10. Óra

Miről lesz szó

Interface-ek

Az interfész egy *"teljesen absztrakt osztály"*, amely csak absztrakt metódusokat és property-ket tartalmazhat

- Benne lévő metódusok és property-k impliciten **absztraktak** és **publikusok**
- Konvenció, hogy nagy `I`-vel kezdődnek a neveik
- "Deklarációk"

```
interface IAnimal {
    int Height { get; set; } // Read-write Property
    int LegCount { get; } // Read only Property

    void MakeSound(int times); // Abstract Method paraméterekkel
}

interface IPet {
    string Name { get; set; }

    void GoForAWalk();
}

class Dog : IAnimal, IPet { // Implementálhat több interface-t is az osztály!
    private int _height;
    private int _legCount;
    private string _name;

    // Property implementáció
    public int Height {
        get => _height;
        set => _height = value;
    }

    // Property implementáció
    public int LegCount => _legCount;

    // Property implementáció
    public string Name {
        get => _name;
        set => _name = value;
    }

    // Metódus implementáció
    public void MakeSound(int times) {
        for (int i = 0; i < times; i++) {
            Console.WriteLine("Vau");
        }
    }

    // Metódus implementáció
    public void GoForAWalk() {
        // ...
    }
}

// Vaaagy a property-k másik implementálási módja

// Ezek itt definíciók, nem deklarációk, nem összekeverendő, a szintaxis nagyon hasonló!
```

```
class Dog : IAnimal, IPet {
    // Property implementáció
    public int Height { get; set; }

    // Property implementáció
    public int LegCount { get; private set; }

    // Property implementáció
    public string Name { get; set; }

    // ...
}
```

Interface-k implementálása mellett lehet (természetesen max 1) *base class*-a az osztályunknak:

```
//   Base class ┐  Interfacek ┐
//              ┌─┐ ┌─────┐ ┌─┐
class Dog : Mammal, IAnimal, IPet {
    // ...
}
```

Kérdések

1

Mi történik, ha létezik **ugyanolyan nevű** és **szignatúrájú** *metódus* két külön interface-ben amit egy osztály implementál?

```
interface IA {
    bool Foo(int n, string str);
}

interface IB {
    bool Foo(int n, string str);
}

public class Test : IA, IB {
    // ???
}
```

Válasz

Semmi `¯\_(ツ)_/¯`
Értelemszerűen nem kell kétszer definiálni

```
public class Test : IA, IB {
    public bool Foo(int n, string str) {
        return n > 10;
    }
}
```

2

Mi történik, ha létezik **ugyanolyan nevű**, de **különböző szignatúrájú** *metódus* két külön interface-ben amit egy osztály implementál?

```
interface IA {
    bool Foo(int n, string str);
}
```

```
interface IB {
    void Foo(bool f);
}

public class Test : IA, IB {
    // ???
}
```

Válasz

Implementálásnál overloadolunk

```
public class Test : IA, IB {
    public bool Foo(int n, string str) {
        return n > 10;
    }

    public void Foo(bool f) {
        return;
    }
}
```

3

Mi történik, ha létezik **ugyanolyan nevű** és **típusú** *property* két interface-ben amit egy osztály implementál?

```
interface IA {
    bool SomeProperty { get; set; }
}

interface IB {
    bool SomeProperty { get; set; }
}

public class Test : IA, IB {
    // ???
}
```

Válasz

Ugyanúgy elég egyszer definiálni:

```
public class Test : IA, IB {
    public bool SomeProperty { get; set; }
}
```

4

Mi történik, ha létezik **ugyanolyan nevű**, de **különböző típusú** *property* két interface-ben amit egy osztály implementál?

```
interface IA {
    int Value { get; }
}

interface IB {
    string Value { get; }
}

class Confusion : IA, IB {
```

```
    // ???
}
```

Válasz

```
class Confusion : IA, IB {
    int IA.Value => 0;
    string IB.Value => "Hello There!";
}

...

Confusion confusion = new Confusion();
int intValue = ((IA)confusion).Value; // Castolni kell az interfacere, ew
string stringValue = ((IB)confusion).Value;
```

Ez igaz akkor is ha a metódusnak a visszatérési értéke különbözik
Nyilván csúnya, kerüljük ha tudjuk

Property vagy Method? (*Paraméter nélküli függvényeknél*)

Mikor érdemes propertybe szervezni a kódot és mikor metódusba?

