgyetemiPolgar{
public:
   virtual ~EgyetemiPolgar();
   // virtuális destruktor
   …
   virtual void UjKurzus(Kurzus* k) = 0;
   // absztrakt metódus
   EgyetemiPolgar& operator=
      (const EgyetemiPolgar& masik);
protected:
   EgyetemiPolgar(const std::string& n);
   // a konstruktort elrejtjük
   …
```


Öröklődés
Példa

Megoldás:

```
class EgyetemiOktato : EgyetemiPolgar{
   // most már származtatott osztály
public:
   EgyetemiOktato(const std::string& n)
   : EgyetemiPolgar(n) {} // ős konstruktor hívás
   EgyetemiOktato(const std::string& n,
                  BankSzamla* szamla)
   : EgyetemiPolgar(n, szamla) {}
   EgyetemiOktato& operator=
      (const EgyetemiOktato& masik);
   // új egyenlőség operátor
   void UjKurzus(Kurzus* k); // felüldefiniálás
};
```


Öröklődés
Példa

Megoldás:

```
vector<EgyetemiOktato* > oktatok;
EgyetemiTanar* et =
   new EgyetemiTanar("Gregorics Tibor");
oktatok.push_back(et);
oktatok.push_back(
   new EgyetemiOktato("Giachetta Roberto"));

vector<Kurzus*> kurzusok;
kurzusok.push_back(new Kurzus("OAF", et,
   oktatok[1], 4));
kurzusok.push_back(new Kurzus("EVA1", et,
   NULL, 4)); // nem adunk meg külön oktatót
```