- Nincs aranyszabály, de legyünk konzisztensek.
- Nyilván első az osztálydiagram, ha meg van adva valahogy akkor csináljuk úgy.
- De megfontolandó:

Property

- Adat, vagy valamiféle query
- Nincsenek mellékhatások
- Olcsó operációk
- Bármilyen sorrendben hívhatók kell legyenek

Method

- Mellékhatásos dolgok
- Drágább operációk (használóval kommunikáljuk, hogy itt valami költségesebb dolog is lefuthat)

```
public class Kolbi {
    private double _weight;
    private double _length;

    public double Value() {
        return _weight * _length;
    }
}
```

↓ ↓

```
public class Kolbi {
    private double _weight;
    private double _length;

    public double Value {
        get { return _weight * _length; } // olcsó művelet
    }
}
```

Megjegyzés: mindkettő megoldás rövidíthető az alábbi módon:

```
public class Kolbi {
    private double _weight;
    private double _length;

    public double Value() => _weight * _length; // Metódus
}
```

```
public class Kolbi {
    private double _weight;
    private double _length;

    public double Value => _weight * _length; // Property
}
```

Feladatok

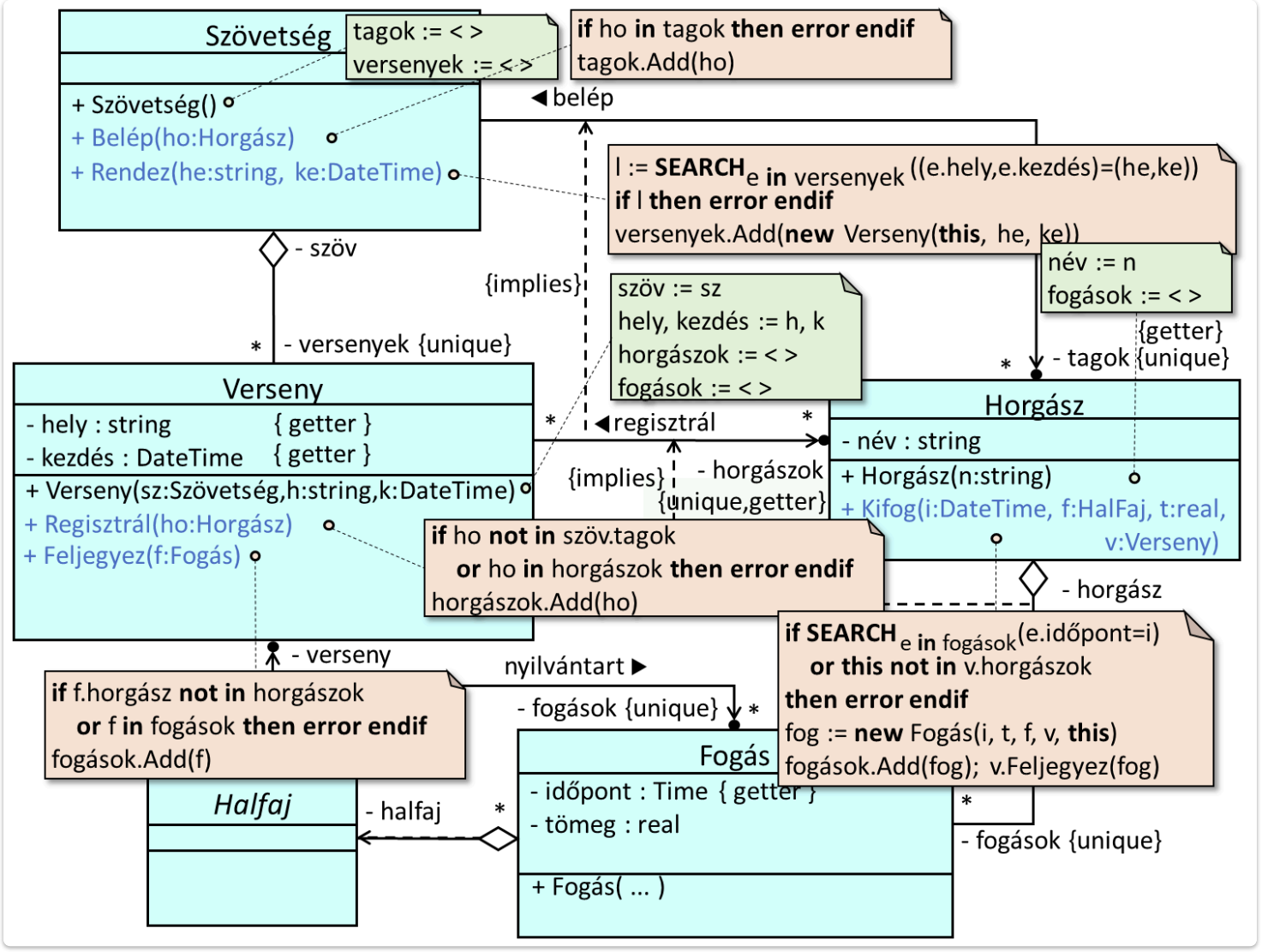
Horgászverseny

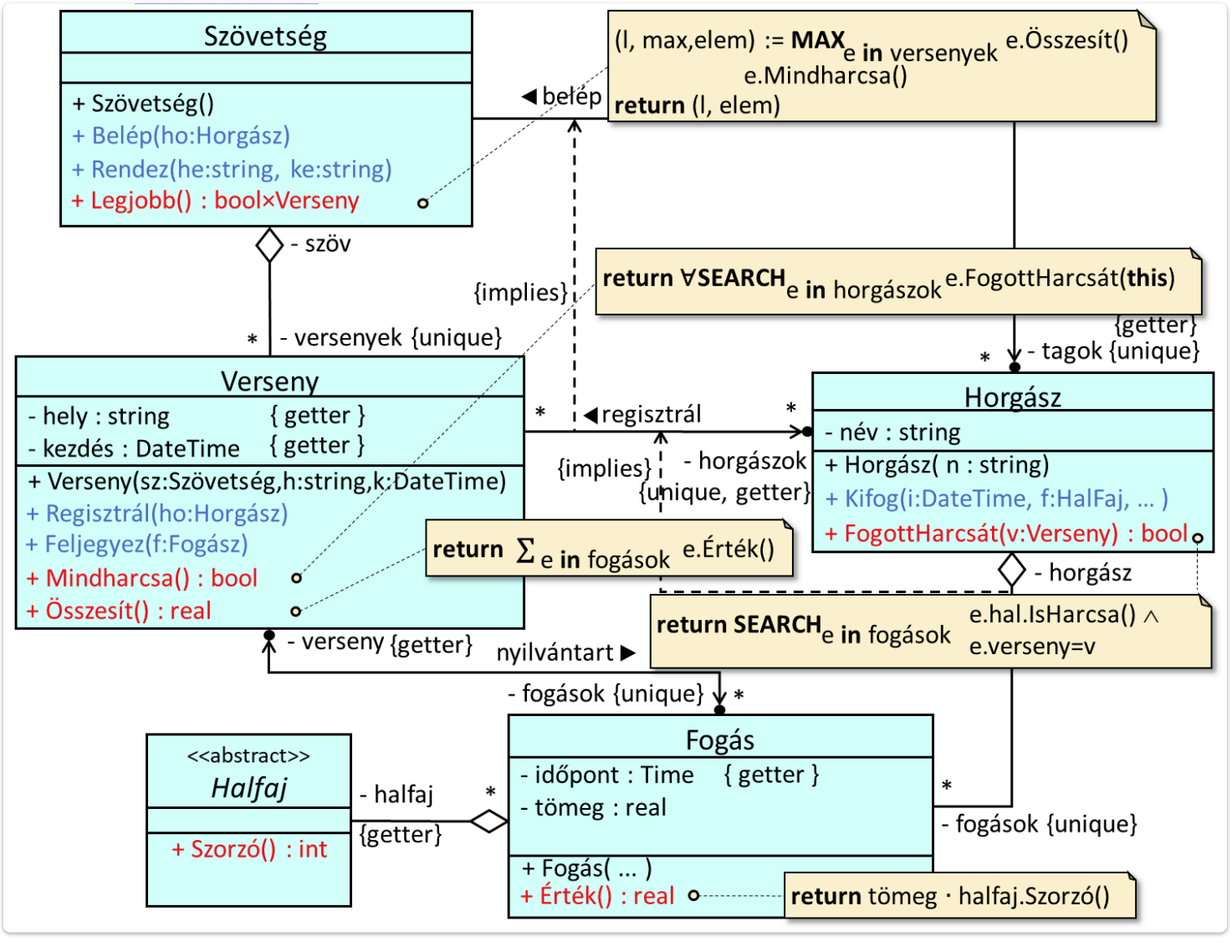
Egy horgászszövetség több horgászversenyt is rendez, amelyekre csak a szövetség tagjai nevezhetnek be; ugyanaz a horgász több versenyen is részt vehet. A versenyeknek ismert a helyszíne. A horgászoknak ismerjük a nevét, tudjuk, hogy milyen fogásaik voltak az egyes versenyeken. Egy fogás leírja, hogy melyik versenyen fogták, ki volt a horgász, mi a kifogott hal fajtája és a tömege (kg-ban). A halak fajtája lehet ponty, keszeg, vagy harcsa. A hal értéke a hal tömegének és a halfajta szorzójának (harcsa:3, ponty:2, keszeg:1) szorzata.
Melyik a legeredményesebb verseny: ahol a horgászok fogásainak összértéke a legnagyobb és mindenki fogott harcsát?

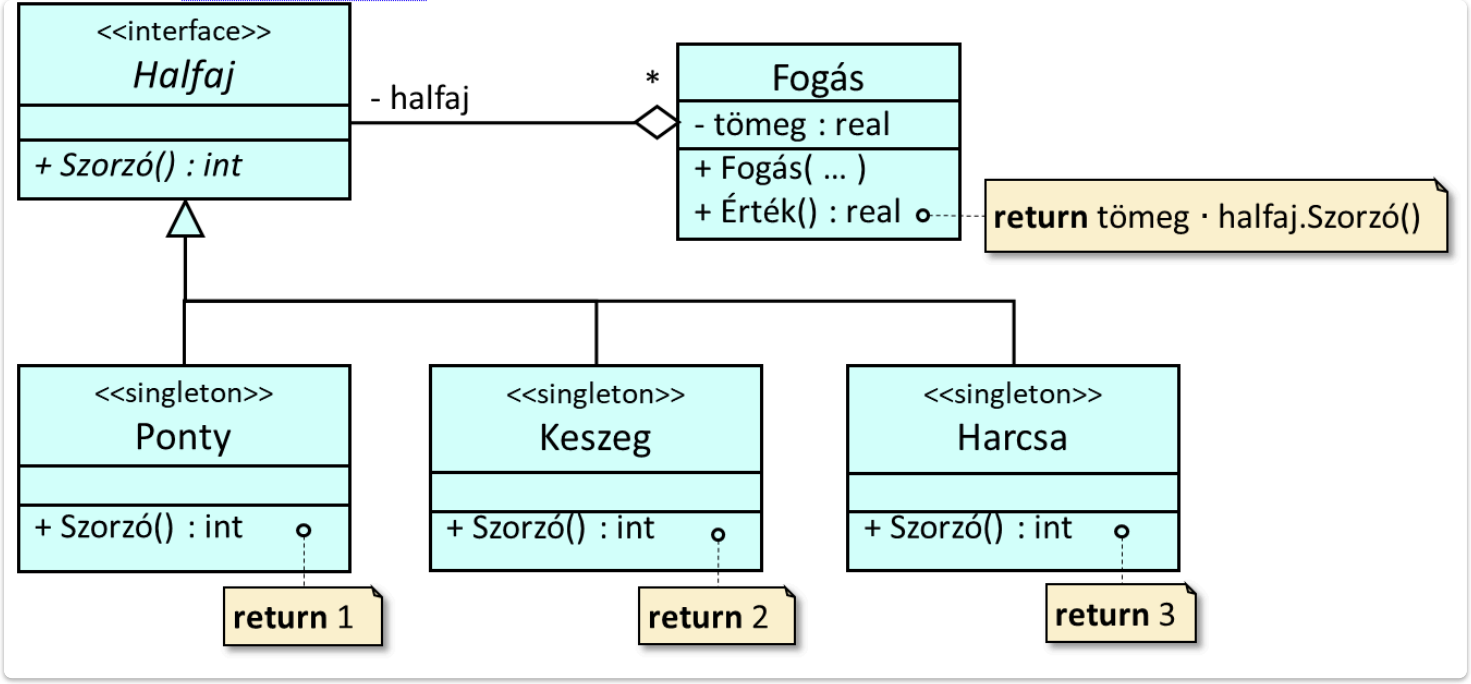
Felpopuláláshoz:

Szövetség osztály

- `Horgász? Keres(string név)` metódus
  - Név alapján visszaadja a horgászt a Szövetség tagjai közül ha van, ha nincs `null`-al tér vissza
- `Rendez()` metódus térjen vissza a rendezett Versennyel
  Horgász osztály
- Getter a `név` mezőhöz


Szövetség
tagok := < >
versenyek := < >
if ho in tagok then error endif
tagok.Add(ho)
+ Szövetség()
+ Belép(ho:Horgász)
+ Rendez(he:string, ke:DateTime)
◀ belép
l := SEARCH e in versenyek ((e.hely,e.kezdés)=(he,ke))
if l then error endif
versenyek.Add(new Verseny(this, he, ke))
- szöv
{implies}
szöv := sz
hely, kezdés := h, k
horgászok := < >
fogások := < >
név := n
fogások := < >
{getter}
* - versenyek {unique}
* - tagok {unique}
Verseny
- hely : string { getter }
- kezdés : DateTime { getter }
+ Verseny(sz:Szövetség,h:string,k:DateTime)
+ Regisztrál(ho:Horgász)
+ Feljegyez(f:Fogás)
* ◀ regisztrál *
Horgász
- név : string
+ Horgász(n:string)
+ Kifog(i:DateTime, f:HalFaj, t:real, v:Verseny)
{implies}
- horgászok {unique,getter}
if ho not in szöv.tagok
or ho in horgászok then error endif
horgászok.Add(ho)
- horgász
- verseny
if SEARCH e in fogások(e.időpont=i)
or this not in v.horgászok
then error endif
fog := new Fogás(i, t, f, v, this)
fogások.Add(fog); v.Feljegyez(fog)
nyilvántart ▶
if f.horgász not in horgászok
or f in fogások then error endif
fogások.Add(f)
- fogások {unique} *
Halfaj
- halfaj *
Fogás
- időpont : Time { getter }
- tömeg : real
+ Fogás(...)
* - fogások {unique}

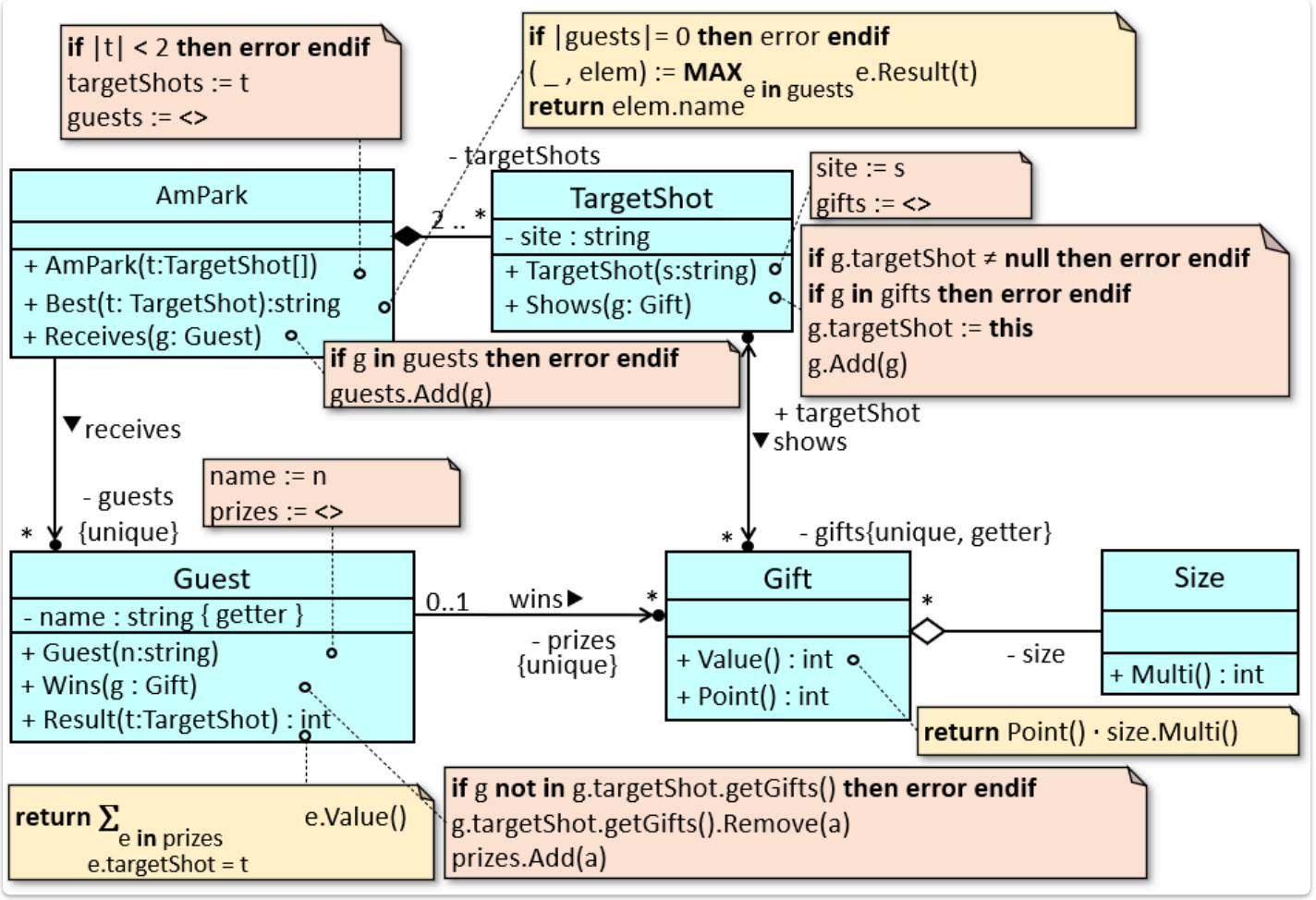

Szövetség
+ Szövetség()
+ Belép(ho:Horgász)
+ Rendez(he:string, ke:string)
+ Legjobb() : bool×Verseny
◀ belép
(l, max,elem) := MAX e in versenyek e.Összesít()
e.Mindharcsa()
return (l, elem)
- szöv
{implies}
return ∀SEARCH e in horgászok e.FogottHarcsát(this)
{getter}
* - versenyek {unique}
* - tagok {unique}
Verseny
- hely : string { getter }
- kezdés : DateTime { getter }
+ Verseny(sz:Szövetség,h:string,k:DateTime)
+ Regisztrál(ho:Horgász)
+ Feljegyez(f:Fogász)
+ Mindharcsa() : bool
+ Összesít() : real
* ◀ regisztrál *
Horgász
- név : string
+ Horgász(n : string)
+ Kifog(i:DateTime, f:HalFaj, ...)
+ FogottHarcsát(v:Verseny) : bool
{implies}
- horgászok {unique, getter}
return Σ e in fogások e.Érték()
- horgász
return SEARCH e in fogások e.hal.IsHarcsa() ∧ e.verseny=v
- verseny {getter}
nyilvántart ▶
- fogások {unique} *
<<abstract>>
Halfaj
+ Szorzó() : int
- halfaj *
{getter}
Fogás
- időpont : Time { getter }
- tömeg : real
+ Fogás(...)
+ Érték() : real
* - fogások {unique}
return tömeg · halfaj.Szorzó()



Kisbeadandó (10/10)

★ Céllövölde

Egy vidámpark különböző helyszíneken több céllövöldét is üzemeltet. A vidámpark vendégei akár több különböző céllövöldében is szerencsét próbálhatnak. Egy vendég egy céllövöldében minden sikeres találat esetén ajándékot nyer. Az ajándékokról később is tudható, hogy melyik céllövöldében nyerték. Egy ajándék típusa szerint lehet labda, műanyag figura, vagy plüss állat, és négy különböző méretben fordulhat elő: S, M, L, XL.
Egy-egy ajándék értékét úgy számolhatjuk ki, hogy a típusa után járó pontszámot (plüss állatra 3 pont, műanyag figurára 2 pont, labdára 1 pont) megszorozzuk a mérete után járó szorzóval (az S méret 1 pont, az M 2 pont, az L 3 pont, az XL 4 pont).
Nevezzük meg egy céllövölde legjobb céllövőjét, azaz azt a vendéget, aki által a céllövöldében nyert ajándékok összértéke a legnagyobb!


if |t| < 2 then error endif
targetShots := t
guests := <>
if |guests|= 0 then error endif
(_ , elem) := MAX e in guests e.Result(t)
return elem.name
- targetShots
AmPark
+ AmPark(t:TargetShot[])
+ Best(t: TargetShot):string
+ Receives(g: Guest)
2 .. *
TargetShot
- site : string
+ TargetShot(s:string)
+ Shows(g: Gift)
site := s
gifts := <>
if g.targetShot ≠ null then error endif
if g in gifts then error endif
g.targetShot := this
g.Add(g)
if g in guests then error endif
guests.Add(g)
receives
- guests
{unique}
*
+ targetShot
shows
name := n
prizes := <>
*
- gifts{unique, getter}
Guest
- name : string { getter }
+ Guest(n:string)
+ Wins(g : Gift)
+ Result(t:TargetShot) : int
0..1
wins
*
- prizes
{unique}
Gift
+ Value() : int
+ Point() : int
*
- size
Size
+ Multi() : int
return Point() · size.Multi()
return ∑ e.Value()
e in prizes
e.targetShot = t
if g not in g.targetShot.getGifts() then error endif
g.targetShot.getGifts().Remove(a)
prizes.Add(a)